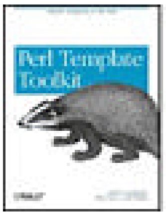
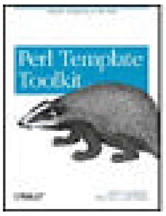


Perl Template Toolkit

By Darren Chamberlain, David Cross, Andy Wardley

Publisher: O'Reilly
Pub Date: December 2003
ISBN: 0-596-00476-1
Pages: 576
Slots: 1.0

Written by core members of the technology's development team, *Perl Template Toolkit* guides you through the entire process of installing, configuring, using, and extending the Template Toolkit. It begins with a fast-paced but thorough tutorial on building web content with the Template Toolkit, and then walks you through generating and using data files, particularly with XML. It also provides detailed information on the Template Toolkit's modules, libraries, and tools in addition to a complete reference manual.



- Table of Contents
- Index
- Reviews
- Examples
- Reader Reviews
- Errata
- Academic

Perl Template Toolkit

By Darren Chamberlain, David Cross, Andy Wardley

Publisher: O'Reilly
Pub Date: December 2003
ISBN: 0-596-00476-1
Pages: 576
Slots: 1.0

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Preface

Perl Template Toolkit is an introduction to the Template Toolkit. The Template Toolkit is most often used in the creation of web sites, but it can be used as a general text manipulation tool. It is a presentation management system that allows you to separate aspects of presentation from the rest of an application, in the same way that a database allows you to separate storage concerns.

The information in this book is based on Version 2.10 of the Template Toolkit, released in July 2003. The Template Toolkit will continue to evolve. Apart from bug fixes and minor updates, the Version 2.* branch will remain pretty much the same as it is now.

Version 3, expected sometime in 2004, will include new features and some changes to the internal architecture. However, it is an important requirement that new versions of the Template Toolkit are backward-compatible with previous versions wherever possible. Although the Template Toolkit may change in some subtle ways, the basic principles, syntax, and style are here to stay.

Audience

This book should be useful to anyone building and maintaining web sites or other complex content systems. No prior knowledge of Perl, the Template Toolkit, or HTML is required to apply the basic techniques taught in this book. Some of the more advanced topics require some degree of familiarity with the Perl programming language. Readers who understand the basic language constructs and idioms of Perl and who already know how to install and use Perl modules will have no trouble integrating the Template Toolkit into their existing or new projects. Some chapters talk about more specific application areas: HTML, web programming, XML, and SQL, for example. Experience in these areas will make the benefits of the Template Toolkit more readily apparent, but isn't required.

About this Book

This book is divided into 12 chapters and 1 appendix.

Chapter 1, *Getting Started with the Template Toolkit*, provides an introduction to the concepts of template processing in general and to the Template Toolkit in particular. It also covers how to install the Template Toolkit on your system and gives a brief tutorial on its use so that you can check that installation is successful. In case it isn't, the chapter also includes pointers to other sources of information on the Template Toolkit.

Chapter 2, *Building a Complete Web Site Using the Template Toolkit*, is a tutorial on building a web site using the Template Toolkit. It gives a brief overview of many of the features of the Template Toolkit that are covered in more detail later in the book.

Chapter 3, *The Template Language*, begins our detailed look at the Template Toolkit. In this chapter, we look at the syntax of the Template Toolkit's presentation language.

Chapter 4, *Template Directives*, covers the syntax and use of the many templating directives that can be used from the Template Toolkit.

Chapter 5, *Filters*, takes a look at filters. These are extensions to the Template Toolkit that allow you to filter your data in various ways before presenting it to your users. This chapter includes a guide to the various standard filters that are included with the Template Toolkit distribution.

Chapter 6, *Plugins*, looks at the Template Toolkit plugins. Plugins are another way to extend the Template Toolkit by giving your templates access to powerful external modules. This chapter includes a guide to the various standard plugins that are included with the Template Toolkit distribution.

Chapter 7, *Anatomy of the Template Toolkit*, looks under the covers of the Template Toolkit and examines in some detail how it all works from the inside.

Chapter 8, *Extending the Template Toolkit*, covers ways to extend the Template Toolkit by writing your own filters and plugins.

Chapter 9, *Accessing Databases*, looks in detail at writing templates that access data held in various different types of databases.

Chapter 10, *XML*, looks at using the Template Toolkit to generate XML. It also covers reading XML documents and using their contents from within your templates.

Chapter 11, *Advanced Static Web Page Techniques*, starts to put together everything we've covered in the previous chapters and shows how to build a static web site using the Template Toolkit.

Chapter 12, *Dynamic Web Content and Web Applications*, extends the example of the previous chapter to add dynamic content to your web site.

Appendix A, describes the configuration options for the Template Toolkit and Apache::Template.

Conventions Used in This Book

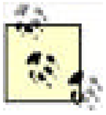
The following typographical conventions are used throughout this book:

`Constant width`

Used for Perl code, Template Toolkit directives, HTML, and code examples.

Italic

Used for filenames, URLs, hostnames, first use of terms, and emphasis.



Indicates a tip, suggestion, or general note.

Indicates a warning or caution.

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/perltt

To comment on or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, software, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

http://www.oreilly.com

Acknowledgments

This book would not be possible without the contribution and support of many individuals, including friends, family, and the hard-working folks at O'Reilly & Associates, Inc. All three of us wish to thank our production team and, in particular, our editor, Nathan Torkington, for his fine word wrangling and masterful cat herding. We would also like to thank our technical reviewers Chris Devers, Mark Fowler, Andrew Langmead, Martin Portman, and Simon Matthews for their detailed and insightful comments.

Andy Wardley

I'd like to start by thanking Dave, Darren, Nat, and the production team at O'Reilly for turning a bunch of words into a book. I would also like to thank Dom Millar for suggesting a badger for the front cover, and the design team for accommodating us with this beautiful animal.

The Template Toolkit has long since ceased to be a product of my work alone, if indeed it ever was. It owes its success to the dedicated efforts of an extended team of developers, testers, documenters, and users. At the time of this writing, the Template Toolkit documentation lists over sixty contributors who have donated their time and effort in different ways. Our collective thanks go to each of them: Chuck Adams, Stephen Adkins, Ivan Adzhubey, Mark Anderson, Bradley Baetz, Thierry-Michel Barral, Craig Barratt, Stas Bekman, Tony Bowden, Neil Bowers, Leon Brocard, Lyle Brooks, David Cantrell, Piers Cawley, Darren Chamberlain, Eric Cholet, Dave Cross, Chris Dean, Francois Desarmenien, Horst Dumcke, Mark Fowler, Michael Fowler, Axel Gerstmair, Dylan William Hardison, Perrin Harkins, Bryce Harrington, Dave Hodgkinson, Harald Joerg, Colin Johnson, Vivek Khera, Rafael Kitover, Ivan Kurmanov, Hans von Lengerke, Jonas Liljegren, Simon Luff, Paul Makepeace, Gervase Markham, Simon Matthews, Robert McArthur, Craig McLane, Leslie Michael Orchard, Eugene Miretskiy, Tatsuhiko Miyagawa, Keith G. Murphy, Chris Nandor, Briac Pilpré, Martin Portman, Slaven Rezic, Christian Schaffner, Randal L. Schwartz, Paul Sharpe, Ville Skyttä, Doug Steinwand, Michael Stevens, Drew Taylor, Swen Thuemmler, Richard Tietjen, Stathy G. Touloumis, Jim Vaughan, Simon Wilcox, and Chris Winters.

Special thanks are due to Simon Matthews, who has been using and abusing the Template Toolkit and its predecessors from the very start. Countless pints of Guinness have been consumed through long evenings spent discussing the design, development, and general direction of the project. I would also like to thank Martin Portman for the many enjoyable hours we have spent at the whiteboard, engaged in animated conversation and frantic scribbling. Many of the important TT design decisions have been thrashed out in the company of Simon and Martin. Their efforts and input continue to be gratefully received.

I would also like to thank all my other friends and colleagues of past and present at Knowledge Pool, Canon Research Centre Europe, and Fotango, many of whom are listed above. Each of these organizations and the people within them have played important roles in the evolution of the Template Toolkit.

Finally I would like to thank my wife, Sheila, and son, Ben, for their love, patience, and understanding. Writing this book ate up far too much of the time that should have been spent with you.

Darren Chamberlain

I'd like to thank my wife and kids for their help and support, and for being so understanding of the time I've spent writing instead of mowing the lawn or playing. This wouldn't have been possible for me otherwise, and I appreciate it more than they know.

Thanks to *Boston.com* for having the incredibly sane policy of using the best tool for the job, which means letting me use the Template Toolkit for so many things; to Andrew Langmead, Chris Devers, and Mike Melillo for proofreading, fact-checking, and putting up with me in general; and to Marc Lavallee, for introducing me to TT in first place.

Thanks to Andy for writing the Template Toolkit, which is as fine and versatile a piece of software as I've seen in a long time. Andy, Dave, and Nat have all been great—I hope I get to work them again.

And, of course, thanks to everyone who buys the book and keeps O'Reilly (and their fine authors!) afloat.

David Cross

I'd like to thank Andy for developing the Template Toolkit and both Darren and Andy for making the process of writing this book as much fun as it was.

Thanks to the members of the London.pm/TT cabal for first introducing me to the Template Toolkit and convincing me that it was the only templating system that I needed to look at.

Thanks to the various clients and employers who have put up with me leaving the office on time to get on with writing the book. Particular thanks should go to the people at Bibliotech who took pity on me trying to write and work simultaneously and resolved the situation by making me redundant.

Most of this book has been written while listening to music. I've found that I write best when listening to either Billy Bragg or any combination of the Waterson/Carthy clan, so thanks to them.

Thank you to Joss Whedon for cancelling "Buffy the Vampire Slayer" while I was working on this book and giving me one less reason to avoid writing.

Thank you to the various friends and family who have ensured that I still have a social life despite my seeming to do my utmost to avoid it.

Thanks, of course, to my parents Jean and John, and to my wife Gill. Their love and support make it all much easier.

Chapter 1. Getting Started with the Template Toolkit

The Template Toolkit is an all-Perl template processing system. Unlike many other Perl templating systems, it is as proficient at producing HTML as it is at producing XML, PDF, or any other output format. It has its own simple templating language, so templates can be written and edited by people who do not know Perl. And it supports command-line, modular, CGI, and mod_perl operation.

In this chapter, we compare the Template Toolkit to other templating systems such as HTML::Mason and HTML::Template, describe how to install it, then show you what templates look like and how to process them. The goal of this chapter is to get you started—you should be able to install the Template Toolkit, write and understand basic templates, and know how to process the templates from the command line, from Perl programs, and from mod_perl.

1.1 What the Template Toolkit Does

The basic task of a template processor is to output some amount of changeable data surrounded by some unchanging data. A simple example of this is a form letter, where the same text is sent to many different people, with just the name, address, and other personal details being changed. The template contains the fixed ("boilerplate") text together with special markup tags indicating where the variable pieces of data are to be placed.

Example 1-1 shows a template for such a form letter. This template is marked up using the default style of the Template Toolkit, where the [% ... %] tags indicate variable values. Everything else is fixed text that passes through the processor untouched.

Example 1-1. A form letter template, destruction.tt

```
People of [% planet %], your attention please.


This is [% captain %] of the
Galactic Hyperspace Planning Council.


As you will no doubt be aware, the plans
for development of the outlying regions
of the Galaxy require the building of a
hyperspatial express route through your
star system, and regrettably your planet
is one of those scheduled for destruction.


The process will take slightly less than
[% time %].


Thank you.
```

A template processor takes the template, together with a list of the variable data to be included in the letter, and produces a finished letter. The Template Toolkit provides *tpage* for doing just that from the command line. Pass the name of the template file to *tpage* as a command-line option, along with any number of --define options to provide values for variables. If the preceding template is stored in the *destruction.tt* file in the current directory, the following command processes it:

```
$ tpage --define planet=Earth \
>       --define captain="Prostetnic Vogon Jeltz" \
>       --define time="two of your earth minutes" \
>       destruction.tt
```

The output this generates is shown in Example 1-2.

Example 1-2. Form letter generated by template in Example 1-1

```
People of Earth, your attention please.


This is Prostetnic Vogon Jeltz of the
```

```
Galactic Hyperspace Planning Council.

As you will no doubt be aware, the plans
for development of the outlying regions
of the Galaxy require the building of a
hyperspatial express route through your
star system, and regrettably your planet
is one of those scheduled for destruction.

The process will take slightly less than
two of your earth minutes.

Thank you.
```

Process the same template a few thousand times with different sets of data and you have the entire basis of the junk-mail industry. Or a Vogon Constructor Fleet.

This book is a good example of a more complex template. All O'Reilly books conform to one of a small number of formats. They all have similar sets of front matter (title page, publication information, table of contents, and preface), followed by the actual chapters, some (optional) appendices, an index, and finally the colophon. Templates that define the look of all of these parts are defined in the publication system, and the data for a particular book is formatted to conform to those rules. If someone decides to change the font used for the chapter titles in forthcoming books, he need only change the setting in the template definition.

Another way to look at a template processor is as a tool for separating *processing* from *presentation*. For example, a company sales report is probably created from data stored in a database. One way to create the report would be to extract the required data into a spreadsheet and then do calculations on the data to produce the information required. The spreadsheet could then be printed out or emailed to the required recipients.

Although you can use templates to generate any kind of text document, the most common use is to generate HTML pages for web content. The whole genre of template processing systems has matured rapidly in less than a decade, particularly within the Perl community, in response to the demands of people struggling to build and maintain ever more complex content and applications for their web sites.

Templates help in a number of ways. The most obvious benefit is that they can be used to apply a consistent look and feel to all the pages in a web site to achieve a common branding. You can use a template to add standard headers, footers, menus, and other user interface components as easily as the Hyperspace Planning Council ruthlessly adds a few lines of Vogon poetry to every planet destruction order, just to rub salt into the wounds.

This is just the tip of the iceberg. In addition to the use of variables, the Template Toolkit provides a number of other directives that instruct it to perform more complex processing actions, such as including another template, repeating a section of markup for different pieces of data, or choosing a section to process based on a particular condition. Example 1-3 illustrates some of these directives in action.

Example 1-3. Loops, conditions, and processing instructions in a template

```
[% FOREACH order IN council.demolition.orders %]
  [% PROCESS header %]

  [% IF order.destruction %]
    As you will no doubt be aware, the plans
    for development of the outlying regions
    of the Galaxy require the building of a
    hyperspatial express route through your
    star system, and regrettably your planet
```

```
    is one of those scheduled for destruction.
  [% ELSE %]
    Our representatives will be visiting your
    star system within the next few weeks,
    and would like to invite you to a reading of
    Vogon Poetry. Attendance is mandatory.
    Resistance is useless!
  [% END %]

  [% PROCESS footer %]

  [% PROCESS poetry/excerpt
       IF today.day = = 'Vogonsday'
  %]
[% END %]
```

We explain the purpose of these directives later in this chapter, and show examples of the different ways they can be used throughout the rest of the book. For now, you can probably work out what they do from their names.

The Template Toolkit is just one example of a template processor. Although it's written in Perl, you don't actually need to know any Perl to use it. The presentation language that it provides is intentionally simple, regular, and easy to understand and use. This makes it simple for web designers and other nonprogrammers to use it without first having to get to grips with Perl. The Template Toolkit provides language features and off-the-shelf plugin modules that allow you to perform many common tasks, including CGI programming, manipulating XML files, and accessing SQL databases.

If you do know Perl, however, you'll be able to get more out of the Template Toolkit by writing custom functions and extensions to handle the specifics of your particular application. The good news for Perl programmers is that the Template Toolkit allows you to separate Perl code clearly from HTML templates. This clear separation means that you don't have to wade through pages of HTML markup to find the part of your web application that needs attention. It allows you to concentrate on one thing at a time, be it the HTML presentation or the Perl application, without having the other aspects in your face and under your feet. It makes both your HTML templates and Perl code more portable and reusable, and easier to read, write, and maintain.

1.2 The Templating Ecosystem

At least half a dozen mature and respected templating systems are available for Perl. The best-known and best-supported template processors include the following:

Text::Template

Text::Template is a library for generating form letters, building HTML pages, or filling in templates generally. A template is a piece of text that has little Perl programs embedded in it here and there. When you fill in a template, you evaluate the little programs and replace them with their values. These programs are written in Perl: you embed Perl code in your template, with { at the beginning and } at the end. If you want a variable interpolated, you write it the way you would in Perl. If you need to make a loop, you can use any of the Perl loop constructions. All the Perl built-in functions are available.

Text::Template is available from http://www.plover.com/~mjd/perl/Template/ or from CPAN (http://search.cpan.org/dist/Text-Template/).

HTML::Template

HTML::Template attempts to make using HTML templates easy and natural. It extends standard HTML with a few HTML-like tags, and enforces the divide between design and programming by restricting what a template is capable of doing. By limiting the programmer to using just simple variables and loops in the HTML, the template remains accessible to designers and other non-Perl people. The use of HTML-like syntax goes further to make the format understandable to others.

HTML::Template is available from CPAN (http://search.cpan.org/dist/HTML-Template/).

HTML::Mason

HTML::Mason is a Perl-based web site development and delivery system. Mason allows web pages and sites to be constructed from shared, reusable building blocks called *components*. Components contain a mix of Perl and HTML, and can call each other and pass values back and forth like subroutines. Components increase modularity and eliminate repetitive work: common design elements (headers, footers, menus, logos) can be extracted into their own components where they need be changed only once to affect the whole site. Mason also includes powerful filtering and templating facilities and an HTML/data caching model.

HTML::Mason is available from http://www.masonhq.com/ and CPAN (http://search.cpan.org/dist/HTML-Mason/).

HTML::Embperl

Embperl gives you the power to embed Perl code in your HTML documents, and the ability to build your web site out of small reusable objects in an object-oriented style. You can also take advantage of all the usual Perl modules (including DBI for database access), use their functionality, and easily include their output in your web pages.

Embperl has several features that are especially useful for creating HTML, including dynamic tables, form field processing, URL escaping/unescaping, session handling, and more.

Embperl is a server-side tool, which means that it's browser-independent. It can run in various ways: under mod_perl, as a CGI script, or offline.

HTML::Embperl is available from http://www.ecos.de/ or CPAN (http://search.cpan.org/dist/HTML-Embperl/).

Apache::ASP

Apache::ASP provides an Active Server Pages port to the Apache web server with Perl scripting only, and enables development of dynamic web applications with session management and embedded Perl code. Apache::ASP also provides many powerful extensions, including XML taglibs, XSLT rendering, and new events not originally part of the ASP API.

Apache::ASP is available from CPAN (http://search.cpan.org/dist/Apache-ASP/).

The Template Toolkit attempts to offer the best features of these modules, including separation of Perl from templates and applicability beyond HTML.

1.2.1 The Template Toolkit Is for More Than HTML

The Template Toolkit is a generic template processing system that will process any kind of document for use in any environment or application. Many other template systems were designed specifically to create HTML pages for web content. In some cases, that is all the system can be used for. In others, it is possible (with varying degrees of difficulty) to use the system in a non-web environment.

The Template Toolkit was originally designed to help Andy create his web site, but he was careful to ensure that it was just as usable outside of that environment. As a result, there is nothing within the Template Toolkit that assumes it is being used to generate HTML. It is equally at home creating any other kind of data.

1.2.2 The Template Toolkit Lets You Choose Your Separation

Template Toolkit doesn't prescribe any particular methodology or framework that forces you to use it in a certain way. Some modules (for example, `HTML::Template`) enforce a very strict interpretation of template processing that intentionally limits what can be done in a template to accessing variables and using simple conditional or looping constructs. Others (such as `HTML::Mason` and `HTML::Embperl`) use embedded Perl code to allow any kind of application functionality to be incorporated directly into the templates.

The Template Toolkit gives you the best of both worlds. It has a powerful data engine (the *Stash*) that does all the hard work of mapping complex data structures from your Perl code, configuration files, SQL databases, XML files, and so on, into template variables that are accessed by a simple and uniform dotted notation (e.g., `person.surname`). You can use this to keep your templates simple without limiting the complexity or functionality of the systems that put data into the templates.

At the opposite end of the spectrum, the Template Toolkit also allows you to embed Perl code directly in your templates. We don't normally encourage this because it tends to defeat the purpose of having a template processing system in the first place. Because this is the exception rather than the norm, template processors must set the `EVAL_PERL` option to embed Perl code in the template (it is disabled by default). We look at how to set options later in this chapter.

Template Toolkit also lets you work between the two extremes. It provides a rich set of language features (*directives*) that allow you to add complex functionality to your templates without requiring you to embed Perl code. It also has a powerful *plugin* mechanism that allows you to load and use Perl modules to extend the functionality in any way you can imagine.

In short, the Template Toolkit allows you to take a modular approach to building your web site or other document system, but doesn't enforce it. Sometimes you want to build a complex and highly structured system to run a web site. Other times you just want to roll up a quick all-in-one template to generate a report from a database. The Template Toolkit encourages whatever approach is most appropriate to the task at hand.

1.2.3 Nonprogrammers Can Maintain Templates

Template Toolkit's template language is designed to be as simple as possible without being too simple. The dotted notation makes accessing variables far less daunting than in Perl. For example:

```
$person->{surname}   # Perl

person.surname       # Template Toolkit
```

This hides the underlying implementation details from the template designer. In the previous example, the Perl syntax implies that `$person` is a reference to a hash array containing a `surname` value. However, you might one day decide to implement `$person` as an object with a `surname( )` method:

```
$person->surname( )  # Perl

person.surname       # Template Toolkit
```

The Perl code requires a different syntax but the Template Toolkit code stays the same. This lets you change the underlying implementation at any time without having to change the templates. As long as the data is laid out in the same way (i.e., don't change `surname` to `last_name`), it doesn't really matter what data structures are used, or whether they are precomputed, fetched from a database, or generated on demand.

This uniform syntax also means that your template designers can remain blissfully ignorant of the difference between a hash array and an object. They don't have to worry about any confusing syntax and can concentrate on the task at hand of presenting the data nicely. This makes the template language as friendly as possible for people who aren't already Perl programmers.

The general rule is to use Perl for programming and the Template Toolkit for presentation. But again, it's not mandatory, so you're still free to bend (or break) the rules when you really need to.

1.2.4 The Template Toolkit Is Easy to Extend

The Template Toolkit is designed to be easy to extend. If it doesn't already do what you want, there's a good chance you can reimplement a small part of it to change it to do what you what. The object-oriented architecture of the Template Toolkit makes this process relatively straightforward, and there are programming hooks throughout the system to give you as much flexibility as possible.

A number of plugins exist for the Template Toolkit, and we cover them in Chapter 6. They are designed to give templates convenient control over things such as HTML tables, database connections, and CGI parameters.

1.3 Installing the Template Toolkit

At any one time you can download from the Web at least two possible versions of the Template Toolkit: a stable version and a developer version. The stable version has a version number such as 2.10, and has been widely tested before release. The developer versions have version numbers such as 2.10a, and typically have bug fixes and early implementations of new features. Generally, you should install the latest stable release.

1.3.1 Downloading

The Template Toolkit is available from the Comprehensive Perl Archive Network (CPAN). You can always download the most recent stable version of the Template Toolkit from http://search.cpan.org/dist/Template-Toolkit/ (which is where most people download it).

In addition, a web site is dedicated to the Template Toolkit. Located at http://www.template-toolkit.org, this site offers the latest stable version, as well as a number of other goodies such as native packages of the Template Toolkit for Debian GNU/Linux, Mac OS X (for installation using Fink), and Microsoft Windows (for installation using ActiveState's Perl Package Manager).

You can also get developer versions of the Template Toolkit from the web site. Normally, you need to download only the current stable version, but if you come across a bug that isn't fixed in the CPAN version, you may need to use a developer release.

If a developer release isn't cutting-edge enough for you, the web site contains information on how to get access to the CVS repository, which is where the very latest versions of the Template Toolkit source code are kept. If you want to add functionality to the Template Toolkit or have found a bug that you can fix, and you want your patch to be accepted by Template Toolkit developers, you should make your changes against the current CVS HEAD.

1.3.2 Installing

Installing the Template Toolkit is like installing any other Perl module (see *perlmodinstall(1)* for platform-specific details). The basic idea is as follows:

```
$ perl Makefile.PL

$ make

$ make test

$ make install
```

A few optional modules and pages of documentation come with the Template Toolkit, and how much of that gets installed is controlled by arguments to perl Makefile.PL. Run perl Makefile.PL TT_HELP to get the following full list of options:

```
The following options can be specified as command-line
arguments to 'perl Makefile.PL'.  e.g.,

  perl Makefile.PL TT_PREFIX=/my/tt2/dir TT_ACCEPT=y

  TT_PREFIX      installation prefix     (/usr/local/tt2)
  TT_IMAGES      images URL              (/tt2/images)
  TT_DOCS        build HTML docs         (y)
  TT_SPLASH      use Splash! for docs    (y)
  TT_THEME       Splash! theme           (default)
  TT_EXAMPLES    build HTML examples     (y)
  TT_EXTRAS      install optional extras (y)
  TT_XS_ENABLE   Enable XS Stash         (y)
  TT_XS_DEFAULT  Use XS Stash by default (y)
```

```
  TT_DBI          run DBI tests         (y if DBI installed)


  TT_LATEX        install LaTeX filter  (y if LaTeX found)
  TT_LATEX_PATH      path to latex      (system dependant)
  TT_PDFLATEX_PATH   path to pdflatex   ( "     "     " )
  TT_DVIPS_PATH      path to dvips      ( "     "     " )


  TT_QUIET        no messages           (n)
  TT_ACCEPT       accept defaults       (n)


By default, the Makefile.PL runs in interactive mode,
prompting for confirmation of the various configuration
options.  Setting the TT_ACCEPT option causes the default
value (possibly modified by other command line options)
to be accepted.  The TT_QUIET option can also be set to
suppress the prompt messages.
```

The make test step is important, especially if you're using a developer release or version from CVS. Over 2,000 tests are provided with the Template Toolkit to ensure that everything works as expected, and to let you know about any problems that you might have. It takes no more than a minute or so to run the tests, and they can save you a great deal of debugging time in the unlikely event that something is wrong with your installation.

Test failures don't necessarily indicate that something is fatally wrong. A serious problem causes nearly all of the tests to fail, although we haven't heard of that happening to anyone for quite some time. More often than not, errors raised in the test suite come from plugin modules whose external Perl modules are not installed on your system or are the wrong version.

This kind of problem is rarely serious. At worst, it may mean that a particular plugin doesn't work as expected—or at all—but that won't stop the rest of the Template Toolkit from doing its job. You can usually solve the problem by installing the latest version of any dependent modules. If you are unsure about whether a particular test failure is significant, ask on the mailing list, or check the mailing list archives mentioned in Section 1.4.3, later in this chapter. Major problems tend to be reported by many people.

The *README* and *INSTALL* files in the Template Toolkit distribution directory provide further information about running the test suite and what to do if something goes wrong.

1.4 Documentation and Support

In this section, we take a look at the support that is available for the Template Toolkit.

1.4.1 Viewing the Documentation

The Template Toolkit comes with an incredible amount of documentation. The documentation is supplied in the standard Perl Plain Old Documentation (POD) format. Once you have installed the Template Toolkit, you can see any of the documentation pages using *perldoc* or *man*, just as you can with any other Perl module:

```
$ perldoc Template          # should always work

$ man Template              # does not work everywhere
```

During the Template Toolkit installation procedure you are offered the chance to install HTML versions of the documentation. The default location for the installation of these files is */usr/local/tt2* under Unix and *C:/Program Files/Template Toolkit 2* under Win32. The installation procedure prompts for alternate locations.

If you are running a web server on your local machine, you can configure it to know where these files are. For example, you might put the contents of Example 1-4 in the *httpd.conf* for an Apache web server.

Example 1-4. Apache configuration directives to view Template Toolkit documentation

```
# TT2

Alias /tt2/images/   /usr/local/tt2/images/

Alias /tt2/docs/     /usr/local/tt2/docs/html/

Alias /tt2/examples/ /usr/local/tt2/examples/html/


<Directory /usr/local/tt2/>

    Options Indexes

    AllowOverride None

    Order allow,deny

    Allow from all

</Directory>
```

You can now access the locally installed documentation by pointing your browser at *http://localhost/tt2/docs*. For more information on configuring your web server, see the *INSTALL* file that comes with the Template Toolkit.

The complete documentation set is also available online at the Template Toolkit web site. You can find it at http://www.template-toolkit.org/docs.html.

1.4.2 Overview of the Documentation

A large number of manual pages come with the Template Toolkit. Here is a list of some of the most useful ones:

Template

The manual page for the Template module, the main module for using the Template Toolkit from Perl.

Template::Manual

An introduction and table of contents for the rest of the manual pages.

Template::Manual::Intro

A brief introduction to using the Template Toolkit. Not unlike this chapter.

Template::Manual::Syntax

The syntax, structure, and semantics of the Template Toolkit directives and general presentation language. Chapter 3 covers this aspect.

Template::Manual::Variables

A description of the various ways that Perl data can be bound to variables for accessing from templates. Chapter 3 also has the details.

Template::Manual::Directives

A reference guide to all Template Toolkit directives, with examples of usage. See Chapter 4.

Template::Manual::VMethods

A guide to the virtual methods available to manipulate Template Toolkit variables. These are also covered in Chapter 3.

Template::Manual::Filters

A guide to the various standard filters that are supplied with the Template Toolkit. See Chapter 5.

Template::Manual::Plugins

A guide to the various standard plugins that are supplied with the Template Toolkit. See Chapter 6.

Template::Manual::Internals

An overview of the internal architecture of Template Toolkit. See Chapter 7.

Template::Manual::Config

Details of the configuration options that can be used to customize the behavior and extend the features of the Template Toolkit. This is covered in the Appendix.

Template::Manual::Views

A description of dynamic views—a powerful but experimental feature in the Template Toolkit. The use of views is covered briefly in Chapter 9.

Template::Tutorial

An introduction and table of contents to the tutorials that are distributed with Template Toolkit. Currently there are two. Template::Tutorial::Web is a quick start to using the Template Toolkit to create web pages, and Template::Tutorial::Datafile is a guide to creating datafiles in various formats (particularly XML). See Chapter 2 and Chapter 10 for more information about using the Template Toolkit to generate web pages and XML, respectively.

Template::Library::HTML and Template::Library::Splash

Two guides to using libraries of user interface components (widgets) for creating HTML with the Template Toolkit.

Template::Modules

A list of the various Perl modules that make up the Template Toolkit. Each module has its own manual page.

1.4.3 Accessing the Mailing List

If you can't find the answer to your questions in any of the documentation, you can always turn to the mailing list set up for discussion of the Template Toolkit. You can subscribe to the mailing list at: http://template-toolkit.org/mailman/listinfo/templates. All previous posts are archived at: http://template-toolkit.org/pipermail/templates.

Activity on the list is moderate (around 100 messages per month) and many of the Template Toolkit experts are on the list.

1.5 Using the Template Toolkit

The rest of this chapter provides a brief introduction to using the Template Toolkit. We look at the structure and syntax of templates, showing how variables and directives are embedded in plain text and expanded by the template processing engine. We talk about some of the different kinds of directives that the Template Toolkit provides, what they're used for, and how you go about using them.

We start by looking at the four main ways of using the Template Toolkit to process templates: from the command line using the *tpage* and *ttree* programs; from a Perl script using the Template module; and in a mod_perl-enabled Apache web server using the Apache::Template module.

1.5.1 tpage

The *tpage* program provides a quick and easy way to process a template file from the command line. The name of the template file is specified as a command-line argument. This is processed through the Template Toolkit processing engine, and the resultant output is printed to STDOUT:

```
$ tpage infile
```

You can use the > file redirect operator (if your operating system supports it, or something similar) to save the output into another file:

```
$ tpage infile > outfile
```

In this example, the input template, *infile*, is processed by *tpage* with the output saved in *outfile*. If something goes wrong and the template can't be processed (for example, if the input file specified doesn't exist or contains an invalid template directive or markup error), an error is printed to STDERR, and *tpage* exits without generating any standard output.

The following shows what happens if we try and coerce *tpage* into processing a file, *nosuchfile*, which doesn't exist on our system:

```
$ tpage nosuchfile

file error - nosuchfile: not found at /usr/bin/tpage line 60.
```

tpage offers just one command-line option, --define, which allows you to provide values for template variables embedded in the document. We saw this earlier in Example 1-1 where it processed the Vogon form letter:

```
$ tpage --define planet=Earth \

>       --define captain="Prostetnic Vogon Jeltz" \

>       --define time="two of your earth minutes" \

>       destruction.tt
```

The *tpage* program is ideal for simple template processing such as this, where nothing more is required than the ability to insert a few variable values. More complex tasks need the *ttree* program or custom programs using the Template module.

However, there is one last *tpage* trick we can show you. If you don't provide *tpage* with the name of a template file, it reads it from STDIN. This allows you to use it as Unix-style pipeline filter. For example, if the output of the *mktemplate* program is a Template Toolkit template, the following command can be used to pipe it into *tpage* to have it processed:

```
$ mktemplate | tpage
```

Invoking *tpage* by itself, with no arguments and no piped input, starts it in interactive mode. In this case, *tpage* sits and waits for you to type in a source template. This can be very useful for trying out small snippets of template syntax to see what they do.

Here's an example:

```
$ tpage

[% subject = 'cat'

   object  = 'mat'

%]

The [% subject %] sat on the [% object %].
```

```
^D

The cat sat on the mat.
```

The first line invokes *tpage* from the command line. The next three lines are the body of the template in which we type, followed by the end-of-file (EOF) character telling *tpage* that we're done. On Unix systems, this is Ctrl-D, shown in the example as ^D. On Microsoft Windows platforms, Ctrl-Z is the EOF character.

The rest of the example shows the output generated by *tpage* from processing the template. The cat is sitting on the mat, and everything is working as expected.

1.5.2 ttree

The *ttree* program offers many more features and options than *tpage* does. The first major difference is that *ttree* works with entire directories of templates rather than with single files. If you're using the Template Toolkit to build a web site, for example, you can point *ttree* at a directory of source templates to process them all, saving the generated HTML pages to corresponding files in an output directory.

The following example shows how you could invoke *ttree* to process all the templates in the *templates* directory (containing the files *cat* and *dog* for the purpose of this example), and save the generated output in files of the same name, which are located in the *output* directory:

```
$ ttree -s templates -d output -v
```

The -s option defines the source directory for templates, and -d defines the destination directory for output files. The -v (verbose) option causes *ttree* to print a summary of what it's doing to STDERR.

Here's an example of the kind of information generated by the -v option:

```
ttree 2.63 (Template Toolkit version 2.10)


      Source: templates

 Destination: output

Include Path: [  ]

      Ignore: [  ]

        Copy: [  ]

      Accept: [ * ]


  + dog

  + cat
```

This is a summary of the processing options, including the Source and Destination that we provided as the -s and -d command-line options. The *dog* and *cat* files are listed as the two files that *ttree* found in the *templates* directory. The + characters indicate that both files were successfully processed, creating *dog* and *cat* files in the *output* directory.

Now that these templates have been processed, *ttree* will not process them again until they are modified or the corresponding output file is deleted. By looking at the file modification times of the source template and destination file, *ttree* can decide which templates have changed and which have not. It saves time by processing only those that have changed.

If you run the same *ttree* command again, you see that the + characters to the left of the filenames have changed to - characters:

```
ttree 2.63 (Template Toolkit version 2.10)


      Source: templates

 Destination: output

Include Path: [  ]

      Ignore: [  ]

        Copy: [  ]
```

```
    Accept: [ * ]


  - dog                  (not modified)

  - cat                  (not modified)
```

These - characters indicate that the template files were not processed this time, with the reason given in parentheses to the right. This can save a great deal of time when building large document systems using templates (e.g., a typical web site) in which only a few pages change at any one time.

The -a option forces *ttree* to process all templates, regardless of their modification times:

```
$ ttree -a
```

A second benefit of *ttree* is that it offers numerous options for changing its behavior. Adding a standard header and footer to each page template, for example, is as easy as setting the relevant option:

```
$ ttree -s templates -d output -v \

>       --pre_process=header \

>       --post_process=footer
```

The number of options can be overwhelming at first, but in practice, only a few are used on a regular basis. To avoid having to always use the command line to specify options—something that can quickly become cumbersome and error prone, especially if you are using more than a few—*ttree* allows you to use configuration files to define all the options for a particular web site or other document system. You can then invoke *ttree*, passing the name of the configuration file using the -f option:

```
$ ttree -f /home/dent/web/ttree.cfg
```

Example 1-5 shows a sample *ttree* configuration file.

Example 1-5. A sample ttree configuration file, ttree.cfg

```
src  = /home/dent/web/templates

dest = /home/dent/web/html

lib  = /home/dent/web/lib



pre_process  = header

post_process = footer



verbose
```

In the configuration file, the -s and -d options are represented by the src and dest options. We also added a lib option (-l on the command line), which tells *ttree* about an additional library directory where our *header* and *footer* templates are found.

Setting up *ttree* is a little more involved than using *tpage*, but the effort quickly pays off in the time it saves you. We look at *ttree* in detail in Chapter 2, showing everything from first-time use through writing and managing configuration files.

1.5.3 The Template Module

Both *tpage* and *ttree* use the Template Perl module to do the dirty work of processing templates. As it happens, the Template module doesn't actually do much in the way of dirty work itself, but delegates it to other modules in the Template Toolkit with exotic names such as Template::Service, Template::Context, Template::Provider, and Template::Stash. The Template module provides a simple interface for using the Template Toolkit from Perl so that you don't have to worry about the complex underlying functionality that makes it work. Chapter 7 goes into greater detail about what lurks beneath the hood of the Template Toolkit, but for now we cover just the basics.

If you are already a Perl hacker experienced in using modules, the Template manpage gives you an executive summary to get you quickly up to speed. If you're not a Perl hacker but would like to be, *Learning Perl*, Third Edition, by Randal Schwartz and Tom Phoenix (O'Reilly) is a good place to start.

However, you don't need to know any Perl to use the Template Toolkit. Thanks to the *tpage* and *ttree* programs, you can build your entire web site or other template-based document system without ever having to write a line of Perl code. Nevertheless, it's useful to have a basic understanding of how the Template module is used in Perl programs (including *tpage* and *ttree*), even if you never plan on using the module. Also, certain features are accessible only through Perl (for example, the ability to define a subroutine to return the value for a variable), so there is a good chance that sooner or later you will want or need those Perl-specific features.

Example 1-6 shows a simple Perl program for processing the *destruction.tt* template from Example 1-1.

Example 1-6. A Perl program for processing the Vogon form letter template

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;

my $tt    = Template->new( );
my $input = 'destruction.tt';
my $vars  = {
    planet  => 'Earth',
    captain => 'Prostetnic Vogon Jeltz',
    time    => 'two of your earth minutes',
};

$tt->process($input, $vars)
    || die $tt->error( );
```

The first line defines the path to the Perl interpreter on your system. This is very much a Unix-specific convention. On a Windows machine, for example, this line is not relevant or required.

In the first block, we enable Perl's strict and warnings pragmata and then load the Template module:

```
use strict;
use warnings;
use Template;
```

> It is good Perl style to include use strict; and use warnings; at the top of every program, or to invoke Perl with the -w switch instead of use warnings; for versions of Perl earlier than 5.6.0. These two precautions will catch many common programming and typographical errors, and warn you about any questionable practices. Perl examples in this book may omit them for brevity, but you should always include them in any nontrivial chunk of code.

The next line creates a new Template object and assigns it to the $tt variable:

```
my $tt = Template->new( );
```

We store the name of the template to be processed in the $input variable and define some template variables in $vars:

```
my $input = 'destruction.tt2';

my $vars  = {

    planet  => 'Earth',

    captain => 'Prostetnic Vogon Jeltz',

    time    => 'two of your earth minutes',

};
```

Then we invoke the process( ) method against the $tt template object to process the source template:

```
$tt->process($input, $vars)

    || die $tt->error( );
```

The name of the source template file, here stored in the $input variable, is passed as the first argument, followed by a reference to a hash array of template variables, defined in $vars.

The process( ) method processes the template and returns a true value to indicate success. The output is printed to STDOUT by default so that you see it scrolling up your screen when you run the program.

If an error occurs, the process( ) method returns false. In this case, we call the error( ) method to find out what went wrong and report it as a fatal error using die. An error can be returned for a number of reasons, such as the file specified could not be found, had embedded directives containing illegal syntax that could not be parsed, or generated a runtime error while the template was being processed.

1.5.3.1 Template configuration options

We mentioned the --pre_process and --post_process options when using *ttree* earlier. Now we can see how these are used in the underlying Perl implementation.

Configuration options are passed to the new( ) constructor method as a reference to a hash, as shown in Example 1-7. The Template module expects options to be provided in uppercase, so the options for *ttree* translate to the PRE_PROCESS and POST_PROCESS options for the Template module. We also set the INCLUDE_PATH option to indicate the location of the source and library templates, which *ttree* provides from the src (or -s) and lib (or -l) options. These are provided as a reference to a list of the two directory paths.

Example 1-7. Specifying options when processing templates, ttperl3.pl

```
my $tt = Template->new({

    PRE_PROCESS  => 'header',

    POST_PROCESS => 'footer',

    INCLUDE_PATH => [

        '/home/dent/web/templates',     # src

        '/home/dent/web/lib',           # lib

    ],

});
```

Now when the process( ) method is invoked against the $tt object, the source template, *destruction.tt*, will be processed complete with the *header* and *footer* added before and after the main page content, respectively. For this example, we are assuming that the *destruction.tt* template is located in the */home/dent/web/templates* directory, and that *header* and *footer* can be found in the */home/dent/web/lib* directory.

The Template Toolkit provides numerous configuration options. These are described in detail in the Appendix. We describe the useful ones as we encounter them in later chapters.

1.5.4 Apache::Template Module

The Apache::Template module marries the Template Toolkit with the Apache web server. It is distributed separately from the rest of the Template Toolkit and can be downloaded at http://search.cpan.org/dist/Apache-Template/. It requires an Apache installation that includes Doug MacEachern's mod_perl extension module, details of which can be found at http://perl.apache.org/. For a full discussion of mod_perl, we recommend *Practical mod_perl*, by Stas Bekman and Eric

Cholet (O'Reilly), which contains an appendix dealing specifically with using the Template Toolkit under Apache and mod_perl.

Apache::Template can be configured via Apache's normal *httpd.conf* configuration file. Example 1-8 shows an extract of an *httpd.conf* file that sets the same options as Example 1-7.

Example 1-8. httpd.conf directives to set options with Apache::Template

```
PerlModule        Apache::Template

TT2IncludePath    /home/dent/web/templates
TT2IncludePath    /home/dent/web/lib
TT2PreProcess     header
TT2PostProcess    footer

TT2Params         uri env params cookies
TT2Headers        modified length

<Files *.tt2>
    SetHandler    perl-script
    PerlHandler   Apache::Template
</Files>
```

The first section loads the Apache::Template module:

```
PerlModule        Apache::Template
```

The next block sets some standard Template Toolkit options:

```
TT2IncludePath    /home/dent/web/templates
TT2IncludePath    /home/dent/web/lib
TT2PreProcess     header
TT2PostProcess    footer
```

Apache::Template adopts the Apache convention of using StudlyCaps for the names of configuration options and also adds a unique TT2 prefix. So the Apache::Template options TT2IncludePath and TT2PreProcess, for example, equate to the INCLUDE_PATH and PRE_PROCESS options for the Template module.

The two options that follow are specific to the Apache::Template handler:

```
TT2Params         uri env params cookies
TT2Headers        modified length
```

The first, TT2Params, provides a list of items that the handler should automatically extract from the Apache request and make available as template variables. Any template can use the uri, env, params, and cookies variables to access the request URI, environment variables, request parameters, and cookies, respectively. The second directive, TT2Headers, indicates that Last-Modified and Content-Length headers should be automatically added to the response sent to the client.

The final section uses the Apache Files directive to define the files that should be processed as templates:

```
<Files *.tt2>
    SetHandler    perl-script
    PerlHandler   Apache::Template
</Files>
```

The SetHandler and PerlHandler directives within the Files block are standard procedure in Apache for binding a mod_perl handler (Apache::Template in this case) to a set of files. With this configuration, the Apache server processes any files with a *.tt2* extension using the Apache::Template handler, but continues to deliver pages with any other extensions as static files, or using any other handlers defined for them.

This is a convenient way of mixing static HTML pages with dynamic page templates in any directory that is currently accessible by the Apache web server. If you want to create a static page, use a *.html* or other appropriate extension. If you want to create a dynamic page from a template, with the appropriate headers and footer added automatically, simply give it a *.tt2* extension and leave Apache::Template to take care of it.

If you would rather not open up your entire web server to the Apache::Template module, you can instead use the Location directive.

```
<Location /tt2/>
    SetHandler      perl-script
    PerlHandler     Apache::Template
</Location>
```

In this case, only those files located under the */tt2/* URI will be processed through the Apache::Template handler.

There are numerous other Apache configuration directives, all of which are described in the documentation provided with Apache. For a full discussion of the Apache::Template configuration, see the Appendix.

1.6 The Template Toolkit Language

The Template Toolkit language is a presentation language rather than a general-purpose programming language. It provides the kind of features that you would expect to see in a regular programming language, including loops, conditional tests, and the ability to manipulate variable values. However, in this case they serve a slightly different purpose. The Template Toolkit is designed for the task of generating content and presenting data, and it generally leaves more complex issues to a real programming language, namely, Perl.

We have already seen the basics of what a template looks like—a mixture of tags (known as directives) and other fixed text. The template processor interprets the directives and the remaining text is passed through unchanged.

By default, the start and end of a directive are marked by the sequences `[%` and `%]`, but the `TAGS` directive can be used to change them if you don't like these. The `TAGS` directive takes either one or two arguments. The single-argument version expects the name of a predefined tag set. For example, the `star` set replaces the tag delimiters with `[*` and `*]`:

```
[% TAGS star %]

People of [* planet *], your attention please.
```

If you give `TAGS` two arguments, they define the start and end tag markers that you want to use. For example, if you're processing plain text, you might find something like this more lightweight and easier to type:

```
[% TAGS { } %]

People of {planet}, your attention please.
```

Or if you are processing HTML and you prefer an HTML style, how about this:

```
[% TAGS <tt: > %]

<p>People of <tt:planet>, your attention please.
```

Changes to tags take effect immediately and affect only the current file.

You can also set these from the command line with *ttree* by using the `--start_tag`, `--end_tag`, and `--tag_style` options. From a Perl script, the corresponding configuration options for the `Template` module are `START_TAG`, `END_TAG`, and `TAG_STYLE`. For `Apache::Template`, the `TT2Tags` option can be used with one or two arguments, as per the `TAGS` directive.

In the rest of this book, we use the default tag style. We like it because it makes the directives stand out from the surrounding text, rather than making them blend in. We think it makes templates easier to read and write when you can more clearly distinguish one part from another.

1.7 Template Variables

The variables that we have used so far have been *scalar* variables. A scalar variable stores a single piece of information—either a string or a number.

The value of a scalar variable is inserted in a template by using the variable name inside a directive like this:

```
[% planet %]
```

A variable wouldn't be worthy of the name if you couldn't also set its value. We have seen examples of doing this using the --define option of the *tpage* command, but it is also possible to set a variable's value inside a template:

```
[% planet = 'Magrethea' %]

People of [% planet % ], your attention please.
```

1.7.1 Complex Variables

In addition to scalar variables, the Template Toolkit also supports two complex data types for storing multiple values: the *list* and *hash array* (also known as a *hash*). A list is an ordered array of other variables, indexed numerically and starting at element 0. A hash is an unordered collection of other variables, which are indexed and accessible by a unique name or *key*.

Perl programmers will already be familiar with these data structures. When you use the Template Toolkit from Perl you can easily define hash arrays and lists that are then passed as template variables to the process( ) method.

Example 1-9 shows a Perl program similar to Example 1-6, which defines a list of friends and a hash of terms as template variables.

Example 1-9. Perl program to process friends.tt

```
use Template;


my $tt    = Template->new( );

my $input = 'friends.tt';

my $vars  = {

    friends => [ 'Ford Prefect', 'Slartibartfast' ],

    terms  => {

        sass  => 'know, be aware of, meet, have sex with',

        hoopy => 'really together guy',

        frood => 'really, amazingly together guy',

    },

};


$tt->process($input, $vars)

    || die $tt->error( );
```

Example 1-10 is the *friends.tt* template that Example 1-9 processes.

Example 1-10. The friends.tt template

```
Your friends are:
[% FOREACH friend IN friends -%]
 * [% friend %]
[% END -%]

You know the following terms:
[% FOREACH term IN terms.keys.sort -%]
 [% term %]: [% terms.$term %]
[% END -%]
```

This is the output generated by Example 1-9:

```
Your friends are:
 * Ford Prefect
 * Slartibartfast

You know the following terms:
 frood: really, amazingly together guy
 hoopy: really together guy
 sass: know, be aware of, meet, have sex with
```

There will be times when you're using the Template Toolkit with *tpage* or *ttree* and don't want to have to write a Perl program, however simple, just to use some complex variables. The Template Toolkit allows you to define lists and hash data structures inside templates, using syntax similar (or identical if you prefer) to the Perl equivalents shown earlier.

The simple examples in the sections that follow should give you a flavor of how lists and hash data structures are defined and used in templates. Chapter 3 describes the Template Toolkit language in detail, showing the different variations in syntax that are permitted to satisfy both Perl programmers (who expect => to be used to separate a hash key from a value, for example) and HTML designers (who probably don't know any different and are just as happy using the simpler =).

1.7.2 Lists

A list variable is defined in a template using the [...] construct. Here's how we would create the equivalent of the friends list from Example 1-9:

```
[% friends = [ 'Ford Prefect', 'Slartibartfast' ] %]
```

List elements are accessed using the dot operator (.). Follow the list name with a dot and then the element number, starting at zero for the first element:

```
[% friends.0 %]        # Ford Prefect
[% friends.1 %]        # Slartibartfast
```

It is also possible to access elements from the list using a variable containing an index value. Simply prefix the variable with a $ character:

```
[% index = 1 %]
[% friends.$index %]    # Slartibartfast
```

1.7.3 Hashes

A hash is defined in a template using the {...} construct:

```
[%  terms = {
      sass  = 'know, be aware of, meet, have sex with'
      hoopy = 'really together guy'
      frood = 'really, amazingly together guy'
   }
%]
```

Each pair of items within the { and } is composed of the key, to the left of the = (or => if you prefer), and the value to the right. Separate pairs of items with commas, although it's not obligatory. Here is the same template written in a Perlish style:

```
[%  terms => {
      sass  => 'know, be aware of, meet, have sex with',
      hoopy => 'really together guy',
      frood => 'really, amazingly together guy',
   }
%]
```

Hash items are also accessed using the dot operator. In this case, the key for the required item is specified after the dot character:

```
[% terms.hoopy %]    # really together guy
```

You can also access hash items using a variable that contains a key. Again, the variable name should be prefixed with a $ character:

```
[% key = 'frood' %]
[% terms.$key %]     # really, amazingly together guy
```

1.7.3.1 Nesting list and hash definitions

Lists and hashes can be nested inside each other to create complex data structures:

```
[% arthur = {
    name    = 'Arthur Dent',
    planet  = 'Earth',
    friends = [
      { name = 'Ford Prefect'
        home = 'Betelgeuse'
        type = 'frood' }
      { name = 'Slartibartfast'
        home = 'Magrethea'
        type = 'hoopy' }
    ]
    terms = {
      sass  = 'know, be aware of, meet, have sex with'
      hoopy = 'really together guy'
      frood = 'really, amazingly together guy'
```

```
    }
  }
%]
```

You can access items buried deep within a nested data structure by chaining together a series of dot operations to create a *compound variable*:

```
[% arthur.friends.1.name %] # Slartibartfast
```

The Template Toolkit works out which dot operators are performing hash lookups (friends and name) and which are performing list lookups (1), and then automatically does the right thing to return the correct value. Comparing this to the equivalent Perl code, the Template Toolkit's uniform dot operator makes things much clearer:

```
# TT
arthur.friends.1.name


# Perl
$vars->{arthur}->{friends}->[1]->{name}
```

This illustrates one of the key benefits of using a presentation language like the Template Toolkit for generating content, rather than a programming language such as Perl.[1] When you write a program using a real programming language such as Perl, it's important to know which variables are scalars and which are lists, hashes, subroutines, objects, and so on. It's also critical that you use exactly the right kind of syntax relevant to each data type. Otherwise, your program might try to do something that it shouldn't, possibly corrupting the data, causing the program to exit with an error, or even failing to compile and run in the first place.

[1] Which of course, we still rely on a great deal, not only as the language in which the Template Toolkit is written, but also as the means by which you can extend it and add your own custom functionality to your templates, as we will see in the next section.

However, when you're writing templates to present your data as HTML pages, or in some other output format, these issues are of less concern. You're far more interested in how the data is going to be laid out, than in how it is stored or calculated on demand by the underlying Perl code (as we see in the next section). As long as the value for a user's name, for example, is inserted in the right place in the template when we ask for arthur.friends.1.name, we're happy. By the time the data is presented as output in a template, it is all text anyway.

You can also used dotted variables as hash keys to reference other variables. The following example shows how this is done using ${ ... } to explicitly scope the range of the second variable name:

```
[% arthur.terms.${arthur.friends.1.type} %]
```

The arthur.friends.1.type variable returns the value hoopy, resulting in a final expression equivalent to arthur.terms.hoopy. This ultimately provides us with the value really together guy.

You can use a temporary variable to break this down into smaller pieces. For example:

```
[% friend = arthur.friends.1 -%]
[% friend.name %] is a [% arthur.terms.${friend.type} %].
```

This generates the following output:

```
Slartibartfast is a really together guy.
```

1.7.4 Dynamic Variables

The examples that we've seen so far have used variables to store static values. When you set a variable to contain a scalar value or a reference to a list or hash array, it remains set to that value until the next time you explicitly modify it. Whenever the variable is used, the Template Toolkit simply looks up the current value for the variable and inserts it in the right place.

The Template Toolkit also allows subroutines and objects to be used to create dynamic variables. Each time such a variable is used, the Template Toolkit will call the subroutine or object method bound to it to return an appropriate value. Whereas static variables contain precomputed values, these dynamic variables return values that are recomputed each time they are used.

Example 1-11 shows a Perl program that defines two template variables, one bound to a subroutine, the other to an object.

Example 1-11. Dynamic data in template variables

```
use Acme::Planet;      # not a real module (yet)


my $vars = {

   help => sub {

      my $entry = shift;

      return "$entry: mostly harmless";

   },

   planet => Acme::Planet->new( name => 'Earth' ),

};
```

In this example, the help variable is a reference to a subroutine that expects a single argument, $entry. The planet variable references a hypothetical Acme::Planet object. This isn't a real module (at the time of this writing), but we're assuming that the new constructor method creates an Acme::Planet object against which we can invoke the name( ) method to return the value provided, Earth.

The following extract shows how these variables can be used in a template:

```
The guide has this to say about [% planet.name %].

  [% help(planet.name) %]
```

This would generate the following output:

```
The guide has this to say about Earth.

  Earth: mostly harmless
```

Notice that when we call the name method on planet we use the dot operator in exactly the same way as we would if planet were a hash with a key called name. The Template Toolkit doesn't care which of these we have, it just looks at the variable and works out what is the right thing to do. This illustrates how you are not tied down to any particular implementation for your underlying data structures, and can freely change from hashes to objects and back again without affecting the templates that use them.

Dynamic variables must be defined in Perl. There is no easy or clean way to define dynamic variables from within a template, other than by enabling the EVAL_PERL configuration option and using embedded Perl. The preferred solution is to write a simple Perl script that defines the relevant subroutines, objects, and other data items and then processes the appropriate template or templates. Another approach is to write a Template Toolkit plugin that encapsulates the Perl code and can be loaded into any template on demand. We look at plugins in detail in Chapter 6.

1.7.5 Virtual Methods

The Template Toolkit provides virtual methods for manipulating and accessing information about template variables. For example, the length virtual method can be applied to any scalar variable to return its string length in characters. The virtual method is applied using the dot operator:

```
[% name = 'Slartibartfast' %]

[% name %]'s name is [% name.length %] characters long.
```

This generates the output:

```
Slartibartfast's name is 14 characters long.
```

Virtual methods are provided for the three main variables types: scalars, lists, and hashes. The following example shows the join list virtual method being used to return the elements in a list joined into a single string. It adds a single space character between each item in the list by default, but you can provide a different delimiter by passing it as an argument in parentheses.

```
[% friends = [ 'Andy', 'Darren', 'Dave'] %]

Your friends are [% friends.join(', ') %].
```

This will display:

```
Your friends are Andy, Darren, Dave.
```

Some virtual methods alter the contents of the variable that they act on. For example, the pop method removes the last item from a list and returns it:

```
[% last = friends.pop %]

Your friends are [% friends.join(', ') %] and [% last %].
```

This will display:

```
Your friends are Andy, Darren and Dave.
```

We saw an example earlier of how virtual methods were combined in a dotted variable:

```
You know the following terms:

[% FOREACH term IN terms.keys.sort -%]

 [% term %]: [% terms.$term %]

[% END -%]
```

The part that we're particularly interested in is this:

```
terms.keys.sort
```

The terms variable contains a reference to a hash. The keys hash virtual method returns a reference to a list of the keys in the hash. The keys aren't returned in any particular order, but now that we have a list, we can go on to call the sort list virtual method to return a second list containing the items sorted in alphabetical order.

We can then go one step further and call the join virtual method on that list, to join the items into a single string:

```
[% terms.keys.sort.join(', ') %]
```

This generates the following output:

```
frood, hoopy, sass
```

Virtual methods are covered in detail in Chapter 3.

1.8 Template Directives

The examples we have looked at so far have concentrated on the use of variables. The Template Toolkit also provides more advanced language constructs called *directives*. These begin with an uppercase keyword such as PROCESS, IF, or FOREACH and tell the template processing engine to do something.

1.8.1 Variable Directives

Given that directives start with an uppercase keyword, you might be forgiven for thinking that the examples we have seen so far don't count as directives:

```
[% name = 'Arthur Dent' %]

[% planet = { name = 'Earth' } %]

Welcome [% name %] of [% planet.name %].
```

However, the syntax that we have been using until now to set and get variables is actually just a convenient shortcut for the full version, which uses the SET and GET keywords like so:

```
[% SET name = 'Arthur Dent' %]

[% SET planet = { name = 'Earth' } %]

Welcome [% GET name %] of [% GET planet.name %].
```

For obvious reasons, the shorter versions are used most of the time.

1.8.2 Template Processing Directives

Another use of template directives is for changing the way templates are processed. The PROCESS directive is one of the simplest. It loads another template file, processes the contents, and inserts the generated output in the calling template:

```
[% PROCESS header %]
```

The Template Toolkit provides the INCLUDE_PATH option, which allows you to specify one or more directories where your template files can be found. This allows you to specify your templates with simple names such *header*, rather than full file paths such as */home/dent/templates/lib/header*, for example.

The reason that it is called INCLUDE_PATH and not PROCESS_PATH becomes obvious when we mention that there is also an INCLUDE directive. The INCLUDE directive and related INCLUDE_PATH option have been part of the Template Toolkit, and the Text::Metatext module that preceded it, from the very beginning. The PROCESS directive, on the other hand, was added at a later date, and was able to reuse the INCLUDE_PATH option for the same purposes.

The difference between PROCESS and INCLUDE is revealed in Chapter 2. For now it suffices to know that INCLUDE is most often used when you want to pass variable values that should remain local to that one template:

```
[% INCLUDE header

   title = 'Vogon Poetry'

%]
```

The Template Toolkit is quite relaxed about how you lay out directives. You can add as little or as much whitespace as you like (including newlines) to help make your directive more readable. The only rule is that you must separate individual words and phrases in the directive (e.g., the INCLUDE keyword and the header template name that follows it) with at least one whitespace character. You don't need any spacing between the opening tag and the start of the directive, or between the end of the directive and the closing tag, but we recommend it to help make directives easier to read.

The following examples are all valid and equivalent ways of writing the same directive:

```
[%INCLUDE header title='Vogon Poetry'%]


[% INCLUDE header title='Vogon Poetry' %]
```

```
[% INCLUDE header
     title = 'Vogon Poetry'
%]
```

1.8.3 Loops

The FOREACH directive allows you to create loops, where a block of template content is processed, once for each item in a list. Here's the general form:

```
[% FOREACH item IN list %]
   block of template content...
   ...can contain directives...
   ...and reference the [% item %] variable...
[% END %]
```

We've already seen a real example of this in action:

```
You know the following terms:
[% FOREACH term IN terms.keys.sort -%]
 [% term %]: [% terms.$term %]
[% END -%]
```

We know from looking at virtual methods earlier that the terms.keys.sort variable returns a list of the items frood, hoopy, and sass. So our loop block will be repeated three times, with the term variable set to each of those values in turn. We print the term followed by its definition, fetched from the terms hash array using the value of term as the key. The term variable must be prefixed with $ to indicate that the value of the variable should be used rather than the literal string term:

```
 [% term %]: [% terms.$term %]
```

The output generated for the complete block is as follows:

```
You know the following terms:
 frood: really, amazingly together guy
 hoopy: really together guy
 sass: know, be aware of, meet, have sex with
```

1.8.4 Conditionals

Conditionals are another powerful language feature that allow your templates to make decisions about what to process and what not to process, based on the values of variables and more complex expressions.

We saw an example of the IF directive in Example 1-3, shown here in condensed form for brevity:

```
[% IF order.destruction %]
   As you will no doubt be aware...
[% ELSE %]
   Our representatives will be...
[% END %]
```

If the order.destruction variable is true, the first block, between the IF and ELSE directives, is processed. Otherwise, the block between the ELSE and END is used.

The notion of *truth* is, in this sense, the same as it is for Perl. If the variable is defined and contains any kind of value except an empty string or the number zero, both Perl and the Template Toolkit will consider it to be true. If the variable is undefined, or contains a zero-length string or the number zero, it is false. This applies to all Template Toolkit directives that perform operations based on evaluating a variable or more complex expressions for truth.

1.8.5 Filters, Plugins, and Macros

There's plenty more in the Template Toolkit that we introduce in the chapters that follow. The following examples give a taste of what is to come.

Filters allow you to postprocess the output of a block of template markup. The html filter, for example, will convert any HTML-sensitive characters, such as <, >, and &, into their equivalent HTML entities, <, >, and &.

```
[% FILTER html %]

  Home > Dent > Friends > Slartibartfast

[% END %]
```

This generates the following output, which, when displayed as HTML on a web browser, will show the original > characters as intended:

```
Home &gt; Dent &gt; Friends &gt; Slartibartfast
```

See Chapter 5 for further details.

Plugins allow you to load and use Perl modules in templates without having to write a Perl wrapper program to do it for you. The following examples show how the CGI plugin (which delegates to Lincoln Stein's CGI.pm module) can be used for CGI programming:

```
[% USE CGI %]

[% name   = CGI.param('name')   or 'Arthur Dent' %]

[% planet = CGI.param('planet') or 'Earth' %]

 Welcome [% name %] of planet [% planet %].
```

Plugins also have their own chapter, Chapter 6.

The final teaser that we're going to show you is the MACRO directive. This allows you to provide simple names for more complex commands, as the following example shows:

```
[% MACRO header(title, author)

    IF name = = 'Arthur Dent';

      INCLUDE arthur/header

        title = "Arthur Dent: $title";

    ELSE;

      INCLUDE guest/header

        title = "Guest User: $title";

    END;

%]
```

Don't worry if you can't make much sense of that now. The point that we're illustrating is that sometimes Template Toolkit code can get quite complex. However, the MACRO directive allows you to define the complicated part in one place so that you can use a much simpler call to the macro in the rest of your templates:

```
[% header('Arthur Dent', 'My Home Page') %]
```

1.9 Integrating and Extending the Template Toolkit

A particular strength of the Template Toolkit is that it doesn't try and do everything by itself. It concentrates on providing features that are generally applicable to template processing, leaving application-specific functionality to be added using Perl.

We've seen how you can define dynamic variables to allow your templates to access subroutines and objects written in Perl. The plugin mechanism allows you to bundle Perl code in self-contained modules that can be loaded straight into a template with a USE directive, eliminating the need to write a Perl wrapper program.

If that isn't enough, you can also define your own filters and virtual methods, and even change the language itself if you're feeling brave. This is covered in Chapter 8.

The fundamental concept that we're trying to get across is that the Template Toolkit is, as the name suggests, a *toolkit* for building things. It was designed to be easily extended and integrated with other components so that it can work within your requirements. It is not a complete web programming language or content management system that tries to do everything, and thus forces you into its way of thinking and working.

Sometimes that means you've got a little more thinking to do for yourself, rather than just blindly following the One True Way that we could have chosen for you. However, the benefit is that your solutions will be more flexible and adaptable, as well as better suited to addressing the problems at hand.

No two web sites (or document systems in general) are alike. Similarly, no two web developers agree on every issue that presents itself in the design and implementation of a web site. They each have their own ideas about the best way to tackle different problems, and prioritize different concerns according to the unique perspective that their past experience affords them. Perfect solutions don't exist (or if they do, we've never encountered them). With this in mind, strive to build a system that works today and tomorrow, even if it doesn't solve every problem overnight. Know when to compromise ideals for the sake of a pragmatic solution and when to stand firm on the issues that are important.

So the golden rule of web programming is that there is no golden rule. There are golden tools, and we like to consider the Template Toolkit among them, but a tool is only as good as the person who uses it. In the next chapter, we look at using the Template Toolkit to generate web content so that you can become familiar with its ways and start crafting your own web sites.

Chapter 2. Building a Complete Web Site Using the Template Toolkit

This chapter puts the Template Toolkit into context. We show several different ways of using the Template Toolkit to simplify the process of building and managing web site content. We start with some simple examples showing the use of template variables and template components that allow web content to be constructed in a modular fashion. As we progress further into the chapter, we look at more advanced techniques that address the issues of managing the site structure, generating menus and other navigation components, and defining and using complex data.

Although the focus of this chapter is on generating web content, it also serves as a general introduction to the Template Toolkit. It demonstrates techniques that can be adapted to different application areas. This chapter will quickly get you up to speed using the Template Toolkit, but without bogging you down in too much gory detail (we're saving that for the rest of the book). We come back to the Web to look at more advanced examples of static and dynamic web content in Chapter 11 and Chapter 12.

Although we may touch briefly on some more advanced issues, we try not to bore you with too much detail, except where it is absolutely necessary to illustrate a key point or explain an important concept. Chapter 3 discusses the syntax and structure of templates and the use of variables, while Chapter 4 covers the various template directives. More information relating to filters and plugins can be found in Chapter 5 and Chapter 6, respectively. More advanced topics concerning the use of the Template Toolkit for generating web content and interfacing to web applications can be found in Chapter 11 and Chapter 12.

We assume a Unix system in the examples in this chapter, but the principles apply equally well to other operating systems. On a Microsoft Windows machine, for example, the File Explorer can be used to create folders (directories) and shortcuts (symbolic links) using the familiar point-and-click interface. Another option we can highly recommend is to install *Cygwin*. *Cygwin* is freely available from http://www.cygwin.com and provides you with a Unix-like environment on Win32.

2.1 Getting Started

Every big web site is made up of individual pages. Let's start with a small and simple page, showing how to eliminate basic repetition using templates. In later sections, we can build on this to generate more pages and add more complex elements.

2.1.1 A Single Page

Example 2-1 shows the HTML markup of a page that displays the customary "Hello World" message, complete with a title, footer, and various other bits of HTML paraphernalia.

Example 2-1. hello.html

```
<html>
 <head>
  <title>Arthur Dent: Greet the Planet</title>
 </head>

 <body bgcolor="#FF6600">
  <h1>Greet the Planet</h1>

  <p>
   Hello World!
  </p>

  <hr />

  <div align="middle">
   &copy; Copyright 2003 Arthur Dent
  </div>
 </body>
</html>
```

HTML is relatively straightforward in terms of syntax and semantics. We'll assume that you've got at least a passing aquaintance with the basics of HTML. If you don't, *HTML & XML* by Chuck Musciano and Bill Kennedy (O'Reilly) provides a definitive guide to the subject.

Although HTML is simple, it does tend to be rather verbose. It's all too easy for the core content of the page to be obscured by the extra markup required around it. There's also some repetition that we would like to avoid. The page title and author's name both appear twice in the same page, for example. We can also assume that other pages in the site will be using similar pieces of data, repeated over and over again in numerous different places.

The author's name, background color, and copyright message are a few examples of items that we would really rather define in just one place in case we ever decide to change them. We don't want to have to edit every page in the site when we need to change the copyright message (at the start of a new year, for example), or decide that blue is the new orange and want to use it as the background color for every page.

2.1.2 A "Hello World" HTML Template

We can address these issues by applying the basic principles of template processing. Rather than creating the HTML page directly, we write a template for generating the HTML page. In this document, we use template variables to store these values instead of hardcoding them.

Example 2-2 shows a source template for the HTML page in Example 2-1. The author's name, page title, background color, and year have been replaced by the variables author, title, bgcol, and year, respectively.

Example 2-2. hello.tt

```
<html>
  <head>
    <title>[% author %]: [% title %]</title>
  </head>

  <body bgcolor="[% bgcol %]">
    <h1>[% title %]</h1>

    <p>
      Hello World!
    </p>

    <hr />

    <div align="middle">
      &copy; Copyright [% year %] [% author %]
    </div>
  </body>
</html>
```

2.1.3 Processing Templates with tpage

Of course, a template isn't something a browser can make sense of. We need to process the template to generate HTML to send to the browser. Let's use the *tpage* command we met in Chapter 1:

```
$ tpage --define author="Arthur Dent" \
>       --define title="Greet the Planet" \
>       --define bgcol="#FF6600" \
>       --define year=2003 \
>       hello.tt > hello.html
```

The *hello.html* now contains the same HTML that we saw in Example 2-1. This time, however, it has been generated from a template. The benefit of this approach is that we easily change any of these variable values and generate a new HTML page, simply by invoking *tpage* with a different set of parameters.

2.2 Template Components

Example 2-2 shows a template for generating a complete HTML page. We refer to this kind of template as a *page template* to distinguish it from the other kind of template that we're now going to introduce: the *template component*.

We use the term "template component" to help us identify those smaller templates that contain a reusable chunk of text, markup, or other content, but don't constitute complete pages in their own right. Template components are no different from page templates as far as the Template Toolkit is concerned—they're all just text files with embedded directives that need processing and get treated equally. Examples of typical template components include headers, footers, menus, and other user interface elements that you will typically want to use and reuse in different page templates across the site.

When we start using *ttree* a little later in this chapter, we will need to be more careful about storing our page templates separately from any template components. For now, however, we can keep them all in the same directory, simplifying matters for the purpose of our examples. As a general naming convention, we use a .tt or .html file extension for page templates (e.g., *hello.tt*), and no extension for component templates (e.g., *header*), but this is entirely arbitrary. If you want to give them an extension (e.g., *header.ttc*), that's fine.

2.2.1 Headers and Footers

Our first components can be created easily. Extract the header and footer blocks from Example 2-2 and save them in their own *header* and *footer* template files, as in Examples Example 2-3 and Example 2-4.

Example 2-3. header

```
<html>
  <head>
    <title>[% author %]: [% title %]</title>
  </head>

  <body bgcolor="[% bgcol %]">
    <h1>[% title %]</h1>
```

Example 2-4. footer

```
    <hr />

    <div align="middle">
      &copy; Copyright [% year %] [% author %]
    </div>
  </body>
</html>
```

2.2.1.1 The PROCESS directive

We can now load these template components into a page template using the PROCESS directive. Example 2-5 shows this in action.

Example 2-5. goodbye.tt

```
[% PROCESS header %]


  <p>
    Goodbye World.
  </p>


[% PROCESS footer %]
```

When the Template Toolkit encounters a PROCESS directive, it loads the template from the file named immediately after the PROCESS keyword (*header* and *footer* are the two templates in this example), processes it to resolve any embedded directives, and then inserts the generated output into the calling template in place of the original directive.

We can use *tpage* to process the *goodbye.tt* template and save the generated output to *goodbye.html*:

```
$ tpage --define author="Arthur Dent" \
>       --define title="We'll Meet Again" \
>       --define bgcol="#FF6600" \
>       --define year=2003 \
>       goodbye.tt > goodbye.html
```

The output generated, shown in Example 2-6, shows how the header and footer have been processed into place and the variable references within them correctly resolved.

Example 2-6. goodbye.html

```
<html>
  <head>
    <title>Arthur Dent: We'll Meet Again</title>
  </head>

  <body bgcolor="#FF6600">
    <h1>We'll Meet Again</h1>



  <p>
    Goodbye World.
  </p>


    <hr />


    <div align="middle">
      &copy; Copyright 2003 Arthur Dent
    </div>
  </body>
</html>
```

2.2.1.2 The INSERT directive

The Template Toolkit provides a number of different directives for loading external template components. The INSERT directive, for example, inserts the contents of a template, but *without* processing any directives that may be embedded in it:

```
[% INSERT footer %]
```

INSERT is faster than PROCESS because there's much less work involved in inserting a file than there is in processing it as a template. It's not going to work for us in our current example because of the year and author variables in the footer that need resolving. If we INSERT the footer as it is, we'll see the [% year %] and [% author %] directives passed through as literal text.

However, we can hardcode the variables in the footer to make it a fixed block of text that we can then load using INSERT. For example:

```
    <hr />


    <div align="middle">

      &copy; Copyright 2003 Arthur Dent

    </div>

  </body>

</html>
```

Although we've no longer got the benefit of using variables or other template directives, we are still defining the footer in one place where we can easily make changes, should we ever need to.

In most day-to-day applications, the difference in speed between INSERT and PROCESS isn't going to be noticeable unless you really go looking for it. You're generally better off using whatever is most convenient for you, the template author. Worry about performance only if and when it ever becomes an issue. With this in mind, we'll leave our variables in the footer and continue to use PROCESS.

The other directives for loading templates are INCLUDE and WRAPPER, which we'll be looking at shortly.

2.2.2 Benefits of Modularity

Separating commonly used blocks of markup into reusable template component files in this way allows you to take a modular approach to building your web content. This brings a number of important benefits.

The first is that the page templates become easier to write, edit, and maintain. You can quickly and easily add new pages by reusing existing template components to do the repetitive work, leaving the template author to concentrate on adding the core content. When it comes to updating the content, it becomes a lot easier to find what you're looking for because you don't have to pore through great chunks of HTML markup that define header, footers, menus, and other user interface elements.

In other words, we're achieving a *clear separation of concerns* between the core content of the pages and the parts that deal mainly with presentation. Content authors can concentrate on writing content without worrying about what kind of fancy user interface the web designers have dreamt up to fit around it

The second benefit is that the headers, footers, and other template components can easily be updated at any time, and need to be modified only in one place. Changing the copyright messages, the background color, or perhaps the layout of the footer, for *every* page on the site, becomes as easy as editing the one template component file and then processing the page templates to rebuild the site content.

So the clear separation of concerns also works the other way around. Web designers can concentrate on building a nice user interface for the entire site without having to worry too much about the content of individual pages.

Even if you're the all-in-one web designer, content author, and webmaster for your site, it is still useful to maintain a clear separation between these different aspects. You may have many hats to wear, but you'll be most comfortable wearing just one at a time.

2.3 Defining Variables

Our current use of *tpage* for processing templates is hardly streamlined. We're spending a lot of time typing variable values on the command line, something that can only get worse as we add more pages that require processing to the site.

It would be easy to mistype the value for a variable, for example, or perhaps supply the wrong value altogether. You wouldn't see any complaint from the Template Toolkit. It would just go right ahead and process the template with whatever values you supplied, possibly leading to an error on an HTML page that could go unnoticed.

2.3.1 Configuration Template

A better approach is to create a template component that defines any commonly used variables in one place. Example 2-7 shows our *config* template.

Example 2-7. config

```
[%  author = 'Arthur Dent'
    bgcol  = '#FF6600'    # orange
    year   = 2003
    copyr  = "Copyright $year $author"
-%]
```

You can define any number of variables in a single directive, as Example 2-7 illustrates. The Template Toolkit is very flexible in terms of the syntax it supports inside its tags, allowing you to spread your directives over several lines, adding as little or as much whitespace as you like for formatting purposes. You don't need to put each on a separate line as we have here—they can all go on the same line as long as some kind of whitespace is separating them. In the end, it's your choice. The Template Tooolkit isn't fussy about how you lay out your directives, as long as you follow the basic rules of syntax, which we'll be introducing throughout this chapter and describing in greater detail in Chapter 3.

2.3.1.1 Comments

You can add comments to annotate your code, as shown in the second line of Example 2-7: # orange. A comment starts with the # character and continues to the end of the current line. The comment is ignored by the Template Toolkit, and processing continues as normal on the next line.

If # is used as the first character immediately following the opening [% tag, the Template Toolkit ignores the entire directive up to the closing %]:

```
[%# this is a comment
    this line is also part of the comment
%]
```

2.3.1.2 Variable values

In Example 2-7, the four variables set are author, bgcol, year, and copyr. The first two are defined as the literal strings 'Arthur Dent' and '#FF6600'. The ' single quotation marks surrounding the values indicate that the contents should be used as provided. This makes it clear to the Template Toolkit that the # character in the definition for bgcol, for example, is part of the value and not the start of a comment. The third variable, year, is defined as the integer value 2003. Numbers such as these (and also floating-point numbers such as 2.718) don't need to be quoted, but can be if you prefer.

The last variable, copyr, shows an example of a double-quoted string, in which the value is enclosed by " characters. Here the Template Toolkit looks for any references to variables embedded in the string, denoted by the $ character, and replaces (*interpolates*) them for the corresponding values. In this example, the values for year and author will be interpolated into the string, resulting in the copyr variable being set to "Copyright 2003 Arthur Dent".

2.3.2 Loading the Configuration Template

The *config* template can now be loaded using the PROCESS directive to gain access to these variable definitions. This is shown in Example 2-8, which also defines the title variable specific to this page. This is really no different from the way you might define a constant or global variable at the start of a program in Perl or some other programming language. It's good practice to do this at the top of the file, where any future changes can easily be made.

Example 2-8. earth.tt

```
[% title = 'Earth' -%]
[% PROCESS config -%]
[% PROCESS header %]

  <p>
    Mostly Harmless.
  </p>

[% PROCESS footer %]
```

Notice the - character placed immediately before the closing %] tags at the end of the directives on the first two lines. This tells the Template Toolkit to remove, or *chomp*, the newline and any other whitespace following the directive. Some older web browsers don't like to see whitespace appearing before the opening <html> element, so this ensures that the *header* file is inserted right at the top of the output. In effect, it is as if we had written the template like so:

```
[% title = 'Earth' %][% PROCESS config %][% PROCESS header %]
...
```

Now the template can be processed using *tpage* without the need to provide variable values as command-line arguments:

```
$ tpage earth.tt > earth.html
```

2.3.2.1 Merging directives

The start of each page template can be simplified by defining the title variable and the PROCESS directives within a single directive tag. Each command is separated from the next by a ; (semicolon) character.

For example, we can write:

```
[% title = 'Earth';
   PROCESS config;
   PROCESS header
%]
```

instead of the more verbose:

```
[% title = 'Earth' -%]
[% PROCESS config -%]
[% PROCESS header %]
```

There's no need for a semicolon at the end of the last directive, but the Template Toolkit won't complain if it finds one there. As we saw earlier, semicolons aren't required between variable definitions that appear one after another. However, a semicolon is required if you switch from setting variables (which is technically the SET directive, although the explicit keyword is rarely used) to another kind of directive (e.g., PROCESS) in the same tag:

```
[% pi  = 3.142      # semicolon optional
   e   = 2.718      # "     "     "
   i   = 1.414;     # semicolon mandatory
   PROCESS config;  # "     "     "
   phi = 1.618      # semicolon optional
%]
```

The distinction becomes a little more obvious when we use the SET keyword explicitly and add some whitespace to format the directives more clearly:

```
[% SET pi  = 3.142
       e   = 2.718
       i   = 1.414;

   PROCESS config;

   SET phi = 1.618
%]
```

There's one final improvement we can make to the block at the start of our page templates. The two PROCESS directives can be merged into one, with the names of the templates separated by a + character:

```
[% title = 'Earth';

   PROCESS config
         + header
%]
```

The general rule of whitespace being insignificant inside directives applies equally well to the PROCESS directive, allowing us to list all the files on the same line, or across a number of lines, as we've done here. This flexibility allows us to lay out this header block in such a way that it's clear from a glance what's going on, and with the bare minimum of extra syntax cluttering up this high-level view.

Example 2-9 shows this in the context of a complete page template.

Example 2-9. magrethea.tt

```
[% title = 'Magrethea';

   PROCESS config
         + header
-%]

<p>
  Home of the custom-made
  luxury-planet building industry.
</p>

[% PROCESS footer %]
```

2.4 Generating Many Pages

The *tpage* program is fine for processing single templates, but isn't really designed to handle the many pages that comprise a typical web site. For this, *ttree* is much more appropriate. It works by drilling down through a source directory of your choosing, looking for templates to process. The output generated is saved in a corresponding file in a separate destination directory.

In addition to working well with a large number of template files, *ttree* also provides a much greater range of configuration options that allow you to modify the behavior of the Template Toolkit when processing templates. This allows you to further simplify the process of generating and maintaining web content in a number of interesting ways that we'll explore throughout this section.

Our templates will need to be organized a little more carefully when using *ttree*. In particular, we need to separate those page templates that represent complete HTML pages (*hello.tt*, *goodbye.tt*, *earth.tt*, and *magrethea.tt* in our previous examples) from those that are reusable template components (*config*, *header*, and *footer*).

2.4.1 Creating a Project Directory

We'll start by creating a directory for our web site, complete with subdirectories for the source templates for HTML pages (*src*), a library of reusable template components (*lib*), and the generated HTML pages (*html*). We'll also create a directory for miscellaneous files (*etc*), including a configuration file for *ttree*, and another (*bin*) for any scripts we accrue to assist in building the site and performing maintenance tasks.

```
$ cd /home/dent

$ mkdir web

$ cd web

$ mkdir src lib html etc bin
```

2.4.2 ttree Configuration File

Now we need to define a configuration file for *ttree*. Example 2-10 shows an example of a typical *etc/ttree.cfg* file.

Example 2-10. etc/ttree.cfg

```
# directories

src  = /home/dent/web/src

lib  = /home/dent/web/lib

dest = /home/dent/web/html


# copy images and other binary files

copy = \.(png|gif|jpg)$


# ignore CVS, RCS, and Emacs temporary files

ignore = \b(CVS|RCS)\b

ignore = ^#


# misc options

verbose

recurse
```

Options can appear in any order in the configuration file. In certain cases (such as lib, copy, and ignore), an option can be repeated any number of times.

The first section defines the three important template directories:

```
# directories

src  = /home/dent/web/src

lib  = /home/dent/web/lib

dest = /home/dent/web/html
```

The src option tells *ttree* where to look for HTML page templates. The lib option (of which there can be many) tells it where the library of additional template components can be found. Finally, the dest option specifies the destination directory for the generated HTML pages.

The next two sections provide regular expressions that *ttree* uses to identify files that should be copied rather than processed through the Template Toolkit (copy), and to identify files that should be ignored altogether (ignore):

```
# copy images and other binary files

copy = \.(png|gif|jpg)$



# ignore CVS, RCS, and Emacs temporary files

ignore = \b(CVS|RCS)\b

ignore = ^#
```

In this example, we're setting the options so that any images with png, gif, or jpg file extensions are copied, and any CVS or temporary files left lying around by our favorite text editor are ignored.

The next section sets two *ttree* flags:

```
# misc options

verbose

recurse
```

The verbose flag causes *ttree* to print additional information to STDERR about what it's doing, while it's doing it. The recurse flag tells it to recurse down into any sub-directories under the src directory.

2.4.3 Running ttree for the First Time

When you run *ttree* for the first time, it will display the following prompt, which asks if you'd like it to create a default *.ttreerc* file:

```
Do you want me to create a sample '.ttreerc' file for you?

(file: /home/dent/.ttreerc)   [y/n]:
```

Answer y to have it create the file in your home directory.

This file is used to provide a default configuration for *ttree*. If you've got only one web site to maintain, you can copy the contents of the *etc/ttree.cfg* file into it and run *ttree* without any command-line options:

```
$ ttree
```

If you've got more than one site to maintain, you'll probably want to keep separate configuration files for each. In that case, you can use the -f command-line option to provide the name of the configuration file when you invoke *ttree*:

```
$ ttree -f /home/dent/web/etc/ttree.cfg
```

2.4.4 Using a Build Script

Rather than providing a command-line configuration option for *ttree* each time you use it, you may prefer to write a simple build script that does it for you (as in Example 2-11).

Example 2-11. bin/build

```
ttree -f /home/dent/web/etc/ttree.cfg $@
```

The $@ at the end of the line passes any command-line arguments on to the *ttree* program, in addition to the -f option that is provided explicitly.

2.4.5 ttree Configuration Directory

Another alternative is to set the cfg option in the *.ttreerc* file to denote a default directory for *ttree* configuration files. You could set this to point to the project directory:

```
cfg = /home/dent/web/etc
```

and then invoke *ttree* with the short name of the configuration file:

```
$ tpage -f ttree.cfg
```

If you have many different web sites to maintain, another option is to create one general directory for *ttree* configuration files and use symbolic links from this directory to the project-specific files. The *.ttree* directory in your home directory is a common choice. In the *.ttreerc* file, we specify it like so:

```
cfg = /home/dent/.ttree
```

Then we prepare the directory, creating a symbolic link to our project-specific configuration file. We give it a memorable name (e.g., dentweb) to distinguish it from the various other *ttree.cfg* files that we may create links to from this directory:

```
$ cd /home/dent

$ mkdir .ttree

$ cd .ttree

$ ln -s /home/dent/web/etc/ttree.cfg dentweb
```

With these changes in place, *ttree* can then be invoked using the -f option to specify the *dentweb* configuration file:

```
$ tpage -f dentweb
```

The settings in the *.ttreerc* file and the magic of symbolic links result in *ttree* ending up with the right configuration file without us having to specify the full path to it every time. The other benefit of this approach is that *ttree* can be invoked from any directory and the correct configuration file will still be located.

2.4.6 Calling ttree Through the Build Script

From now on we'll assume that the *bin/build* script invokes *ttree* with the appropriate option to locate the configuration file. For the sake of clarity, we'll use it in the examples that follow whenever we want to build the site content, rather than calling *ttree* directly. Any other commands that you want performed when the site is built (e.g., copying files, restarting the web server or database) can also be added here.

As we saw in Example 2-11, any command-line options that we provide to the script are forwarded to *ttree*. One particularly useful option is -h, which provides a helpful summary of all the different *ttree* options:

```
$ bin/build -h

ttree 2.63 (Template Toolkit version 2.10)



usage: ttree [options] [files]



Options:

   -a     (--all)          Process all files, regardless of modification

   -r     (--recurse)      Recurse into sub-directories

   -p     (--preserve)     Preserve file ownership and permission

   -n     (--nothing)      Do nothing, just print summary (enables -v)

   -v     (--verbose)      Verbose mode
```

```
  -h      (--help)         This help
  -dbg    (--debug)        Debug mode
  -s DIR  (--src=DIR)      Source directory
  -d DIR  (--dest=DIR)     Destination directory
  -c DIR  (--cfg=DIR)      Location of configuration files
  -l DIR  (--lib=DIR)      Library directory (INCLUDE_PATH)  (multiple)
  -f FILE (--file=FILE)    Read named configuration file     (multiple)

File search specifications (all may appear multiple times):
  --ignore=REGEX           Ignore files matching REGEX
  --copy=REGEX             Copy files matching REGEX
  --accept=REGEX           Process only files matching REGEX

Additional options to set Template Toolkit configuration items:
  --define var=value       Define template variable
  --interpolate            Interpolate '$var' references in text
  --anycase                Accept directive keywords in any case.
  --pre_chomp              Chomp leading whitespace
  --post_chomp             Chomp trailing whitespace
  --trim                   Trim blank lines around template blocks
  --eval_perl              Evaluate [% PERL %] ... [% END %] code blocks
  --load_perl              Load regular Perl modules via USE directive
  --pre_process=TEMPLATE   Process TEMPLATE before each main template
  --post_process=TEMPLATE  Process TEMPLATE after each main template
  --process=TEMPLATE       Process TEMPLATE instead of main template
  --wrapper=TEMPLATE       Process TEMPLATE wrapper around main template
  --default=TEMPLATE       Use TEMPLATE as default
  --error=TEMPLATE         Use TEMPLATE to handle errors
  --start_tag=STRING       STRING defines start of directive tag
  --end_tag=STRING         STRING defined end of directive tag
  --tag_style=STYLE        Use pre-defined tag STYLE
  --plugin_base=PACKAGE    Base PACKAGE for plugins
  --compile_ext=STRING     File extension for compiled template files
  --compile_dir=DIR        Directory for compiled template files
  --perl5lib=DIR           Specify additional Perl library directories
```

2.4.7 A Place for Everything, and Everything in Its Place

Before we can run the build script to generate the site content, we will need to move our page and library template files into place.

The source templates for the HTML pages should now be moved into the *src* directory where *ttree* can find them. The HTML files that *ttree* generates in the *html* output directory will be given the same filename as the *src* template from which they are generated. For this reason, we'll be using a *.html* file extension on our page templates from now on.

Also, move the template components *config*, *header*, and *footer* into the *lib* directory. These are (for now) also identical to those shown in the earlier examples.

2.4.8 Running the Build Script

Now we can run the *bin/build* script to invoke *ttree* to build the site content:

```
$ bin/build
ttree 2.63 (Template Toolkit version 2.10)

      Source: /home/dent/web/src
 Destination: /home/dent/web/html
Include Path: [ /home/dent/web/lib ]
      Ignore: [ \b(CVS|RCS)\b, ^# ]
        Copy: [ \.(png|gif|jpg)$ ]
      Accept: [ * ]

  + earth.html
  + magrethea.html
```

The sample output from *ttree* shown here indicates that two page templates, *earth.html* and *magrethea.html*, were found in the *src* directory. The + character to the left of the filenames indicates that the templates were processed successfully. Corresponding *earth.html* and *magrethea.html* files will have been created in the *html* directory containing the output generated by processing the templates.

Now that we've set up *ttree* and told it where our page templates are located, we can add new pages to the site by simply adding them to the *src* directory. When you next run the build script, *ttree* will locate the new page templates, even if they're located deep in a subdirectory (thanks to the recurse option), and process them into the corresponding place in the *html* directory.

You can now build all the static web pages in your site using a single, simple command.

2.4.9 Skipping Unmodified Templates

When *ttree* is run it tries to be smart in working out which templates need to be processed and which don't. It does this by comparing the file modification time of the page template with the corresponding output file (if any) that it previously generated.

Run the *bin/build* script again, and the + characters to the left of the filename change to the - character:

```
$ bin/build
ttree 2.63 (Template Toolkit version 2.10)

      Source: /home/dent/web/src
 Destination: /home/dent/web/html
Include Path: [ /home/dent/web/lib ]
      Ignore: [ \b(CVS|RCS)\b, ^# ]
        Copy: [ \.(png|gif|jpg)$ ]
```

```
Accept: [ * ]

  - earth.html                    (not modified)
  - magrethea.html                (not modified)
```

This indicates that the templates weren't processed the second time around, with the message to the right of the filenames explaining why. In this case, *ttree* has recognized that the source templates, *src/earth.html* and *src/magrethea.html*, haven't been modified since the corresponding output files, *html/earth.html* and *html/magrethea.html*, were created. Given that nothing has changed, there's no need to reprocess the templates.

There may be times when you want to force *ttree* to build a particular page or even all the pages on the site, regardless of any file modification times. You can process one or more pages by naming them explicitly on the command line:

```
$ bin/build earth.html magrethea.html
```

One time that you might want to force all pages to be rebuilt is when you modify a header, footer, or some other template component that is used by all the pages. Unfortunately, *ttree* isn't smart enough to figure out which library templates are used by which page templates.[1] The -a option tells *ttree* to ignore file modification times and process all page templates, regardless:

[1] This occurs not because *ttree* is being lazy. It's actually very difficult, if not impossible, to do it accurately without processing the templates in their entirety. By this time, the Template Toolkit has already done the hard work, so there's nothing to be gained by discovering that the template didn't need processing after all.

```
$ bin/build -a
```

2.5 Adding Headers and Footers Automatically

In addition to the fact that *ttree* works well with large collections of page templates, it also has the benefit of providing a large number of configuration options that allow you to change the way it works and how it uses the underlying Template Toolkit processor. Two of the most convenient and frequently used options are pre_process and post_process. These allow you to specify one or more templates that should be *automatically* added to the top or bottom of each page template, respectively. This can be used to add standard headers and footers to a generated page, but pre- and postprocessed templates may not generate any visible output at all. For example, we can use a preprocessed template to configure some variables that we might want defined for use in the page template or other template components.

The following can be added to the bottom of the *etc/ttree.cfg* file to have the *config* and *header* templates preprocessed (in that order so that we can use variables defined in *config* in the *header*) and the *footer* template postprocessed:

```
pre_process  = config
pre_process  = header
post_process = footer
```

Now the page templates can be made even simpler, as Example 2-12 shows.

Example 2-12. src/magrethea.html

```
[% title = 'Magrethea' -%]


<p>
  Home of the custom-made
  luxury-planet building industry.
</p>
```

Remember that you'll need to use the -a option to force *ttree* to rebuild all pages in the site to have the changes take effect:

```
$ bin/build -a
```

2.5.1 Defining META Tags

There is one problem with this approach. The *header* template is processed in its entirety before the main page template gets a look in. This means that the title variable isn't set to any value when the *header* is processed. It doesn't get set until the page template is processed, by which time it's too late for the *header* to use it.

The Template Toolkit won't complain if it encounters a variable for which it doesn't have a value defined. Instead, it will quietly use an empty string (i.e., nothing at all) for the value of the variable and continue to process the remainder of the template. The DEBUG option (described in the Appendix) can be set to have it raise an error in these cases, and can be useful to help track down mistyped variable names and those that have somehow eluded definition.

We can use the META directive to solve our immediate problem. It works by allowing us to define values within the page template that *are* accessible for use in the *header* and any other preprocessed templates, *before* the main page template is itself processed.

Example 2-13 shows how this is done. Instead of defining the title in a SET directive (which technically we were, even if we had omitted the SET keyword for convenience), we use the META directive, but otherwise leave the definition of the variable unmodified.

Example 2-13. src/milliways.html

```
[% META title = 'Milliways' %]


<p>

  The Restaurant at the

  End of the Universe.

</p>
```

Variables defined like this are made available as soon as the template is loaded. This happens *before* any of the preprocessed templates are processed so that these META variables are defined and ready for use.

There are some subtle differences between META variables and normal SET variables. The first is that you can't use double-quoted strings to interpolate other variables into the values for META variables. You *can* use double-quoted strings, but you can't embed variables in them and expect them to get resolved. The simple reason for this is that META variables are defined before the template is processed with any live data. At this time, there aren't any variables defined, so there's no point trying to use them.

The second difference is that the variables must be accessed using the template. prefix:

```
[% template.title %] not [% title %]
```

The template variable is a special variable provided by the Template Toolkit containing information about the current page template being processed. It defines a number of items, including the name of the template file (template.name) and the modification time (template.modtime), as well as any META variables defined in the template (template.title).

The dot operator, ., is the Template Toolkit's standard notation for accessing a variable such as title that is one small part of a larger, more complex data structure such as template. It doesn't matter for now (or generally at all) how this is implemented behind the scenes because the dot operator hides or *abstracts* that detail from you so that you don't need to worry about it.

We'll be coming back to the dot operator later on in this chapter when we look at defining and using complex data structures. For now, it is sufficient to know that template.title is how we access the title META variable defined in the main page template.

We can easily modify our *header* template to accommodate these requirements and restore the page title to the generated header (see Example 2-14).

Example 2-14. lib/header

```
<html>

  <head>

    <title>[% author %]: [% template.title %]</title>

  </head>


  <body bgcolor="[% bgcol %]">

    <h1>[% template.title %]</h1>
```

2.6 More Template Components

You can create any number of different reusable template components to help you generate the content for your web site. Whenever you find yourself repeating the same, or a similar, block of markup in more than one place, you might want to consider moving it into a separate template file that you can then use and reuse whenever you need it. This not only saves you a lot of typing, but also ensures that the HTML generated in each place you use it is identical, or as near to identical as you would like it to be, accounting for any variables that might change from one use to the next.

Example 2-15 shows a template component for displaying an entry from Arthur's favorite reference book.

Example 2-15. lib/entry

```
<p>
  The Hitch Hiker's Guide to the Galaxy
  has this to say on the subject of
  "[% title %]".
</p>

<table border="0">
  <tr valign="top">
    <td>
      <b>[% title %]:</b>
    </td>
    <td>
      [% content %]
    </td>
  </tr>
</table>
```

The template uses two variables, title and content. The value for title can in this case be copied from template.title, thereby providing the title set in the META directive for the page. A value for content will be set explicitly for the sake of simplicity. These variables can be set either before the PROCESS directive:

```
[% title   = template.title
   content = 'Mostly harmless'
%]

[% PROCESS entry %]
```

or as part of the PROCESS directive, following the template name as additional arguments:

```
[% PROCESS entry
     title   = template.title
     content = 'Mostly harmless'
%]
```

The end result is the same. The Template Toolkit treats all variables as global by default so that you can define a variable in one template and use it later in another without having to explicitly pass it as an argument every time. In both of the preceding examples, the title and content variables are defined globally and can subsequently be used in both the called template (*entry*) and the calling template (*earth.tt*) after the point of definition.

In the following fragment, for example, the reference to the content variable at the end of the template will generate the value "Mostly harmless" as set in the earlier PROCESS directive:

```
[% PROCESS entry
     title   = template.title
     content = 'Mostly harmless'
%]

[% content %]   # Mostly harmless
```

2.6.1 The INCLUDE Directive

There may be times when you would rather keep the definition of certain variables local to a particular template. The INCLUDE directive provides a way of doing this. In terms of syntax, it is used in exactly the same way as the PROCESS directive in all except the keyword.

The key difference between INCLUDE and PROCESS is that INCLUDE *localizes* any variables that are passed to the template as arguments in the directive. The variables passed have local values for the template component being processed by INCLUDE, but then revert to their previous values or undefined states.

In the following fragment, we define two variables at the start of the template whose values we would like to preserve to be used in the sentence at the end:

```
[% name  = 'Zaphod Beeblebrox'
   title = 'President of the Galaxy'
%]

[% INCLUDE entry
     title   = 'Earth'
     content = 'Mostly harmless'
%]

Hi!  I'm [% name %], [% title %].
```

The INCLUDE directive provides local definitions for the title and content variables for the *entry* template to display. However, the original value for the title variable will be left untouched, and there will be no trace of the content variable outside of the *entry* template.

The final line of the template generates the output that we're expecting:

```
Hi!  I'm Zaphod Beeblebrox, President of the Galaxy.
```

Had we used PROCESS instead of INCLUDE, the value for title would have been overwritten and the output generated by the final line would incorrectly read:

```
Hi!  I'm Zaphod Beeblebrox, Earth.
```

There is one important caveat to be aware of. The INCLUDE directive only localizes simple variables. Any complex variables containing dot operators are effectively global regardless of whether you use INCLUDE, PROCESS, or any other directive.

Dotted variables are a little like Perl's package variables. In Perl, you can refer to a variable as, for example, $My::Dog::Spot. This tells Perl the precise location for the variable $Spot in the My::Dog package. In the Template Toolkit, the equivalent variable would be something like my.dog.spot.

On the other hand, a Perl variable written as just $Spot could be either a "global" (for these purposes) variable defined in the current package, or a lexically scoped variable in the current subroutine, for example. Similarly, in the Template Toolkit, the equivalent variable spot could also be a global variable or a local copy created by invoking a template using INCLUDE.

The explanation isn't important as long as you remember the simple rule: the INCLUDE localizes only simple variables that don't contain any "." dots.

2.6.2 Setting Default Values

When you define a reusable template component, you may want to provide default values for any variables used in the template. For example, the following template component might want to ensure that sensible values are provided for the <title> element and bgcolor attribute in the <body>, even if the respective title and bgcol variables aren't set:

```
<html>
  <head>
    <title>[% title %]</title>
  </head>
  <body bgcolor="[% bgcol %]">
    ...
```

2.6.2.1 The DEFAULT directive

One way to achieve this is by using the DEFAULT directive. The syntax is the same as SET in everything but the keyword, allowing you to provide default values for one or more variables:

```
[% DEFAULT
     title = "Arthur Dent's Web Site"
     bgcol = '#FF6600'
-%]
<html>
  <head>
    <title>[% title %]</title>
  </head>
  <body bgcolor="[% bgcol %]">
    ...
```

The key difference between DEFAULT and SET is that DEFAULT will set the variable to the value prescribed only if it is currently undefined, if it is set to an empty string, or if it contains the number zero. (Perl programmers will recognize the similarity with Perl's idea about what is *true* and *false* when it comes to the value of a variable.) The component will use any existing values for title and bgcol, either defined globally or passed as explicit arguments when the template is used. Otherwise, it will use the values provided in the DEFAULT directive.

2.6.2.2 Expressions

Another approach is to use Template Toolkit *expressions* instead of just variables. Expressions allow you to make logical statements including the and and or operators, both of which can be written in either upper- or lowercase. For example, we can write:

```
[% bgcol or '#FF6600' %]
```

instead of just:

```
[% bgcol %]
```

The tertiary ?: operator is another option. It provides the equivalent of an IF...THEN...ELSE construct, in which the expression to the left of the ? is evaluated to determine whether it is true or false. If true, whatever comes after the ? and before the : is used. Otherwise, it returns whatever follows the :.

Here's an example showing how the ?: operator can be used to generate an appropriate title for the page:

```
[% title ? "Arthur Dent: $title"
         : "Arthur Dent's Web Site"
%]
```

If the title variable is set, the string "Arthur Dent: $title" is used. This uses variable interpolation to insert the current value for the title variable into the string, following Arthur's name. If title isn't set to anything that the Template Toolkit considers meaningfully true, the string "Arthur Dent's Web Site" is instead used. The expression doesn't need to be split across two lines as we've shown here, but in this case it helps to make the code clearer and easier to read.

So if title is set to Earth, the directive will generate the following output:

```
Arthur Dent: Earth
```

If the title isn't set, it will instead generate this output:

```
Arthur Dent's Web Site
```

Expressions can also contain comparison operators, as shown in the following example. These are discussed in detail in Chapter 3.

```
[% age > 18 ? 'Welcome to my site...'

          : "Sorry, but you're not old enough..."

%]
```

2.6.2.2.1 = versus = =

One important distinction worth mentioning now is the difference between = and = =. The first performs an assignment, setting the variable named on the left to the value (or expression) on the right:

```
[% foo = bar %]
```

The second is the equality comparison operator, which tests to see whether the string values of the items on either side are identical:

```
[% foo =  = bar ? 'equal' : 'not equal' %]
```

2.6.2.2.2 Setting variables using expressions

Expressions can also be used to set the value of a variable. For example, the pagetitle variable can be set to either of the values previously shown, depending on the setting of title, using the following code:

```
[% pagetitle = title ? "Arthur Dent: $title"

                  : "Arthur Dent's Web Site"

%]
```

It's perfectly valid to use a variable in an expression to update the same variable. Everything to the right of the = is evaluated first, and the resulting value is then used to set the variable specified to the left of the =:

```
[% title = title ? "Arthur Dent: $title"

              : "Arthur Dent's Web Site"

%]
```

2.6.2.2.3 Setting variables using directives

You can also assign the output of a directive to a variable. In the following example, the *header* template is processed using the PROCESS directive and the generated output is stored in the headtext variable:

```
[% headtext = PROCESS header %]
```

2.6.3 The IF Directive

The IF directive can be used to encode more complex conditional logic in templates. It evaluates the expression following the IF keyword, which in these examples will be a simple variable. If the expression is true, the following block, up to the matching END directive, is processed. Otherwise, it is ignored.

Here's a simple example:

```
<body

[%- IF bgcol -%]

 bgcolor="[% bgcol %]"

[%- END -%]

>
```

This example uses an IF block to add the bgcolor attribute to the HTML <body> element, but only if the bgcol variable is defined and contains a true value. By careful placement of - characters at the start and end of the IF and END directives, we're enabling the Template Toolkit's prechomping and postchomping facility. This removes the newline characters before the [% tags and after the %] tags so that the output lines up in the correct place in the <body> element.

So, for a bgcol value of #FF6600, the following output would be generated:

```
<body bgcolor="#FF6600">
```

For an undefined bgcol, we would instead see the following:

```
<body>
```

Like many of the Template Toolkit directives that expect a block to follow, the IF directive can be used in *side-effect* notation.

For example, you can write:

```
[% INCLUDE header IF title %]
```

instead of the more laborious:

```
[% IF title; INCLUDE header; END %]
```

This works only when you've got a single directive or variable as the content for the block—in this example, it's the INCLUDE header directive. Our earlier example, which constructed the <body> tag, included both text and a reference to the bgcol variable in the block. However, we can write this using a double-quoted string to interpolate the value for bgcol:

```
<body [%- " bgcolor=\"$bgcol\"" IF bgcol %]>
```

Matters are complicated a little by the need to escape the double quotes inside the double quotes. The \ character tells the Template Toolkit that the following " is part of the string, and not the quote that terminates it. Overall it's an improvement over the more explicit IF...END form and illustrates a useful principle.

You can add an ELSE block after the IF block, which will be processed if the variable (or more generally, the expression) is false. For example:

```
[% IF bgcol -%]

<body bgcolor="[% bgcol %]">

[%- ELSE -%]

<body>

[%- END -%]
```

There is also the ELSIF directive, which allows you to define different blocks for different conditions:

```
[% IF name =  = 'Arthur Dent'

   OR name =  = 'Ford Prefect' %]

Hello [% name %]!

[% ELSIF name.match('(?i:vogon)') %]

I'm sorry, but there's no one at home.

Please don't bother calling again.

[% ELSE %]

Hello World!

[% END %]
```

In this example, the ELSIF expression uses the match virtual method to test whether the name contains anything looking remotely Vogon. The argument passed to the match method is a Perl regular expression, allowing us to use the (?i:...)

grouping to construct a case-insensitive match. An ELSE block is also provided in case neither the IF nor ELSIF conditions match.

The SWITCH directive, described in detail in Chapter 4, provides an alternative for more complicated multiway matching.

2.7 Wrapper and Layout Templates

Now it's time to bring out some of the bigger guns of the Template Toolkit. The WRAPPER directive and layout templates let you define a common look for web pages in a single file, rather than scattering the components over *header* and *footer* files.

2.7.1 The WRAPPER Directive

The *entry* template from Example 2-15 works well when the content to be displayed is relatively simple. However, it quickly becomes cumbersome for longer entries such as the one shown here:

```
[% INCLUDE entry
   title   = 'Vogon Poetry'
   content = 'Vogon poetry is of course the
         third worst in the Universe.
         The second worst is that of...

         ...etc...

         ...in the destruction of the
         planet Earth'
%]
```

Special care must be taken when quoting content that contains quote characters. Consider the following extract that illustrates this problem:

```
Grunthos is reported to have been "disappointed"
by the poem's reception.
```

If this is enclosed in single-quote characters, the apostrophe in "poem's" must be escaped by preceding it with a backslash \ character (the apostrophe and single-quote characters are one and the same for these purposes):

```
[% INCLUDE entry
   title   = 'Grunthos the Flatulent'
   content = 'Grunthos is reported to have
         been "disappointed" by the
         poem\'s reception.'
%]
```

Another alternative is to use double quotes to define the variable, allowing single quotes to remain as they are. But in this case, any occurrences of double quotes will then need to be escaped:

```
[% INCLUDE entry
   title   = 'Grunthos the Flatulent'
   content = "Grunthos is reported to have
         been \"disappointed\" by the
         poem's reception."
%]
```

A better solution is to use the WRAPPER directive. It works in a similar way to INCLUDE, but uses an additional END directive to enclose a block of template content. The WRAPPER directive uses this block as the value for the content variable:

```
[% WRAPPER entry
     title = 'Grunthos the Flatulent'
%]
   Grunthos is reported to have
   been "disappointed" by the
   poem's reception.
[% END %]
```

The immediate benefit in this example is that the extract is now a block of plain text rather than a quoted string. There is no longer any need to escape the quote characters within it.

The WRAPPER block can contain any combination of text and template directives, even including other nested WRAPPER blocks. The following fragment shows a simple example in which the reaction variable is used to report Grunthos' reaction:

```
[% reaction = 'disappointed' %]


[% WRAPPER entry
     title = 'Grunthos the Flatulent'
%]
   Grunthos is reported to have
   been "[% reaction %]" by the
   poem's reception.
[% END %]
```

The WRAPPER block is processed first to resolve any directives within it. Then the complete block, including any output generated dynamically by embedded directives, is passed to the *entry* template as the value for the content variable.

It's no coincidence that we chose content as a variable name in the *entry* template in Example 2-15, knowing full well that we would later use it in this example for WRAPPER. The WRAPPER directive always assigns the block content to the content variable, and in that sense it's one of the Template Toolkit's "special" variables, like the template variable that we used earlier. However, there's nothing to stop you from using it as a regular variable, and indeed it makes a good choice in any template for a variable that you might one day want to define as a block in a WRAPPER directive.

The end result is that the *entry* template works as expected, whether we call it using INCLUDE and pass the content explicitly as a variable, or call it using WRAPPER and define the content implicitly in the enclosed block.

2.7.2 Using an Automatic Wrapper Template

In Examples Example 2-4 and Example 2-14, we created separate *header* and *footer* files to add to the start and end of each HTML page generated. One problem with this approach is that neither file contains valid HTML markup. The *header* provides the opening tag of the html element, for example, but the corresponding closing tag is located at the end of the *footer* file.

Having HTML elements split across separate files makes them harder to maintain, and increases the likelihood of them being accidentally mismatched or incorrectly nested. It is also likely to confuse or infuriate any HTML-aware text editors or validation tools that you may be using.

A better approach is to use a *wrapper* template to combine the *header* and *footer* into one template. The content variable is used to denote the position for the page content. This is shown in Example 2-16.

Example 2-16. lib/wrapper

```
<html>
  <head>
    <title>[% author %]: [% template.title %]</title>
```

```
  </head>

  <body bgcolor="[% bgcol %]">
    <h1>[% template.title %]</h1>

    [% content %]

    <hr />

    <div align="middle">
      &copy; [% copyr %]
    </div>
  </body>
</html>
```

We need to modify the *etc/ttree.cfg* file to specify the new *wrapper* template using the wrapper option. The fact that our *wrapper* template happens to be called *wrapper* is entirely coincidental (but intentional). We could have named the file *tom*, *dick*, *larry*, or something else if we wanted to, but it wouldn't be as succinct or descriptive as *wrapper*.

We're still using the pre_process option to load the *config* template, but we can now remove the references to the *header* and *footer* (or comment them out as shown here), replacing them with a single wrapper option:

```
pre_process  = config
wrapper      = wrapper
# pre_process  = header
# post_process = footer
```

With the wrapper option in place, the Template Toolkit processes the main page template (after preprocessing the *config* template) and then calls the *wrapper* template, passing the generated page content as the content variable. It has the same effect as if there were an explicit WRAPPER directive around the entire page content:

```
[% WRAPPER wrapper %]
   The entire page content goes here...
[% END %]
```

Of course, the benefit of having the Template Toolkit apply a *wrapper* automatically is that you don't need to edit any of your page templates to add it explicitly. You can switch from using pre_process and post_process to wrapper, or you can change the name of any of the *header*, *footer*, or *wrapper* templates, without having to make any changes to your core content.

To put the change into effect, run the *bin/build* script with the -a option to have it rebuild all pages in the site:

```
$ bin/build -a
```

2.7.3 Using Layout Templates

Most real web sites will require far more complex layout templates than the simple *wrapper* we saw in Example 2-16. A common practice is to use HTML tables to place different elements such as headers, footers, and menus in a consistent position and formatting style. These elements may themselves be built using tables and other HTML elements, perhaps nested several times over. This can quickly lead to confusing markup that is hard to read and even harder to update.

Consider the following example, which illustrates how difficult nested tables can be to write and maintain:

```
<table border="0" cellpadding="0" cellspacing="0">
  <tr valign="top">
    <td>
      <table border="0">
        <tr>
          <td>
            Oh Dear!
          </td>
          <td>
            This is not a good example
            of a layout template...
          </td>
          <td>
            <table>
              ...etc...
            </table>
        </tr>
      </table>
    </td>
    <td>
      <table>
         ...etc...
      </table>
    </td>
    .
    .
    .
```

The sensible formatting helps to make the structure clearer through use of indenting. However, it is still difficult to match rows and cells with their corresponding tables, and there is little indication of what the different tables contribute to the overall layout.

A better approach is to build the layout using several different templates. For example, we can simplify the preceding template by moving the inner tables to separate templates:

```
<table border="0" cellpadding="0" cellspacing="0">
  <tr valign="top">
    <td>
      [% PROCESS sidebar %]
    </td>
    <td>
      [% PROCESS topmenu %]
    </td>
```

.

.

.

Now we can easily see the high-level structure without getting bogged down in the detail of the nested tables. Furthermore, by giving our templates names that reflect their purpose (e.g., sidebar and topmenu), we effectively have a self-documenting template that shows at a glance what it does. Another benefit is that the individual elements, the sidebar and topmenu in this example, will themselves be much easier to write and maintain in isolation. They also become reusable, allowing you to incorporate them into another part of the site (or perhaps another site) with a PROCESS or similar directive.

2.7.4 Layout Example

Let's work through a complete example now, applying this principle to the presentation framework for our web site. Example 2-17 shows an alternate version of the *wrapper* template that delegates the task to two further templates, *html* and *layout*.

Example 2-17. lib/wrapper2

```
[% WRAPPER html + layout;
     content;
   END
-%]
```

The two wrapper templates, *html* and *layout*, are both specified in the one WRAPPER directive, separated using the + character in the same way that we used it with the PROCESS directive in Example 2-9. In this case, the page content will be processed first, then the *layout* template, and finally the *html* template. Remember that the WRAPPER directive works "inside out" by processing the wrapped content first, and then the wrapping templates.

If we unwrap the preceding directive into two separate WRAPPER calls, it should become more obvious why the WRAPPER directive processes the templates in the *reverse* order to how they're specified:

```
[% WRAPPER html;
     WRAPPER layout;
       content;
     END;
   END
%]
```

The end result is that it does what you would expect, regardless of the slightly counterintuitive order in which it does it. The *html* template ends up wrapping the *layout* template, which in turn wraps the value of the content variable, which in this case is the output from processing the main page template.

2.7.4.1 Side-effect wrappers

The WRAPPER directive can also be used in side-effect notation. Consider the following fragment:

```
[% WRAPPER layout;
     content;
   END
%]
```

You can simplify this by writing it as follows:

```
[% content WRAPPER layout %]
```

The *wrapper* template shown in Example 2-17 can be rewritten in the same way, as shown in Example 2-18.

Example 2-18. lib/wrapper3

```
[% content WRAPPER html + layout -%]
```

2.7.4.2 Separating layout concerns

Using two separate layout templates, *html* and *layout*, allows us to make a clear separation between the different kinds of markup that we're adding to each page. The *html* template adds the <head> and <body> elements required to make each page valid HTML. The *layout* template deals with the overall presentation of the visible page content, adding a header, footer, menu, and other user interface components.

Example 2-19 shows the *html* template.

Example 2-19. lib/html

```
<html>
  <head>
    <title>[% author %]: [% template.title %]</title>
  </head>

  <body bgcolor="[% bgcol %]">
    [% content %]
  </body>
</html>
```

Example 2-20 shows the *layout* template.

Example 2-20. lib/layout

```
<table border="0" width="100%">
  <tr>
    <td colspan="2">
      [% PROCESS pagehead %]
    </td>
  </tr>
  <tr valign="top">
    <td width="150">
      [% PROCESS menu %]
    </td>
    <td>
      [% content %]
    </td>
  </tr>
  <tr>
    <td colspan="2" align="center">
```

```
      [% PROCESS pageinfo %]
    </td>
  </tr>
</table>
```

We've created a new header template, *pagehead*, shown in Example 2-21, which generates a headline for the page. It's simple for now, but we can easily change it to something more complicated at a later date.

Example 2-21. lib/pagehead

```
<h1>[% template.title %]</h1>
```

We're also using another template, *menu*, to handle the generation of a menu for the site. We'll be looking at this shortly.

Example 2-22 shows the final template used in the layout, *pageinfo*. This incorporates the copyright message and some information about the page template being processed.

Example 2-22. lib/pageinfo

```
[% USE Date %]

&copy; [% copyr %]

<br />

[% template.name -%]
last modified
[%- Date.format(template.modtime) %]
```

Notice how we're using the template.name and template.modtime variables to access the filename and modification time of the current page template. The template.modtime value is returned as a large number that means something to computers[2] but not a great deal to humans. To turn this into something more meaningful, we're using the Date plugin to format the number as a human-readable string.

[2] It's the number of seconds that have elapsed since January 1, 1970, known as the the *Unix epoch*.

2.7.4.3 Plugins and the USE directive

Plugins are a powerful feature of the Template Toolkit that allow you to load and use complex functionality in your templates, but without having to worry about any of the underlying implementation detail. Plugins are covered in detail in Chapter 6, but there's not much you need to know to start using them.

In Example 2-22, we first load the Date plugin with the USE directive:

```
[% USE Date %]
```

This creates a Date template variable that contains a reference to a plugin object (of the Template::Plugin::Date class, but you don't need to know that). We can then call the format method against the Date object using the dot operator, passing the value for template.modtime as an argument:

```
[%- Date.format(template.modtime) %]
```

The output generated would look something like this:

```
17:43:35 14-Jul-2003
```

That's all we need to do to load and use the Date plugin. Dozens of plugins are available for doing all kinds of different tasks, described in detail in Chapter 6.

2.8 Menu Components

In the *layout* template in Example 2-20, we delegate the task of generating a menu for the web site to the *menu* template. Before we look at how the template does this, let's see an example of the kind of HTML that we would like it to generate.

```
<table border="0">
  <tr>
    <td>
      <img src="/images/icon.png" width="4" height="4" />
    </td>
    <td>
      <a href="earth.html">Earth</a>
    </td>
  </tr>
  <tr>
    <td>
      <img src="/images/icon.png" width="4" height="4" />
    </td>
    <td>
      <a href="magrethea.html">Magrethea</a>
    </td>
  </tr>
</table>
```

The entire menu is defined as a <table> element, containing one <tr> row for each item, each of which holds two <td> cells, one to display an icon, the other a link to a particular page. Only two items are in this simple example, but already we can see how it gets repetitive very quickly. This suggests that we can modularize the markup into separate template components.

2.8.1 Simple Menu Template

Example 2-23 shows a *menu* template that defines the outer <table> elements and uses a second template, *menuitem*, to generate each item.

Example 2-23. lib/menu

```
<table border="0">
[%
 PROCESS menuitem
  text = 'Earth'
  link = 'earth.html';
```

```
 PROCESS menuitem
  text = 'Magrethea'
  link = 'magrethea.html';
%]
</table>

[% BLOCK menuitem %]
<tr>
  <td>
    <img src="/images/icon.png" width="4" height="4" />
  </td>
  <td>
    <a href="[% link %]">[% text %]</a>
  </td>
</tr>
[% END %]
```

2.8.1.1 The BLOCK directive

We could easily define the *menuitem* template in a separate file as we have with other components, but it would require us to split the HTML `<table>` markup into different files. This would make it harder to maintain and possibly lead to tag mismatch or other formatting errors.

Instead, we define the *menuitem* template inside the *menu* template using the `BLOCK` directive. The argument following the `BLOCK` keyword is a name for the template component, which can then be used in any `PROCESS`, `INCLUDE`, or `WRAPPER` directives. The content of the component follows, and can contain any kind of Template Toolkit directives up to the corresponding `END` directive.

```
[% BLOCK menuitem %]
   <tr>
     <td>
       <img src="/images/icon.png" width="4" height="4" />
     </td>
     <td>
       <a href="[% link %]">[% text %]</a>
     </td>
   </tr>
[% END %]
```

The *menuitem* template block is defined at the bottom of the *menu* template, but that doesn't stop us from using it earlier in the same template, before it is defined.

The *menuitem* block will remain defined while the *menu* template is being processed. Any other templates that are called from within the *menu* template (e.g., by a `PROCESS` or `INCLUDE` directive) will also be able to use the *menuitem* block.

2.8.2 Component Libraries

When a template is loaded using the PROCESS directive, any BLOCK definitions within it will be imported and available for use in the calling template. Templates loaded using the INCLUDE directive keep to themselves and don't export their BLOCK definitions (or any of their local variables, as described in the earlier discussion of the INCLUDE directive).

This feature allows you to create single template files that contain libraries of smaller template components, defined using the BLOCK directive. This is illustrated in Example 2-24.

Example 2-24. lib/mylib

```
[% BLOCK image -%]
  <img src="[% src %]" alt="[% alt %]"
  width="[% width %]" height="[% height %]" />
[%- END %]

[% BLOCK link -%]
  <a href="[% link %]">[% text %]</a>
[%- END %]

[% BLOCK icon;
   INCLUDE image
     src    = '/images/icon.png'
     alt    = 'dot icon'
     width  = 4
     height = 4 ;
END
-%]
```

Notice how the *icon* BLOCK definition is defined within a single directive, and consists of nothing more than a call to the *image* template component, defined earlier in the same file. This illustrates how easy it is to reuse existing components to quickly adapt them for more specific, or alternate purposes.

The BLOCK definitions can be loaded from the *mylib* template with a PROCESS directive. Then they can be used just like any other template component. Example 2-25 shows a variation of the *menu* template from Example 2-23 in which the *icon* and *link* components are used to generate the menu items.

Example 2-25. lib/menu2

```
[% PROCESS mylib %]

<table border="0">
[%
  PROCESS menuitem
    text = 'Earth'
    link = 'earth.html';

  PROCESS menuitem
```

```
    text = 'Magrethea'
    link = 'magrethea.html';
%]
</table>

[% BLOCK menuitem %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link %]
  </td>
</tr>
[% END %]
```

2.8.2.1 The EXPOSE_BLOCKS option

You can also set an option that allows you to use BLOCK directives without having to first PROCESS the template in which they're defined. The expose_blocks option for *ttree* and the corresponding EXPOSE_BLOCKS option for the Template module can be set to make this possible.

For example, by adding the following to the *etc/ttree.cfg* file:

```
expose_blocks
```

we can then access a BLOCK in the *mylib* template like so:

```
[% PROCESS mylib/icon %]
```

The template name, *mylib*, is followed by the BLOCK name, *icon*, separated by a / (slash) character. The notation is intentionally identical to how you would specify the *icon* file in the *mylib* directory. This is another example of how the Template Toolkit abstracts certain underlying implementation details so that you don't tie yourself down to one particular way of doing something.

At a later date, for example, you might decide to split the *mylib* template into separate files, stored in the *mylib* directory. The same directive will continue to work because the syntax is exactly the same for blocks in files as it is for files in directories:

```
[% PROCESS mylib/icon %]
```

This gives you more flexibility in allowing you to change the way you organize your template components, without having to worry about how that might affect the templates that use them.

2.8.3 The FOREACH Directive

The menu component from Example 2-25 can be simplified further by first defining a list of menu items and then iterating over them using the FOREACH directive. Example 2-26 demonstrates this.

Example 2-26. lib/menu3

```
[% PROCESS mylib %]

[% menu = [
    { text = 'Earth'
```

```
      link = 'earth.html' }
    { text = 'Magrethea'
      link = 'magrethea.html' }
   ]
%]

<table border="0">
[% FOREACH item IN menu %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link
         text = item.text
         link = item.link
    %]
  </td>
</tr>
[% END %]
</table>
```

The menu variable is defined as a list of hash arrays, each containing a text and link item:

```
[% menu = [
     { text = 'Earth'
       link = 'earth.html' }
     { text = 'Magrethea'
       link = 'magrethea.html' }
   ]
%]
```

The main body of the template defines an HTML <table> element. Within the table, the FOREACH directive iterates through the menu list, setting the item variable to each element in turn.

```
<table border="0">
[% FOREACH item IN menu %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link
```

```
        text = item.text
        link = item.link
     %]
   </td>
 </tr>
[% END %]
</table>
```

The block following the FOREACH directive, up to the corresponding END, can contain text and other directives, even including nested FOREACH blocks. To make the code easier to read, we might prefer to define the *menuitem* BLOCK, as shown in Example 2-25. This allows us to simplify the FOREACH directive, merging it into a single tag.

```
<table border="0">
[% FOREACH item IN menu;
     PROCESS menuitem
       text = item.text
       link = item.link;
   END
%]
</table>
```

The FOREACH block now contains just one directive to PROCESS the *menuitem* component. The text and link variables are set to the item.text and item.link values, respectively.

When the items in a FOREACH list are hash arrays, as they are in Example 2-26, you can omit the name of the item variable:

```
<table border="0">
[% FOREACH menu;
     PROCESS menuitem;
   END
%]
</table>
```

In this case, the values in each hash array will be made available as local variables inside the FOREACH block. So item.text becomes the text variable, and item.link becomes link, but only within the scope of the FOREACH block. This conveniently allows us to process the *menuitem* template without needing to explicitly dereference the item variables.

There's one more improvement we can make by taking advantage of the Template Toolkit's side-effect notation. Instead of writing the PROCESS menuitem directive in the FOREACH block all by itself, we can put it *before* the FOREACH and do away with the semicolons and END keyword:

```
<table border="0">
[% PROCESS menuitem FOREACH menu %]
</table>
```

All these enhancements to the *menu* template are shown in Example 2-27.

Example 2-27. lib/menu4

```
[% PROCESS mylib %]

[% menu = [
    { text = 'Earth'
      link = 'earth.html' }
    { text = 'Magrethea'
      link = 'magrethea.html' }
   ]
%]

<table border="0">
[% PROCESS menuitem FOREACH menu %]
</table>

[% BLOCK menuitem %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link %]
  </td>
</tr>
[% END %]
```

2.9 Defining and Using Complex Data

The variables that we have used so far have mostly been simple *scalar* variables that contain just one value. The few exceptions include the tantalizing glimpses of the template variable, and the Date plugin in Example 2-22. As we saw in Chapter 1, the Template Toolkit also supports lists and hash arrays for complex data, and allows you to access Perl subroutines and objects.

In this section, we will look more closely at defining and using complex data structures, and describe the different Template Toolkit directives for inspecting, presenting, and manipulating them.

2.9.1 Structured Configuration Templates

Larger sites will typically use dozens of different global site variables to represent colors, titles, URLs, copyright messages, and various other parameters. The Template Toolkit places no restriction on the number of different variables you use, but you and your template authors may soon lose track of them if you have too many.

Another problem with having lots of global variables lying around is that you might accidentally overwrite one of them. We saw in Example 2-7 how the author variable was used to store the name of the site author, Arthur Dent, for use in the *header* and *footer* templates. At some later date, we might decide to add a *quote* template component that also uses the author variable. This is shown in Example 2-28.

Example 2-28. lib/quote

```
<blockquote>
  [% quote %]
</blockquote>


  -- [% author %]
```

There's no problem if we use INCLUDE to load the template, providing a local variable value for author:

```
[% INCLUDE quote
     author = 'Douglas Adams'
     quote  = 'I love deadlines. I like the
               whooshing sound they make as
               they fly by.'
%]
```

The value for author supplied as a parameter to the INCLUDE directive (Douglas Adams) remains set as a local variable within the *quote* template. It doesn't affect the global author variable that is defined in the config (Arthur Dent).

However, it is all too easy to forget that the author variable is "reserved"—especially if it's just one of a large number of such variables—and to use PROCESS instead of INCLUDE:

```
[% PROCESS quote
     author = 'Douglas Adams'
     quote  = 'I love deadlines. I like the
               whooshing sound they make as
               they fly by.'
%]
```

The PROCESS directive doesn't localize any variables. As a result, our global author variable now is incorrectly set to Douglas Adams instead of Arthur Dent. One solution is to religiously use INCLUDE instead of PROCESS at every opportunity. However, that's just working around the problem rather than addressing the real issue. Furthermore, the INCLUDE directive is quite a bit slower than PROCESS, and if performance is a concern for you, you should be looking to use PROCESS wherever possible.

Variables are localized for the INCLUDE directive in a part of the Template Toolkit called the *Stash*. It saves a copy of all the current variables in use before the template is processed, and then restores them to these original values when processing is complete. Understandably, this process takes a certain amount of time (not much in human terms, but still a finite amount), and the more variables you have, the longer it takes.

It is worth stressing that for most users of the Template Toolkit, these performance issues will be of no concern whatsoever. If you're using the Template Toolkit to generate static web content offline, it makes little difference if a template takes a few hundredths or thousandths of a second longer to process. Even for generating dynamic content online, performance issues such as these probably aren't going to concern you unless you have particularly complicated templates or your site is heavily loaded and continually generating lots of dynamic content.

The more important issue is one of human efficiency. We would like to make it easier for template authors to keep track of the variables in use, make it harder for them to accidentally trample on them in a template component, and ideally, allow them to use PROCESS or INCLUDE, whichever is most appropriate to the task at hand.

The answer is to use a nested data structure to define all the sitewide variables under one global variable. Example 2-29 shows how numerous configuration variables can be defined as part of the site data structure, in this case implemented using a hash array.

Example 2-29. lib/site

```
[%  site = {
    author = 'Arthur Dent'
    bgcol  = '#FF6600'    # orange
    year   = 2003
  }

  site.copyr = "Copyright $site.year $site.author"
%]
```

To interpolate the values for the year and author to generate the copyright string, we must now give them their full names, site.year and site.author. We need to set the site.copyr variable *after* the initial site data structure is defined so that we can use these variables. In effect, the site variable doesn't exist until the closing brace, so any references to it before that point will return empty values (unless the site has previously been set to contain these items at some earlier point).

```
[% site = {
     author = 'Arthur Dent'
     bgcol  = '#FF6600'    # orange
     year   = 2003

     # this doesn't work because site.year
     # and site.author are undefined at
     # this point
     copyr  = "Copyright $site.year $site.author"
   }
%]
```

Sitewide values can now be accessed through the site hash in all templates, leaving author, bgcol, year, and all the other variables (except site, of course) free to be used, modified, and updated as "temporary" variables by page templates and template components. Now there's just one variable to keep track of, so there's much less chance of accidentally overwriting an important piece of data because you forgot it was there. It also means that the INCLUDE directive works faster because it has only one variable to localize instead of many. The Stash copies only the top-level variables in the process of localizing them and doesn't drill down through any of the nested data structures it finds.

2.9.2 Layered Configuration Templates

As your site data structure becomes more complicated, you might find it easier to build it in layers using several templates. Example 2-30 shows a preprocessed configuration template that loads the *site*, *col*, and *url* templates using PROCESS.

Example 2-30. lib/configs

```
[% PROCESS site
     + col
     + url
-%]
```

We have already seen the *site* template in Example 2-29. Example 2-31 shows the *col* and *url* configuration templates.

Example 2-31. lib/col

```
[%  site.rgb = {
      white  = '#FFFFFF'
      black  = '#000000'
      orange = '#FF6600'
    }

    site.col = {
      back = site.rgb.orange
      text = site.rgb.white
    }
-%]
```

Example 2-31 shows the definition of a site.rgb hash and then another, site.col, which references values in the first. Template authors can use explicit colors, by referencing site.rgb.orange, for example, to fetch the correct RGB value, #FF6600. Or they can code their templates to use colors defined in the site.col structure—for example, referencing site.col.back in the *html* template to set the bgcolor attribute of the HTML <body> element. Either way, the colors are defined in one place, and the symbolic names allow us to see at a glance that the background color for the pages in the site is currently orange.

The *url* template is a little simpler, but also illustrates how variables can be built in stages (see Example 2-32).

Example 2-32. lib/url

```
[%  url = 'http://tt2.org/ttbook'

    site.url = {
      root   = url
      home   = "$url/index.html"
      help   = "$url/help.html"
      images = "$url/images"
    }

-%]
```

The benefits of this approach are twofold. The first is that you can save yourself a great deal of typing by replacing a long-winded URL with a shorter variable name. The second benefit is that you can easily change all the URL values in a single stroke by changing the root url from which they are constructed.

One advantage of building a complex data structure from several templates is that you can easily replace one of the templates without affecting the others. For example, you might want to use a different set of URL values at some point. Rather than edit the *url* template, you can copy the contents to a new file (e.g., *url2*), make the changes there, and then update the *configs* template accordingly:

```
[% PROCESS site
         + col
         + url2
-%]
```

If you must revert to the old URLs at a later date, you need to change only the *configs* template to load *url* instead of *url2*. You can also use this approach to load different configuration templates based on a conditional expression. For example:

```
[% PROCESS site
         + col;

   IF developing;
     PROCESS url2;
   ELSE;
     PROCESS url;
   END
-%]
```

2.9.3 Choosing Global Variables Wisely

Fewer global variables are better, but don't try to cram everything into the one site variable if more would do the job better. Try and separate your variables into structures according to their general purpose and relevance to different aspects of the site. For example, you can define one structure containing everything related to the site as a whole (e.g., site), and another related to the individual page being processed (e.g., page):

```
[% site = {
     title  = "Arthur Dent's Web Site"
     author = 'Arthur Dent'
     # ...etc...
   }

   page = {
     title  = template.title
     author = template.author or site.author
   }
%]
```

You may also want to define others to represent a user, server, application, or request depending on how you're using the Template Toolkit and what you're using it for.

The Template Toolkit allows you to use upper- or lowercase, or some combination of the two, to specify variable names. It's not recommended that you use all uppercase variable names, as they might clash with current (or future) Template Toolkit directives. However, you might like to capitalize your global variables to help you remember that they're special in some way (e.g., Site versus site):

```
[% Site = {
     # ...etc...
   }
   Page = {
     # ...etc...
   }
   User = {
     # ...etc...
   }
%]
```

2.9.4 Passing Around Data Structures

You can pass a complex data structure around the Template Toolkit as easily as you would a scalar variable. Example 2-33 shows a configuration template that defines the site.menu data structure to contain the menu items that we used earlier in Example 2-26.

Example 2-33. lib/menudef

```
[% site.menu = [
    { text = 'Earth'
      link = 'earth.html' }
    { text = 'Magrethea'
      link = 'magrethea.html' }
   ]
%]
```

We've moved the definition of the sitewide menu into a central configuration file and will need to add it to the list of templates loaded by the PROCESS directive in the pre-processed *configs* template shown in Example 2-30:

```
[% PROCESS site
         + col
         + url
         + menudef
-%]
```

Now we can remove the definition of the menu structure from the component (or components) that generate the menu in a particular style, as shown in Example 2-34.

Example 2-34. lib/menu5

```
[% PROCESS mylib %]


<table border="0">
[%- FOREACH item IN menu;
      PROCESS menuitem
        text = item.text
```

```
    link = item.link;
END
-%]
</table>

[% BLOCK menuitem %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link %]
  </td>
</tr>
[% END %]
```

The value for menu (site.menu in this case) is passed to the *menu5* template as an argument in an INCLUDE directive:

```
[% INCLUDE menu5
     menu = site.menu
%]
```

The benefit of this approach is that the component that generates the menu is now generic, and will work with any menu data you care to define. Wherever you need a menu in the same style, simply call the component and pass in a different definition of menu data:

```
[% INCLUDE menu5
     menu = [
       { text = 'Milliways'
         link = 'milliways.html' }
       { text = 'Hotblack Desiato'
         link = 'desiato.html' }
     ]
%]
```

Separating the definition of a menu from its presentation also makes it easier to change the menu style at a later date. There's only one generic menu component to update or replace, regardless of how many times it is used in various places around the site. If you want two or more different menu styles, simply create additional menu components with different names or in different locations. For example, you may have *site_menu* and *page_menu*, or *site/menu* and *page/menu*, or perhaps something such as *slick/graphical/menu* and *plain/text/menu*.

2.10 Assessment

This brings us nicely back to where we started, looking at the basic principle of template processing: separating your *data* from the way it is *presented*. It's not always clear where your data belongs: in a configuration template; defined in a Perl script; or perhaps stored in a SQL database or XML file. Sometimes you'll want to begin by defining some simple variables in a configuration template so that you can start designing the layout and look and feel of the site. Later on, you might choose to define that data somewhere else, passing it in from a Perl script or making it available through a plugin.

The beauty of the Template Toolkit is that it really doesn't matter. It abstracts the details of the underlying implementation behind the uniform dotted notation for accessing data so that your templates keep working when your storage requirements change, as they inevitably will for many web sites.

It also makes it easy to include things such as loops, conditional statements, and other templates as easy as possible so that you can concentrate on presentation, rather than getting bogged down in the more precise details of full-blown programming language syntax. This is what we mean when we describe the Template Toolkit as a *presentation language* rather than a *programming language*.

It is an example of a *domain-specific language* that in many ways is similar to SQL, which is a domain-specific language for formulating database queries. As such, it should generally be used for what it is good at, rather than being contorted into doing something that might be a lot easier in another language. That doesn't mean that you can't use the Template Toolkit to do CGI programming, embed Perl, or even write Vogon poetry, if that's your thing, but that's not necessarily where its particular strengths lie.[3]

[3] Although the jury is still grooping hooptiously at the implorations of generating Vogon Poetry using the Template Toolkit.

And that's where Perl comes in. The Template Toolkit is designed to integrate with Perl code as cleanly and as easily as possible. When you want to do something more than the Template Toolkit provides, it is easy to append your own additions using a real programming language such as Perl. The plugin mechanism makes it easy to load external Perl code into templates so that you're not always writing Perl wrapper scripts just to add something of your own.

However, this total separation is not something that the Template Toolkit enforces, although the default settings for various configuration options such as EVAL_PERL do tend to encourage it. Sometimes you just want to define a simple Perl subroutine in a template, for example, and don't want to bother with a separate Perl script or plugin module. The Template Toolkit gives you the freedom to do things such as this when you really want to.

For example, by enabling the EVAL_PERL option (see Chapter 4 and the Appendix for details), we can quickly define a Perl subroutine and bind it to a template variable, using a PERL block such as the following:

```
[% PERL %]
$stash->set( help => sub {
   my $entry = shift;
   return "$entry: mostly harmless";
} );
[% END %]
```

The $stash->set(var => $value) code, shown here binding the help variable to the Perl subroutine, is the Perl equivalent of writing [% var = value %] in a template—except, of course, that you can't usually define a subroutine directly in a template, only by using Perl code with EVAL_PERL set (which we think is a sensible restriction). This block can easily be defined in a preprocessed configuration template to keep it out of harm's way, leaving the template authors to use the simple variable:

```
[% help('Earth') %]
```

The important thing is to achieve an *appropriate* separation of concerns, rather than a *total* separation of concerns. Sometimes it's easier to define everything in one template or Perl program and to use a clear layout to separate the different parts. Splitting a small and self-contained document into several different pieces, each comprising just one part of the jigsaw puzzle, can make it hard to see the big picture. On the other hand, a more complex web site may have bigger pieces that absolutely need to be maintained in isolation from the other parts. Remember, there is no golden rule, so the Template Toolkit doesn't try and enforce one on you.

The techniques that we've taught you in this chapter will allow you to address most, if not all, of the simple but common problems that you'll typically face when building and maintaining a web site. We'll be coming back to the Web in Chapter 11 to look at some further ways in which the Template Toolkit can be used to enhance your site and make your life easier. In Chapter 12, we'll be showing how it can be used to handle the presentation layer to simplify the process of building and customizing web applications.

Chapter 3. The Template Language

While a programming language is designed to manipulate data, a presentation language is used to turn the data into plain text, HTML, or some other format.[1] As long as the data is made available to us in a textual representation when we ask for it, we really don't need to worry too much about how it is stored or computed behind the scenes.

[1] We'll assume for now that the presentation formats are all different kinds of text, although you can also use the Template Toolkit to generate binary files such as images.

That's not to say that you can't create and manipulate variables in templates. However, their most common use is for dealing only with presentation aspects, by using variables to define colors or other layout parameters, displaying the first *N* search results, or sorting a list of names into alphabetical order, for example. It is unusual (but not unheard of) to use the Template Toolkit to modify data that has any lasting effect. In general, data is passed to a template and then thrown away, so it doesn't matter if it's changed in any way.

In this chapter, we take a closer look at the details of the Template Toolkit presentation language. The general syntax of templates comes under scrutiny first, and we give examples of how the default style can be customized using configuration options and template directives. The rest of the chapter is then dedicated to an in-depth study of variables. We describe the various data types, showing how they are defined and used in both Perl and template markup.

We concentrate on the general characteristics of the language without looking too closely at any of the specific directives that the Template Toolkit provides (PROCESS, WRAPPER, USE, and so on). These are described in detail in Chapter 4. A full discussion of filters and plugins is left for Chapter 5 and Chapter 6, respectively.

While you can write templates that have a lasting effect on data—say, by updating a database directly—that's not really how the Template language was intended to be used. We return to this subject in Chapter 11 and Chapter 12, when we look more closely at separating the functional parts of an application from those that deal only with presentation.

3.1 Template Syntax

The Template Toolkit has many configuration options to change the appearance and meaning of the directives in a template. This section looks at the different types of directives, shows how to change the directive tags, and describes the various ways you can control the processing of whitespace around directives.

3.1.1 Text and Directives

A template contains a mixture of fixed text and directive tags, denoted by the [% and %] markers. Everything coming after the [% and before the following %] is part of the directive tag. Everything else in the document is fixed text that is passed through intact.

Well, that's the default behavior, anyway. There are certain occasions when the text surrounding directives *will* be modified. For example, the whitespace *chomping* options (PRE_CHOMP and POST_CHOMP) and related flags (which we'll be looking at shortly) tell the Template Toolkit to remove any extraneous whitespace in the text on either side of (i.e., before or after) a directive. The INTERPOLATE option is another example that, when set (which it isn't by default), causes the text part of the template to be passed through a second scanning process to look for any embedded variables, denoted by a $ prefix—e.g., Hello $planet. More on that later.

You can also change the characters used to denote tags with the TAG_STYLE, START_TAG, and END_TAG configuration options, and with the TAGS directive. We'll also be looking at this shortly.

3.1.1.1 Template parser

All of this happens inside a part of the Template Toolkit called the *parser* (implemented in the Template::Parser module, and assisted by various others including Template::Grammar and Template::Directives). The job of the parser is to scan the source template to figure out which parts are text and which are directives, taking all the relevant configuration options and any values set by the TAGS directive into account. Having worked out where the directive tags are, it then parses the statements within them, checking that their syntax and structure are correct. If they aren't, the parser returns a parse error along with a short message explaining the problem.

3.1.1.2 Parse errors

We can demonstrate a parse error by having *tpage* process the template in Example 3-1, which contains an erroneous directive. The mandatory template filename after the PROCESS keyword is missing.

Example 3-1. badfile

```
[% # this is an invalid directive
   # and will raise a parse error
   PROCESS
%]
```

This is what happens when we run *tpage*:

```
$ tpage badfile
file error - parse error - badfile line 1-4:
  unexpected end of directive
  [% # this is an invalid directive
     # and will raise a parse error
     PROCESS
  %]
  at /usr/bin/tpage line 60.
```

We've edited the output a little for the sake of clarity, but all the important parts are there. The message tells us what kinds of errors occurred (in this case, a general file error and a parse error), what the error was (unexpected end of directive), and where it occurred (badfile line 1-4). It also shows the offending directive and reports the line number in the *tpage* program where the error was raised (at /usr/bin/tpage line 60).

3.1.1.3 Caching templates

If the template content is valid, the parser *compiles* it into a Perl subroutine that faithfully reproduces its exact functionality. Although the subroutine takes a little time to parse and compile the template into the equivalent Perl code, it is more than paid back by the speed at which it then runs. The great benefit of this approach is that the compiled template (i.e., the Perl subroutine) can be *cached* internally by the Template Toolkit for subsequent reuse. It keeps hold of the subroutine for each template that gets compiled so that it doesn't have to do the hard job of parsing and compiling it again the next time you want to use it.

This caching lasts for the lifetime of the Perl Template object being used. When you run *ttree* to build all the pages in a web site, for example, one Template object is used throughout. Every page can call the *menuitem* template a dozen times, for instance, but it will only be parsed and compiled the first time it is used. This is also ideal when you're using the Template Toolkit to serve dynamic pages from a persistent web server process (i.e., Apache and mod_perl). In contrast to a CGI script, which is restarted each time it is used and must create a new Template object each time, an Apache mod_perl handler can reuse a shared Template object, allowing the compiled templates to remain cached and ready to be used over and over again.

3.1.1.4 Flexible syntax

The job of parsing a template document is not an easy one. The Template Toolkit parser tries to be as flexible as possible with regard to the syntax and structure of directive tags. It doesn't complain if you forget (or choose not) to put a comma between items in a list, for example. As long as there's some kind of whitespace to separate them and the meaning isn't ambiguous, it will work around you so that you don't have to work around it.

Understandably, there are some basic rules that you'll need to follow, as well as some general guidelines that can help to make your templates easier to read and write. This section covers them in detail and shows the various ways in which the default behavior can be modified through the use of configuration options and other means.

As long as you follow the basic rules, the matter of how you lay out your directives, incorporating whitespace, formatting, and comments, is very much one of personal taste. You don't have to lay out your templates (or Perl code) nicely at all if you don't want to, but you will appreciate it when you come back to them after an absence and have to try and figure out what is going on. Anyone else who has to maintain your templates will also appreciate your efforts in making them as simple and clear as possible.

3.1.2 Template Tags

The default characters that the Template Toolkit uses to denote the position of directive tags are [% and %].

We saw an example in Chapter 2 showing how the TAGS directive can be used to set a different tag style for a single template file:

```
[% TAGS star %]

People of [* planet *], your attention please.
```

The tag style can be changed any number of times within a template and will revert to the current default at the end.

Figure 3-1 shows a list of the different tag styles available.

Figure 3-1. Tag styles

Tag style	Start tag	End tag
template	[%	%]
template1	[% or %%	%] or %%
metatext	%%	%%
html	<!--	-->
mason	<%	>
asp	<%	%>
php	<?	?>
star	[*	*]

Custom start and end tags can be set using the two-argument form of the TAGS directive:

```
[% TAGS { } %]

People of {planet}, your attention please.
```

The TAGS directive should always be specified in a tag by itself. It is something of a special case for the parser and doesn't obey the usual rule for directives of allowing a semicolon to separate one statement from the next.

```
[% TAGS star;
   # don't do this... it doesn't work
   PROCESS header
%]
```

However, you can use the whitespace chomping flags in a TAGS directive:

```
[% TAGS star -%]
[* PROCESS header -*]
```

The Template Toolkit provides the TAG_STYLE configuration option for setting a named tag style from Perl:

```
my $tt = Template->new({
    TAG_STYLE => 'star',
});
```

If you can't find an existing style you like, you can define custom start and end tags using the START_TAG and END_TAG options:

```
my $tt = Template->new({
    START_TAG => quotemeta('[*'),
    END_TAG   => quotemeta('*]'),
});
```

The START_TAG and END_TAG options support Perl regular expressions, giving you precise control over exactly what you want to match. One side effect of this is that any regular expression metacharacters (such as [and *) will need to be explicitly escaped with a \ prefix (e.g., '\[*') or passed through Perl's quotemeta function, as shown in the previous example.

The next example shows how regular expressions can be used for the START_TAG and END_TAG options:

```
my $tt = Template->new({
    START_TAG => '<(?i:tt):',
    END_TAG   => '/?>',
});
```

Here we allow the <tt: prefix to be specified in uppercase, lowercase, or mixed case (the (?i:...) part of the START_TAG regular expression), and the END_TAG to permit an optional / before the closing >. The following fragment shows four tags in slightly different styles, all of which will be matched by the START_TAG and END_TAG regular expressions:

```
<tt:pi=3.142/>
<tt:e=2.718>
pi: <TT:pi>
 e: <TT:e/>
```

The TAG_STYLE option takes priority over any values for START_TAG and END_TAG, so it makes no sense to mix them in the same configuration. Use either TAG_STYLE or START_TAG and END_TAG.

3.1.3 Interpolated Variables

The INTERPOLATE option allows you to embed variables in plain text using a simple $variable or ${variable} syntax. It is disabled by default, but can be set to any true value as a configuration option to enable this behavior.

```
my $tt = Template->new({
    INTERPOLATE => 1,
});
```

With the INTERPOLATE option enabled, the following template fragments have the same effect:

```
# using explicit directives
<a href="[% path %]">[% page.title %]</a>


# using interpolated variables
<a href="$path">$page.title</a>
```

Variable names can contain dotted elements, as shown by $page.title in the preceding example. The explicit braces can be used to delimit a variable name where necessary.

For example:

```
<img src="/images/${icon.file}.png">
```

Without the explicit scoping, the parser would treat icon.file.png as the variable name:

```
# incorrect usage
<img src="/images/$icon.file.png">
```

You must also use braces to explicitly scope embedded variables if you want to pass arguments to any of the dotted elements:

```
<img src="/images/${icon.file(extension='png')}">
```

If you've got the INTERPOLATE mode set and want to use a $ character in your document *without* it triggering a variable lookup, escape it with a \ prefix to nullify its special meaning.

For example:

```
...costing less than one
   Altairian dollar (\$1.00 ALD)
   per day...
```

The backslash tells the parser to treat the $ that follows it as just that, a literal $ character, rather than trying to interpret it as the start of a reference to a nonexistant $1.00 variable. Rather surprisingly, 1.00 is a perfectly valid variable name, given that variables can be dotted, with each part being composed of any combination of letters, numbers, or underscores. You'll have a difficult job trying to use a variable called 1.00 because the Template Toolkit will assume that you really mean the floating-point number 1.00 whenever you try and use it. Nevertheless, it's enough to confuse the parser in this case, so the preceeding \ is used to clarify our meaning.

3.1.4 Comments

Comments can be added to directives, either to provide explanations of what's going on for future maintainers (i.e., you, in six months time, when you've forgotten what you did and why you did it), or to temporarily disable all or part of the directive for testing or debugging purposes.

The # character introduces a comment in a directive. Everything from the # to the end of the current line is ignored. Here's an example that would be cryptic (at best) without the liberal use of comments that we've afforded it:

```
[%  # Calculate whether year is a leap year
    # if it's evenly divisible by 4...
    IF (year % 4) =  = 0;
        # if it is not a century year...
        IF (year % 100) =  = 0;
            is_leap = 1;    # it's a leap year
        # if it is a century year and divisible by 400 ...
```

```
        ELSIF (year % 400) =  = 0;
            is_leap = 1;     # it's a leap year
        END;
    END;
  %]
```

Comments can begin at the start of a line or part of the way through it. In either case, once you've started a comment on a line, there's no turning back. The rest of the line is a comment, and there's no character that will put you back into "uncommenting" mode.

If the # comment character immediately follows the [% start tag (or the appropriate value for the start tag if you're using something other than the default), with no intervening whitespace, the whole directive is treated as one big comment and is totally ignored. This can be used to temporarily disable an entire directive tag.

```
[%# this is broken, so disable it...
     IF skateboarding;
         kickflip(
             rotation  = 180,
             direction = 'backside'
         );
     END
%]
```

The first # character in the preceding directive temporarily disables the entire block of code. When and if we want to use it again, we can simply remove the leading comment line, or add a space between the [% and # to make it a single-line comment:

```
[% # this is working again!
     IF skateboarding;
         kickflip(
             rotation  = 180,
             direction = 'backside'
         );
     END
%]
```

There's not a lot to distinguish between these two examples, so be aware of the big difference that a single space can make.

3.1.5 Whitespace Chomping

Anything outside a directive tag is considered fixed text and is passed through unaltered. This includes all whitespace and newline characters surrounding directive tags. Directives such as SET and BLOCK that don't generate any output by themselves will leave gaps in the output document.

For example:

```
Foo
[% a = 10 %]
Bar
```

The newline following the directive is left intact, resulting in the following output:

Foo

Bar

This generally isn't a problem when you're generating HTML, which treats whitespace as (mostly) irrelevant. However, it will be of greater concern when generating plain-text documents or other formats in which whitespace is significant.

3.1.5.1 Chomping flags

The - chomping flag can be placed immediately after an opening directive tag (e.g., [% or the current value for the start tag) to have the Template Toolkit remove the newline and any other whitespace immediately preceding the directive tag. This is called *prechomping*.

Here is a trivial example to illustrate:

```
Foo

[%- 'Bar' %]

Baz
```

The template is parsed as if written:

```
Foo[% 'Bar' %]

Baz
```

and therefore generates the following output:

```
FooBar

Baz
```

As you might expect, you can also place a - immediately before the closing directive tag (e.g., %] or the current value for the end tag) to enable *postchomping*.

The following example:

```
Foo

[% 'Bar' -%]

Baz
```

is parsed as if written:

```
Foo

[% 'Bar' %]Baz
```

and generates the following output:

```
Foo

BarBaz
```

Both prechomping and postchomping flags can be set for a directive, as shown in the following example, which generates the output FooBarBaz:

```
Foo

[%- 'Bar' -%]

Baz
```

3.1.5.2 Chomping options

You can set the PRE_CHOMP and POST_CHOMP options to enable prechomping and postchomping as the default for all directives:

```
my $tt = Template->new({
    PRE_CHOMP  => 1,
    POST_CHOMP => 1,
});
```

With these options set, the following example:

```
Foo
[% 'Bar' %]
Baz
```

is equivalent to explicitly adding a - at the start and end of the tag:

```
Foo
[%- 'Bar' -%]
Baz
```

You can then use + in place of where the - would usually go if you want to *disable* the default prechomping or postchomping behavior on a per-directive basis. In other words, the + tells the Template Toolkit to not chomp the whitespace coming before or after a directive, regardless of the current settings of the PRE_CHOMP and POST_CHOMP options.

```
Foo
[%+ 'Bar' +%]
Baz
```

To summarize, the PRE_CHOMP and POST_CHOMP options define the default behavior, but the - and + options take priority on an individual directive basis.

The PRE_CHOMP and POST_CHOMP options also support a different style of chomping that you can enable by setting their values to 2 instead of 1. Instead of removing the whitespace entirely, it is *collapsed* into a single space.

3.1.5.3 Chomping constants

The Template::Constants module defines an exportable set of constants, CHOMP_NONE (0), CHOMP_ALL (1), and CHOMP_COLLAPSE (2), that you can use to make your code more readable. They are loaded into a Perl program when you use the Template::Constants module, providing the quoted name :chomp as an argument. The following example demonstrates this, and shows how the CHOMP_COLLAPSE constants can then be used:

```
use Template;
use Template::Constants qw( :chomp );


my $tt = Template->new({
    PRE_CHOMP  => CHOMP_COLLAPSE,
    POST_CHOMP => CHOMP_COLLAPSE,
});
```

When the following template is processed:

```
Foo
[% 'Bar' %]
Baz
```

it is parsed as if written:

```
Foo [% 'Bar' %] Baz
```

and therefore generates the following output:

```
Foo Bar Baz
```

The + flags have the same effect of protecting whitespace around a directive regardless of the PRE_CHOMP or POST_CHOMP option being set to CHOMP_ALL or CHOMP_COLLAPSE.

3.1.6 Multiple Directive Tags

When you start to use more complex directives, you may find your templates start to look a little cluttered, as Example 3-2 shows.

Example 3-2. printer1

```
[% IF title %]
  [% IF printer_friendly %]
    [% INCLUDE headers/printer_friendly %]
  [% ELSE %]
    [% INCLUDE headers/standard %]
  [% END %]
[% END %]
```

The default tag style is designed to make the directives stand out from the rest of the document. However, the [% and %] characters overwhelm the important part of this example, the content of the various directives, making the template harder to both read and write.

Fortunately, the Template Toolkit has been around long enough for people to get bored of typing [% and %] and demand a better solution. The answer is to merge the directives into one tag, using the ; (semicolon) character to delimit one directive statement from the next.

Example 3-3 demonstrates this, showing how much simpler Example 3-2 can be written.

Example 3-3. printer2

```
[% IF title;
  IF printer_friendly;
    INCLUDE headers/printer_friendly;
  ELSE;
    INCLUDE headers/standard;
  END;
END
%]
```

When you merge directives together, you lose any whitespace that might previously have been nestling between the directives. That may be what you want. If it isn't, you can easily add it back where you need it by adding literal strings, including any text and whitespace required, as part of the directive block. This is shown in Example 3-4.

Example 3-4. person1

```
[% FOREACH person IN company.employees;
  "* ";
  person.name;
  "\n  ";
  person.email;
  "\n\n";
END
```

```
END

%]
```

With a "double-quoted" string, the \n sequence introduces a newline character. So given the following definition for company:

```
[% company = {

     employees = [

       { name = 'Tom' email = 'tom@tt2.org' },

       { name = 'Dick' email = 'dick@tt2.org' },

       { name = 'Larry' email = 'larry@tt2.org' },

     ]

   }

%]
```

the output generated by Example 3-4 would be:

```
* Tom

  tom@tt2.org



* Dick

  dick@tt2.org



* Larry

  larry@tt2.org
```

3.1.7 Side-Effect Notation

The IF, UNLESS, FOREACH, WHILE, WRAPPER, and FILTER directives expect a template block to follow them, up to the relevant END directive (or ELSIF or ELSE in the case of IF and UNLESS). They can also be used in a "side-effect" notation. This is a concept borrowed from Perl in which looping or conditional logic can be placed *after* the statement that it controls. Here is an example:

```
[% PROCESS config IF something %]
```

The equivalent code, writing the directive in full, would look like this:

```
[% IF something;

     PROCESS config;

   END

%]
```

It works only when you've got one variable, directive, or piece of text that you want to use in the block. This isn't the case in Example 3-4, which we looked at in the previous section. However, Example 3-5 shows how it can be rewritten to define the block as one double-quoted string, using variable interpolation to insert the values for person.name and person.email in the right place.

Example 3-5. person2

```
[% FOREACH person IN company.employees;

  "* $person.name\n  $person.email\n\n";

END

%]
```

With a single string as the content for the block, FOREACH can now be used in side-effect notation, as shown in Example

3-6.

Example 3-6. person3

```
[% "* $person.name\n  $person.email\n\n"
  FOREACH person IN company.employees
%]
```

More complex content can be moved into a separate template file or BLOCK definition that is then called using a single PROCESS or INCLUDE directive, as shown in Example 3-7.

Example 3-7. person4

```
[% PROCESS info
  FOREACH person IN company.employees
%]

[% BLOCK info %]
  * [% person.name %]
  [% person.email %]
[% END %]
```

3.1.8 Capturing Directive Output

The output of a directive can be captured by assigning it to a variable. The following example shows this in action:

```
[% headtext = PROCESS header
              title = "Hello World"
%]
```

In the next example, it is used to capture the output of a side-effect block:

```
[% people = PROCESS userinfo
            FOREACH user = userlist
%]
```

It can also be used in conjunction with the BLOCK directive for defining large blocks of text or other content:

```
[% quote = BLOCK %]
   'Where,' said Ford Prefect quietly,
   'does it say teleport?'

   'Well, just over here in fact,'
   said Arthur, pointing at a dark
   control box in the rear of the cabin.

   'Just under the word "emergency",
   above the word "system" and beside
   the sign saying "out of order".'
```

```
[% END %]
```

Note one important caveat of using this syntax in conjunction with side-effect notation. The following directive does not behave as might be expected:

```
[% # WRONG

  description =  'Mostly Harmless'

    IF planet =  = 'Earth'

%]
```

Our intention is to set the description variable (using the single equals assignment operator, =) to the value Mostly Harmless if the planet variable contains the value Earth (tested using the double equals comparison operator, = =):

```
[% # RIGHT

  IF planet =  = 'Earth';

     description = 'Mostly Harmless';

  END

%]
```

Unfortunately, that's not how the Template Toolkit parser sees things. The directive is interpreted as if written:

```
[% # WRONG

  description = BLOCK;

    IF planet =  = 'Earth';

      'Mostly Harmless';

    END;

  END

%]
```

The variable is assigned the output of the IF block. This returns Mostly Harmless correctly for planet Earth, but nothing in all other cases, resulting in the description variable being unintentionally cleared.

To achieve the expected behavior, the directive should use the SET keyword explicitly:

```
[% # RIGHT

  SET description =  'Mostly Harmless'

        IF planet =  = 'Earth'

%]
```

3.1.9 Template Filenames

Like Perl, the Template Toolkit treats data differently depending on whether it is quoted. For example, foo.bar accesses the value in a variable, but 'foo.bar' is a literal string.

The INSERT, INCLUDE, PROCESS, and WRAPPER directives expect a filename to be pro vided as the first argument:

```
[% PROCESS header %]
```

You can use single or double quotes around the filename, but they're generally not required:

```
[% PROCESS 'header' %]

[% PROCESS "header" %]
```

The Template Toolkit assumes that the first argument is a filename, even if it includes dot characters:

```
[% PROCESS header.tt %]
```

If you do use double quotes around the string, any variable references within it will be interpolated. For example:

```
[% file = 'header'
   ext  = 'tt'
%]
[% PROCESS "${file}.$ext" %]   # header.tt
```

You'll also need to explicitly quote the filename if it contains any characters other than alphanumerics, underscores, dots, and slashes:

```
[% PROCESS no/need_2_quote/this.txt %]
[% PROCESS 'My Documents/q&a.txt' %]
```

If you want to use a variable value to denote the name of a file, you can interpolate it into a double-quote string:

```
[% file = 'header' %]
[% PROCESS "$file" %]    # header
```

As a convenience, you can do away with the double quotes and simply use the $ prefix to tell the parser that a variable name follows:

```
[% PROCESS $file %]      # header
```

3.2 Template Variables

The Template Toolkit's simple-to-access variables are one of its strengths. In this section, we describe the syntax and semantics of variables—what names are allowed, the different types of data that can be stored in a variable, the predefined Template Toolkit variables, and so on.

3.2.1 What's in a Name?

Variable names can contain alphanumeric characters or underscores. They can be lowercase, uppercase, or mixed case, although the usual convention is to use lowercase to avoid confusion with uppercase directives. The case *is* significant, however, so foo, Foo, and FOO are all different variables. Here are some examples of valid variable names:

```
foo
foo123
foo_bar
foo_bar_123
FooBar123
Foo_Bar_123
```

The kind of data you can store in a variable depends on its *type*. The Template Toolkit is written in Perl and provides template authors with access to the full range of underlying Perl variable types. Although there are different variable types for different purposes, you can change a Template Toolkit variable from one to the other at any time. Both Perl and the Template Toolkit are examples of *dynamic languages* that don't require the type of variable to be set in stone.

The basic data types are *scalars*, which store a single value, *arrays* (or *lists*), which store multiple values in order, and *hash arrays* (or *hashes*), which store multiple values indexed by a name. In addition to these *static* data types, the Template Toolkit provides *dynamic* data types that can reference Perl *subroutines*, and *objects* that can implement any kind of functionality you require to fetch or compute a variable value on demand.

Unlike Perl, the Template Toolkit does not require you to use a different leading character, or *sigil*, on a variable name to denote its type—e.g., $item, @list, %hash. In fact, it requires you not to do it. The only time you ever use a leading $ on a variable in a template is to tell the parser that a variable for interpolation follows where it otherwise wouldn't be expecting one—for example, in a double-quoted string such as "Hello $planet", or following a directive keyword that usually expects a filename, such as [% PROCESS $myfile %].

The $ prefix should *always* be used for variable interpolation, *regardless* of the underlying data type. For example, the string "$msg.greeting $planet.0" shows how $ is used to access a hypothetical hash value, msg.greeting, and also a list item, planet.0. In both cases, $ is used as the prefix.

3.2.2 Simple Data Types

The simplest variables are scalars that hold just one value:

```
[% answer = 42 %]
[% author = 'Douglas Adams' %]
```

The values are referenced in a template by embedding the variable name in a tag:

```
The answer to the Ultimate Question of Life, the
Universe and Everything is [% answer %].


                    -- [% author %]
```

The optional SET and GET directive keywords can be used when defining and subsequently retrieving variable values:

```
[% SET author = 'Douglas Adams' %]
[% GET author %]
```

However, you'll rarely see the GET and SET keywords used because the Template Toolkit allows you to omit them. The common use is to update and access variables directly, as shown here:

```
[% author = 'Douglas Adams' %]

[% author %]
```

Scalar variables can contain numbers or text strings that both the Template Toolkit and Perl treat as interchangeable. Strings are automatically converted into numbers and numbers into strings whenever one or the other is required.

```
The answer to the Ultimate Question of Life, the

Universe and Everything is 42.



                    -- Douglas Adams
```

You can set any number of variables in the same directive:

```
[% answer = 42

  author = 'Douglas Adams'

%]
```

You don't need a semicolon between each item in a SET list, but you will need one after the last item if other directives follow. Semicolons are always required to separate GET directives in the same tag:

```
[% answer = 42             # implicit SET..

  author = 'Douglas Adams';   #    ...continued

  answer;                   # implicit GET

  author;                   # implicit GET

%]
```

Numbers can be specified as integers or in floating-point format:

```
[% answer = 42

  pi     = 3.14

%]
```

String values can be enclosed in single quotes or double quotes and can span several lines:

```
[% author = 'Douglas Adams'

  book   = "The Hitch Hiker's Guide to the Galaxy"

  advice = "Don't Panic"

  about  = "On thursday lunchtime the Earth gets

           unexpectedly demolished to make way

           for a new hyperspace bypass..."

%]
```

Using single or double quotes can be a matter of convenience, such as in this example in which the values for book and advice contain apostrophes that would otherwise be mistaken for the closing single-quote character. However, the main reason for choosing double quotes over single quotes is to allow variable values to be embedded in the string.

In single quotes, the $ character is treated as a literal and has no special meaning:

```
[% price = '$4.20' %]
```

In double quotes, on the other hand, the $ is used to mark the start of a variable name:

```
[% summary = "$book by $author" %]

Summary: [% summary %]
```

The values of the $book and $author variables will be *interpolated* into the relevant places in the string:

```
Summary: The Hitch Hiker's Guide to the Galaxy by Douglas Adams
```

You can also embed dotted variables in double-quoted strings:

```
[% summary = "$book.title by $book.author" %]
```

The ${...} delimiters can be used to explicitly scope a variable name. You'll need this whenever you have a variable nestling up tight against a dot (.) or other characters that could be mistaken for part of the name.

```
[% webpage = "h2hg/chapter_${chapter.number}.html" %]
```

Watch out in particular for periods used to mark the end of a sentence. Without the ${ and } in place to scope the your.name variable in the next example, the template fails to compile and raises a parse error:

```
[% greeting = "Hello ${your.name}." %]   # GOOD

[% greeting = "Hello $your.name." %]     # BAD - parse error!
```

If you want to include a literal $ character in a double-quoted string, precede it with a \ (backslash) character to *escape* it from any special meaning:

```
[% language = 'Perl'

   pledge   = "Will hack $language for \$\$\$"

%]

I pledge: [% pledge %]
```

The backslash characters are removed, leaving the dollar signs ringing:

```
I pledge: Will hack Perl for $$$
```

You can also use the backslash character to escape any occurrences in the string of the quote character you're using, ' or ":

```
[% advice  = 'Don\'t Panic'

   suggest = "Read \"$book\" by ${author}."

%]

1) [% advice  %]

2) [% suggest %]
```

This is the output generated:

```
1) Don't Panic

2) Read "The Hitch Hiker's Guide to the Galaxy" by Douglas Adams.
```

One final use of the backslash is to embed special metacharacters in a double-quoted string. For example, the \n sequence indicates a newline, \r a carriage return, and \t a tab character:

```
[% blockquote = "$advice\n\t-- $author" %]
```

When the value of blockquote is displayed, a newline and tab character are printed in the correct place:

```
Don't Panic

        -- Douglas Adams
```

If you want a literal backslash character in either a single- or double-quote string, you'll need to escape it with another backslash:

```
[% dospath = "C:\\dos\\path" %]
```

It's ugly, but it works. The backslash is a relatively uncommon character (except in DOS filenames, as in this example), so it's not something you normally need to worry about.

3.2.3 Complex Data Types

In contrast to simple data types that hold only a single value, the Template Toolkit supports two complex data types for storing multiple values: the *list* and *hash*. A list is an ordered array of other variables, indexed numerically and starting at element 0. A hash is an unordered collection of other variables that are indexed and accessible by a unique name or *key*.

If you're using the Template Toolkit from Perl, you can define template variables that reference any existing hash and array data structures in your Perl program that you want to make accessible in the templates:

```
my $vars = {
    primes => [ 2, 3, 5, 7, 11, 13 ],
    terms  => {
        sass  => 'know, be aware of, meet, have sex with',
        hoopy => 'really together guy',
        frood => 'really, amazingly together guy',
    },
};

$tt->process($input, $vars)
    || die $tt->error( );
```

List and hash data structures can also be defined within templates using a syntax similar to the Perl equivalents shown earlier. The default syntax is actually a little simpler than in Perl, allowing = to be used in place of => and treating commas between items as optional. However, the Template Toolkit is also comfortable with data structures laid out "Perl-style" using => and commas. This is particularly useful if you're coming from a Perl background or trying to merge existing Perl data definitions into template code, or vice versa.

Let's look at the syntax for lists and hashes in more detail.

3.2.3.1 Lists

A list variable is defined in a template using the [...] construct. Individual elements can be separated with whitespace, commas, or any combination of the two. The following all create equivalent lists:

```
[% primes = [2,3,5,7,11,13] %]

[% primes = [ 2 3 5 7 11 13 ] %]

[% primes = [ 2, 3, 5, 7, 11, 13 ] %]

[% primes = [ 2, 3 5, 7 11, 13 ] %]
```

The elements can be literal number or string values, or can reference other variables:

```
[% two    = 2
   three  = 3
   primes = [ two, three, 5, 7, 11, 13 ]
%]
```

You can also use the .. operator to create a range of values. Whitespace is optional on either side of it.

```
[% one_to_four = [ 1..4 ] %]
```

The values in a range can also be specified using variables:

```
[% start = 1
   end   = 4
   items = [ start .. end ]
%]
```

List elements are accessed using the *dot operator*. The list name is followed by the . character and then the element number.

```
[% primes.0 %]     # 2

[% primes.3 %]     # 7
```

Like Perl, the first element of a list is element 0, not element 1, meaning that primes.3 is the *fourth* element in the list, not the third. If this is confusing, it might help if you think of this number as an offset from the beginning of the list, rather than as the element number.

3.2.3.2 Hashes

A hash variable is defined in a template using the {...} construct:

```
[%  terms = {

        sass  = 'know, be aware of, meet, have sex with'

        hoopy = 'really together guy'

        frood = 'really, amazingly together guy'

    }

%]
```

Each entry in a hash is composed of a pair of values. The first is the key through which the second, the value, will be indexed in the hash. You can use either = or => to separate the key from the value. As with lists, commas can be used to delimit each pair but are not required.

```
[%  terms = {

        sass  => 'know, be aware of, meet, have sex with',

        hoopy => 'really together guy',

        frood => 'really, amazingly together guy',

    }

%]
```

Hash items are also accessed using the dot operator. In this case, the key for the required item is specified after the . character:

```
[% terms.hoopy %]    # really together guy
```

If you assign a value to an element in a hash that doesn't yet exist, it will *autovivify* the parent hash and any intermediate hashes so that the variable just springs into life when you first use it:

```
[% foo.bar.baz = 'hello world' %]
```

In this example, the foo hash and nested bar hash will be created automatically (assuming they didn't already exist), and bar will contain a baz item assigned the value hello world.

3.2.4 Dot Operator

We've already seen some simple examples of using the dot operator to access elements of complex variables. In the case of a list, an integer follows the dot operator to reference a particular item in the list. Remember that lists start counting their elements at 0, not 1, so the following directive fetches the *fourth* item in the primes list—in this case, the number 7:

```
[% primes = [2, 3, 5, 7, 11, 13] %]

[% primes.3 %]  # 7
```

For hash arrays, the dot operator is followed by the key for the item required:

```
[% terms = {

     sass  = 'know, be aware of, meet, have sex with'

     hoopy = 'really together guy'

     frood = 'really, amazingly together guy'

   }

%]

[% terms.hoopy %]
```

3.2.4.1 Compound dot operations

A variable reference can include many dot operators chained together to access data nested deeply in a complex data structure.

Here's an example of some nested data:

```
[% arthur = {
     name    = 'Arthur Dent',
     planet  = 'Earth',
     friends = {
       ford  = {
         name = 'Ford Prefect'
         home = 'Betelgeuse'
       }
       slarti = {
         name = 'Slartibartfast'
         home = 'Magrethea'
       }
     }
   }
%]
```

The following compound variables access different parts of the data structure, returning the values shown as comments to the right:

```
[% arthur.friends.ford.name %]   # Ford Prefect
[% arthur.friends.slarti.home %] # Magrethea
```

3.2.4.2 Interpolated variables names

The Template Toolkit uses the $ character to indicate that a variable should be interpolated in position. Most frequently, you see this in double-quoted strings:

```
[% fullname = "$honorific $firstname $surname" %]
```

or embedded in plain text when the INTERPOLATE option is set:

```
Dear $honorific $firstname $surname,
```

The same rules apply within directives. If a variable or part of a variable is prefixed with a $, it is replaced with its value before being used. The most common use is to retrieve an element from a hash where the key is stored in a variable.

We saw an example of this in Chapter 2:

```
[%  terms = {
      sass  = 'know, be aware of, meet, have sex with'
      hoopy = 'really together guy'
      frood = 'really, amazingly together guy'
    }
%]
[% key = 'frood' %]
[% terms.$key %]     # really, amazingly together guy
```

The value for key is interpolated into the terms.$key expression, resulting in the correct value being displayed for terms.frood.

Curly braces can be used to delimit interpolated variable names where necessary. For example:

```
[% ford = {

      name = 'Ford Prefect'

      type = 'frood'

   }

%]


[% ford.name %] is a [% terms.${ford.type} %]
```

3.2.4.3 Private variables

In Perl, it is common practice to use a leading underscore before the names of variables in an object hash to indicate those that should be considered "private" and not for use outside of the object methods. The Template Toolkit honors this and will not return any item from a hash array or object whose name begins with _ or . (which could be confused with the dot operator).

```
[% stuff = {

   _private = "You won't see me"

    public  = "You will see me"

   }

%]

[% stuff.public   %] # You will see me

[% stuff._private %] # [nothing]
```

Any attempts to retrieve these values, even indirectly by use of a variable key, will return the empty string, indicated in these examples as [nothing]:

```
[% var = "_private";

   stuff.$var        # [nothing]

%]
```

3.2.5 Dynamic Data Types

The common feature of scalars, lists, and hash arrays is that they contain *static* values. What this means in the context of template processing is that they contain pre-defined values that don't change from one minute to the next *unless* you specifically update the variable. In other words, the value is "there for the taking" once set, and can be inserted directly into a template without requiring any additional computation.

A *dynamic* value, on the other hand, is one that is computed *each* time it is used. The Template Toolkit allows template variables to be bound to Perl subroutines and objects. When the variable is accessed, the subroutine or appropriate object method is called and can perform whatever operation or calculation is required to return a value. The value returned can be different each time and may depend on any number of different factors. Hence the name *dynamic*.

Static and dynamic variables are accessed using exactly the same dotted notation. You don't need to change your templates if you decide to one day switch from using a static hash array to a dynamic subroutine that fetches some data from a database and returns a generated hash, for example. These are the kinds of implementation details that the Template Toolkit hides from you so that your templates can remain simple and portable.

Using dynamic variables when calling the Template Toolkit from Perl is as simple as passing references to subroutines or objects:

```
use CGI;


my $vars = {

    prime_number => sub {
```

```
        # return a random prime number from first 6
        my @primes = (2, 3, 5, 7, 11, 13);
        return $primes[ rand @primes ];
    },

    cgi => CGI->new( ),
};

$tt->process($input, $vars)
    || die $tt->error( );
```

There is no way to define new subroutines or objects directly in a template without resorting to embedding Perl code using the PERL or RAWPERL directives (and enabling the EVAL_PERL option, of course). However, the Template Toolkit plugin architecture allows you to define plugins that *can* be loaded directly into a template to define new subroutine and object variables. This will be covered in detail in Chapter 6.

3.2.5.1 Subroutines

The subroutine bound to a template variable will be invoked each time the value is required, in a GET directive, for example, or perhaps for interpolating into a string:

```
[% prime %]                    # calls subroutine
[% more = "$prime $prime $prime" %]  # three calls
```

The subroutine returns a value for the template variable, in this case returning a random choice of one of the first six prime numbers. Each time the variable is used, the subroutine is called and a different value returned. Of course, the nature of random numbers is such that the same value could actually be returned any number of times in the example. However, the important fact is that the value is computed each time, and any similarity between the values returned for any particular invocations is coincidental.

3.2.5.2 Objects

A variable can also be bound to a Perl object whose methods can be invoked using the same dotted notation as for accessing elements in a hash array:

```
This CGI script is running on [% cgi.server_name %]
```

The use of identical syntax for accessing hash items and object methods is an intentional and powerful feature of the Template Toolkit language. The *Uniform Access Principle* hides the implementation details behind an abstract notation that effectively "does the right thing" for whatever kind of data you're using. It provides a *clear separation of concerns* between the *representation* and *presentation* of the data, allowing one to change without affecting the other.

3.2.5.3 Passing arguments

Arguments can be passed to subroutines or object methods called from a template by adding them in parentheses immediately after the variable name. The following example shows how the literal string value docid is passed to the param( ) method of the CGI object bound to the cgi variable:

```
[% cgi.param('docid') %]
```

Here's an example of a subroutine that takes a list of arguments and returns them joined together in a single string, delimited by a comma and space:

```
my $vars = {
    join => sub {
        return join(', ', @_);
```

```
    },

};



$tt->process($input, $vars)

    || die $tt->error( );
```

Any number of arguments can be passed to the subroutine, either as numbers, as literal strings, or by referencing other variables. This is shown in Example 3-8.

Example 3-8. join

```
[% ten    = 10

   thirty = 30;

   join(ten, 20, thirty, '40')

%]
```

The output generated by Example 3-8 is as follows:

```
10, 20, 30, 40
```

3.2.5.4 Pointless arguments

Strictly speaking, you can pass arguments to any template variable, even if the variables aren't defined as references to subroutines or objects:

```
[% arthur = {

     name    = 'Arthur Dent',

     planet  = {

        name = 'Earth',

        info = 'Mostly Harmless'

     }

   }

%]



[% arthur(6).planet(7).name(42) %]  # Earth
```

In this example, the data structure is entirely static. There are no subroutines or objects lurking around that might make use of the arguments, so they are silently ignored. However, it illustrates the basic principle that any variable component can be provided with parenthesized parameters.

Providing arguments for variables that ignore them is not entirely pointless. When you're designing the look and feel of a web site, for example, you can define some simple, static data to use as "dummy" values for the page content. If you plan to implement some of these data items using subroutines or objects, you can go ahead and add any relevant parameters now so that you don't have to update your templates when the data model changes.

3.2.5.5 Named parameters

Named parameters can also be passed to subroutines and object methods. These are automatically collated into a hash reference and passed as the *last* argument to the subroutine or method.

```
my $vars = {

    join => sub {

        # look for hash ref as last argument
```

```
        my $params = ref $_[-1] eq 'HASH' ? pop : { };
        my $joint  = $params->{ joint };
        $joint = ', ' unless defined $joint;
        return join($joint, @_);
    },
};

$tt->process($input, $vars)
    || die $tt->error( );
```

Example 3-9 shows a named parameter, joint, provided in addition to the positional arguments, ten, 20, thirty, and '40'.

Example 3-9. joint

```
[% ten    = 10
   thirty = 30;
   join(ten, 20, thirty, '40', joint = '+')
%]
```

The output generated by Example 3-9 is as follows:

```
10+20+30+40
```

Named parameters can be specified anywhere in the argument list:

```
[% join(joint='+', ten, 20, thirty, '40') %]
[% join(ten, 20, joint='+', thirty, '40') %]
```

They are automatically removed from the list of positional arguments and passed to the subroutine or object method as the last argument, bound together in a single hash array reference. For this reason, and for the sake of clarity, we recommend that you always specify named parameters at the end of the list:

```
[% join(ten, 20, thirty, '40', joint='+') %]
```

In all these examples, the subroutine bound to the join variable would be called with the following list of arguments:

```
(10, 20, 30, 40, { joint => '+' })
```

In this subroutine, we look to see whether the last argument is a reference to a hash array. If it is, we pop it from the list. Otherwise, we create an empty Perl hash reference for $params.

```
# look for hash ref as last argument
my $params = ref $_[-1] eq 'HASH' ? pop : { };
```

We then look for the joint item in the named parameter hash and provide a sensible default if it isn't defined:

```
my $joint  = $params->{ joint };
$joint = ', ' unless defined $joint;
```

The subroutine calls Perl's join function, passing the $joint value along with the rest of the argument list. The resulting string is then returned:

```
return join($joint, @_);
```

Arguments can be passed to any variable, even those that are set to static values and have no use for an argument. In this case, they are simply ignored. As such, the following code:

```
[% meaning_of_life = 42 %]
[% meaning_of_life("Monday") %]
```

produces:

```
42
```

The argument "Monday" is ignored when the value of meaning_of_life is evaluated. The static value, 42, is simply inserted in its place.

3.2.5.6 Mixing dynamic and static data

Static and dynamic data structures can be freely intermixed. Static lists and hash arrays can contain references to dynamic subroutines and object methods. These can return complex data structures, including any combination of scalars, hash arrays, lists, subroutines, and object references.

```
my $vars = {
    zero => sub {
        return {
            one => sub {
                return [ $obj1, $obj2, $obj3 ],
            },
        };
    },
};
```

Compound dot operations work with dynamic data items exactly as they do for static ones. A series of dot operations can be chained together into a single expression to fetch an item from deep within a data structure, some or all of which might be computed on demand.

```
[% zero.one.2.three %]
```

In this example, zero is bound to a subroutine that returns a reference to a hash array. This contains another subroutine, one, which returns a list of objects. We take the third object, $obj3 (yes, the third, don't forget they start at 0), and call the three( ) method against it. Other than knowing that one returns a list (and so requires an index number—e.g., 2) and the others are hashes or objects (requiring index keys—e.g., one and three), we can remain blissfully ignorant of any of the underlying implementation details.

Furthermore, there's nothing to stop you from changing the one subroutine to return a hash array (or object) that contains the items (or methods) 0, 1, 2, and so on:

```
my $vars = {
    zero => sub {
        return {
            one => sub {
                return {
                    0 => $obj1,
                    1 => $obj2,
                    2 => $obj3,
                },
            },
        };
    },
};
```

It probably isn't something that you would want to do that often, but it does illustrate the point that all data types are equal as far as the dot operator is concerned. The following fragment continues to work unmodified, with 2 now being treated as a hash key instead of a list index:

```
[% zero.one.2.three %]
```

3.2.5.7 Returning values

A subroutine or object method can return any kind of value when called. Hash arrays and lists should be returned using references rather than a list of multiple items.

```
my $vars = {
    moregood => sub {
        return [ 3.14, 2.718 ];
    },
    lessgood => sub {
        return ( 3.14, 2.718 );
    },
};
```

If your subroutine does return multiple values, the Template Toolkit will automatically combine them into a list reference. This isn't the recommended usage, but it provides some level of support for existing Perl code that wasn't written with the Template Toolkit in mind.

```
# both work as expected
[% moregood.0 %] [% moregood.1 %]
[% lessgood.0 %] [% lessgood.1  %]
```

If you're writing new subroutines and methods from scratch, we suggest that you return a reference to a list rather than a list of items whenever possible. Be warned that if you do return a list of items, the first of which is undefined, the Template Toolkit will assume an error has occurred and raise it as such:

```
return (undef, ...);   # NOT OK: undef indicates error!
```

If you want to return a list of items that contains an undefined value as the first element, you should always return it as a reference to a list:

```
return [undef, ...];   # OK, returns list reference
```

3.2.5.8 Error handling

Errors can be reported from subroutines and object methods by calling die( ). This example shows a subroutine that dies as soon as it is called:

```
my $vars = {
    barf => sub {
        die "a sick error has occurred\n";
    },
};
```

If we process a template containing a reference to the barf variable, like so:

```
I think I'm going to [% barf %]
```

the Template process( ) method will return a false value and the error( ) method will report:

```
undef error - a sick error has occurred
```

Errors raised by calling die are caught by the Template Toolkit and converted to a Template::Exception object that includes the error message (a sick error has occurred) and an error type (undef). To throw an exception of a type other than the default undef, Perl code should die( ) with a reference to a Template::Exception object.

```
use Template::Exception;


my $vars = {
    barf => sub {
        die Template::Exception->new( sick => 'feel ill' );
```

```
    },
};
```

Now when the variable is accessed and the subroutine invoked, the error reported will be:

```
sick error - feel ill
```

Exceptions can be caught within templates using the TRY / CATCH directive construct:

```
[% TRY;
     barf;
  CATCH sick;
     "Eeew!  We just caught a sick error ($error.info)";
  END
%]
```

In this example, the sick error will be caught by the CATCH block, generating the following output:

```
Eeew!  We just caught a sick error (feel ill)
```

In this case, the process( ) method will return a true value. The error has been caught and dealt with, and as far as we're concerned, the template was processed successfully. Any exceptions of other types will still be passed through unless we add other CATCH blocks to catch them. This ensures that anything besides a sick exception will not be caught here.

The exception types 'stop' and 'return' are used to implement the STOP and RETURN directives. Throwing an exception as:

```
die (Template::Exception->new('stop'));
```

has the same effect as the directive:

```
[% STOP %]
```

See Chapter 4 for further information on error handling and flow control directives.

3.2.6 Special Variables

The Template Toolkit defines a number of special variables. Some, such as template and component, are universally defined and can be accessed from anywhere. Others, such as loop and content, are available only in a particular context, such as inside a FOREACH block (loop) and in a template loaded into another using the WRAPPER directive (content).

There's nothing to stop you from creating your own variables with the same name. In that case, they will simply mask the special variables provided by the Template Toolkit. However, if you define your own variable called loop, for example, it will be masked by the special variable provided in a FOREACH loop. However, the original value for your loop variable will be restored at the end of the FOREACH block.

The special variables defined by the Template Toolkit are covered in the sections that follow.

3.2.6.1 template

The template variable contains a reference to the main template being processed. It is implemented as a Template::Document object, described in detail in Chapter 8. The template variable is correctly defined within templates that are processed via the PRE_PROCESS, PROCESS, WRAPPER, and POST_PROCESS configuration options. This allows standard headers, footers, and other user interface templates to access metadata about the main page template being processed, even before it is processed.

The name and modtime metadata items are automatically defined, providing the template name and modification time in seconds since January 1, 1970 (the Unix *Epoch*), respectively. Any other items defined in META tags in the template will also be available via the appropriately named method.

For example, if the main page template defines the following:

```
[% META title  = 'My Test Page'
        author = 'Arthur Dent'
%]
```

a *header* template, defined as a PRE_PROCESS option, can access the template.title and template.author variables:

```
<html>
  <head>
    <title>[% template.title %]</title>
  </head>

  <body>
   <h1>[% template.title %]</h1>
   <h2>by [% template.author %]</h2>
```

Note that the template variable always references the main page template, regardless of any additional template components that may be processed.

3.2.6.2 component

The component variable is like template but always contains a reference to the current template component being processed.

This example demonstrates the difference:

```
$tt->process('foo')
   || die $tt->error( ), "\n";
```

A F<foo> template:

```
[% template.name %]           # foo
[% component.name %]           # foo
[% PROCESS footer %]
```

A F<footer> template:

```
[% template.name %]           # foo
[% component.name %]           # footer
```

In the main page template, *foo*, the template and component variables both reference the same Template::Document object, returning a value of foo for both template.name and component.name. In the *footer* template, the template variable remains unchanged, but the component now references the Template::Document object for the *footer* and returns the value of footer for component.footer accordingly.

3.2.6.3 loop

Inside the block of a FOREACH directive, the loop variable references a special object called an *iterator*, which is responsible for controlling and monitoring the execution of the loop. The following example shows it in use:

```
[% FOREACH item IN items %]
   [% IF loop.first %]
     <ul>
   [% END %]
       <li>[% item %] ([% loop.count %] of [% loop.size %])</li>
   [% IF loop.last %]
     </ul>
   [% END %]
[% END %]
```

The loop variable is implemented by a Template::Iterator object. It provides methods such as first and last, shown in the previous example, which return true only on the first and last iteration of the loop. The count method returns the current iteration count, starting at one (use index to get the real index number, starting at zero). The size method returns the size of the list.

The loop iterator is covered in detail in the discussion of the FOREACH directive in Chapter 4.

3.2.6.4 error

The Template Toolkit provides the TRY...CATCH construct to allow you to catch (and throw) runtime errors in your templates. Within a CATCH block, the error variable contains a reference to the Template::Exception object thrown from within the TRY block. The type and info methods can be called against it to determine what kind of error occurred and what (hopefully) informative error message was reported.

```
[% TRY %]
  ...some template code that
     may throw an error...
[% CATCH %]
  An error occurred:
     [% error.type %] - [% error.info %]
[% END %]
```

For convenience, the error variable can be referenced by itself and it will automatically be presented as a string of the form $type error - $info:

```
[% TRY;
    THROW food 'cheese roll';
  CATCH;
    error;    # food error - cheese roll
  END
%]
```

The TRY, CATCH, and other related directives are covered in detail in Chapter 4. For further information about the Template::Exception object, see Chapter 8 and the Template::Exception manpage.

3.2.6.5 content

The content variable is used by the WRAPPER directive to pass the output generated by processing the WRAPPER content block to the wrapping template. Example 3-10 shows it in action.

Example 3-10. content

```
[% scared = 'afeared'
  beats  = 'noises'
  vibes  = 'sweet airs'
  chill  = 'give delight'
-%]

[% WRAPPER box border=1 %]
  Be not [% scared %]; the isle is full of [% beats %],
  Sounds and [% vibes %], that [% chill %] and hurt not.
```

```
[% END -%]


[% BLOCK box -%]

<table border="[% border %]">

  <tr>

    <td>

      [%- content -%]

    </td>

  </tr>

</table>

[% END -%]
```

In the first section, we define some simple variables:

```
[% scared = 'afeared'

   beats  = 'noises'

   vibes  = 'sweet airs'

   chill  = 'give delight'

-%]
```

This is a rather contrived way of illustrating how the WRAPPER directive first processes the block following it, and up to the corresponding END directive, to resolve any directives embedded within. In this case, the values for the scared, beats, vibes, and chill variables are substituted into their correct places.

```
[% WRAPPER box border=1 %]

    Be not [% scared %]; the isle is full of [% beats %],

    Sounds and [% vibes %], that [% chill %] and hurt not.

[% END -%]
```

The WRAPPER directive then calls the *box* template as if it were an INCLUDE directive. In addition to any local variables specified with the WRAPPER (border in this example), it also sets the content variable to contain the processed block output. Here content contains the completed quote from "Be not afeard..." through "...give delight and hurt not".

In the BLOCK box defined at the end of the example, the content variable is referenced like any other, along with the border variable passed in as an explicit argument to the WRAPPER directive:

```
[% BLOCK box %]

<table border="[% border %]">

  <tr>

    <td>

      [%- content -%]

    </td>

  </tr>

</table>

[% END %]
```

This example generates the following output:

```
<table border="1">
  <tr>
    <td>
    Be not afeared; the isle is full of noises,
    Sounds and sweet airs, that give delight and hurt not.
    </td>
  </tr>
</table>
```

3.2.6.6 global

The global variable references a predefined hash array, which is initially empty. It can be used to store any global data that you want shared between templates, regardless of how they are processed, using PROCESS, INCLUDE, etc.

```
[% global.copyright = '&copy; 2003 Arthur Dent' %]
```

3.2.6.7 view, item

The Template Toolkit provides an experimental VIEW directive. It simplifies the process of displaying complex data structures by automatically mapping different data types onto templates designed specifically to deal with them.

In Example 3-11, a VIEW called people_view is defined that contains three BLOCK definitions, for hash, list, and text data items.

Example 3-11. view

```
[% VIEW people_view;
  BLOCK hash;
    "$item.name is from $item.home\n";
  END;

  BLOCK list;
    view.print(person)
      FOREACH person IN item;
  END;

  BLOCK text;
    item;
  END;
END;
-%]

[% people = [
  { name = 'Arthur Dent',
    home = 'Earth' }
```

```
  { name = 'Ford Prefect',
    home = 'Betelgeuse' }
    'Slartibartfast from Magrethea'
  ]
-%]

[% people_view.print(people) %]
```

The BLOCK definitions within the scope of the VIEW...END directives effectively remain local to the VIEW. Each can access the view and item variables that respectively reference the current view object, implemented by the Template::View module, and the current item of data being presented by the view.

The hash block, for example, will be called whenever the view has a hash array that needs presenting. The item variable references the hash array in question, allowing the block to access the item.name and item.home values.

```
BLOCK hash;
  "$item.name is from $item.home\n";
END;
```

The list block is called whenever the view has a list to present. In this case, we use a FOREACH directive to iterate through the items in the list that item now references. For each list element, we call back to the print method of the current view object so that it can correctly select the appropriate template for displaying it.

```
BLOCK list;
  view.print(person)
    FOREACH person IN item;
END;
```

The final block, text, is called whenever the view has a piece of plain text to present. All we need to do is output the value of item. If you want to pass all your text through a filter—to escape any HTML entities, for example—this is where you would do it.

```
BLOCK text;
  item;
END;
```

Having defined some sample data in people, we can then call the print method against the people_view view, passing the people data as an argument:

```
[% people_view.print(people) %]
```

The view will recognize that the argument is a reference to a list, and will call the list block to handle it. This will call the print method for each item in the list. For the first two items, this will result in the hash block being processed. For the last, it will call instead to the text block. The end result is that the right template gets called to handle the right kind of data.

Example 3-11, therefore, outputs the following:

```
Arthur Dent is from Earth
Ford Prefect is from Betelgeuse
Slartibartfast is from Magrethea
```

3.2.7 Variable Scope

Any simple variables that you create, or any changes you make to existing variables, will persist only while the template is being processed. The top-level variable hash is copied before processing begins, and any changes to variables are made in this copy, leaving the original intact. The same thing happens when you INCLUDE another template. The current namespace hash is cloned to prevent any variable changes made in the included template from interfering with existing variables. The PROCESS option bypasses the localization step altogether, making it slightly faster but requiring greater attention to the possibility of side effects caused by creating or changing any variables within the processed template.

Here is an example showing the difference between INCLUDE and PROCESS:

```
[% BLOCK change_name %]
   [% name = 'bar' %]
[% END %]

[% name = 'foo' %]
[% INCLUDE change_name %]
[% name %]              # foo
[% PROCESS change_name %]
[% name %]              # bar
```

Dotted compound variables behave slightly differently because the localization process is only skin-deep. The current variable namespace hash is copied, but no attempt is made to perform a deep-copy of other structures within it (hashes, arrays, objects, and so on). A variable referencing a hash, for example, will be copied to create a new reference, but one that points to the same hash. Thus, the general rule is that simple variables (undotted variables) are localized, but existing complex structures (dotted variables) are not.

This examples demonstrates this subtle effect:

```
[% BLOCK all_change %]
   [% x = 20 %]           # changes copy
   [% y.z = 'zulu' %]     # changes original
[% END %]

[% x = 10
   y = { z => 'zebra' }
%]
[% INCLUDE all_change %]
[% x %]             # still '10'
[% y.z %]           # now 'zulu'
```

If you create a complex structure such as a hash or list reference within a local template context, it will cease to exist when the template is finished processing:

```
[% BLOCK new_stuff %]
   [% # define a new 'y' hash array in local context
      y = { z => 'zulu' }
   %]
[% END %]

[% x = 10 %]
[% INCLUDE new_stuff %]
[% x %]             # outputs '10'
[% y %]             # outputs nothing, y is undefined
```

Similarly, if you update an element of a compound variable that *doesn't* already exist, a hash will be created automatically and deleted again at the end of the block:

```
[% BLOCK new_stuff %]

   [% y.z = 'zulu' %]

[% END %]
```

However, if the hash *does* already exist, you will modify the original with permanent effect. To avoid potential confusion, it is recommended that you don't update elements of complex variables from within blocks or templates included by another block or template.

If you want to create or update truly global variables, use the global namespace, described earlier.

3.2.8 Compile-Time Constant Folding

The default behavior for the Template Toolkit is to look up the value for a variable each and every time it is used in a template. This is what you want most of the time, but it can also be a little wasteful if you have variables that never or rarely change.

For example, you might want to define a set of variables to specify a particular color scheme for your web site. You want to use variables so that you can change the colors quickly and easily at some point in the future. However, you don't expect any of the values to change from one page, template, or web server request to the next. In fact, you would probably prefer it if they couldn't be changed, to protect them from being accidentally overwritten by a careless template author.

The solution is to use the CONSTANTS configuration option to provide a reference to a hash array of variables whose values are constant. The hash array can contain any kind of complex, nested, or dynamic data structures that you would normally define as a regular variable.

```
my $tt = Template->new({

    CONSTANTS => {

        version => 3.14,

        release => 'skyrocket',

        col     => {

            back => '#ffffff',

            fore => '#000000',

        },

        myobj => My::Object->new( ),

        mysub => sub { ... },

        joint => ', ',

    },

});
```

Within a template, these variables are accessed using the constants namespace prefix:

```
Version [% constants.version %] ([% constants.release %])



Background: [% constants.col.back %]
```

When the template is compiled, these variable references are replaced with the corresponding value. No further variable lookup is then performed when the template is processed. This results in templates that can be processed significantly faster by virtue of the fact that they have less work to do in looking up variable values. This can be an important optimization if you're using the Template Toolkit to generate dynamic pages behind an online web server.

Subroutines and objects can be provided as CONSTANTS items. You can even call virtual methods on constant variables:

```
[% constants.mysub(10, 20) %]

[% constants.myobj(30, 40) %]

[% constants.col.keys.sort.join(', ') %]
```

One important proviso is that any arguments you pass to subroutines or methods must also be literal values or compile-time constants.

For example, these are both fine:

```
# literal argument

[% constants.col.keys.sort.join(', ') %]



# constant argument

[% constants.col.keys.sort.join(constants.joint) %]
```

But this next example will raise an error at parse time, complaining that joint is a runtime variable that cannot be determined at compile time:

```
# ERROR: runtime variable argument!

[% constants.col.keys.sort.join(joint) %]
```

The CONSTANTS_NAMESPACE option can be used to provide a different namespace prefix for constant variables. For example:

```
my $tt = Template->new({

    CONSTANTS => {

        version => 3.14,

        # ...etc...

    },

    CONSTANTS_NAMESPACE => 'const',

});
```

Constants would then be referenced in templates as:

```
[% const.version %]
```

3.3 Virtual Methods

The Template Toolkit provides a number of virtual methods, or *vmethods*, that allow you to perform common operations on the three main types of data: scalars, lists, and hash arrays. In many cases, they are analogous to the Perl functions of the same name. The length scalar virtual method, for example, is implemented using Perl's length function.

Some virtual methods are interchangeable between data types. For example, you can call any list virtual method on a single scalar item and it will be treated as if it were a single element list. In other cases, the same virtual method is provided for different data types, providing alternate implementations of similar functionality. The size virtual method, for example, returns 1 for a scalar item, the number of elements in a list, or the number of key/value pairs in a hash array.

Virtual methods are invoked using the regular dot operator syntax:

```
[% string.length %]

[% list.join %]

[% hash.size %]
```

They can be chained together in compound variables, as shown here:

```
[% hash.keys.sort.join(', ') %]
```

The majority of virtual methods compute and return a value without modifying the underlying data (e.g., size). However, there are a number of virtual methods that do, one of which is pop, which removes the last item from a list. Example 3-12 shows examples of both in use.

Example 3-12. beer

```
[% beers   = [ 'Bass' 'Guinness' "Murphy's" ]
   bottles = 'bottles';

   WHILE (n = beers.size)
-%]
   [% n %] [% bottles %] of beer in my list,
   [% n %] [% bottles %] of beer,
   Take one down,
   Pass it around,
[%
   beer = beers.pop
   bottles = beers.max ? 'bottles' : 'bottle'
-%]
   (a bottle of [% beer %] is hastily drunk)
   [% beers.size or 'no' %] [% bottles %] of beer in my list.

[% END %]
```

Example 3-12 will output the following:

```
3 bottles of beer in my list,
3 bottles of beer,
Take one down,
```

Pass it around,

(a bottle of Murphy's is hastily drunk)

2 bottles of beer in my list.

2 bottles of beer in my list,

2 bottles of beer,

Take one down,

Pass it around,

(a bottle of Guinness is hastily drunk)

1 bottle of beer in my list.

1 bottle of beer in my list,

1 bottle of beer,

Take one down,

Pass it around,

(a bottle of Bass is hastily drunk)

no bottles of beer in my list.

3.3.1 Scalar Virtual Methods

The Template Toolkit defines the following virtual methods that operate on scalar values.

3.3.1.1 chunk(size)

This splits the input text into a list of smaller chunks. The argument defines the maximum length in characters of each chunk.

```
[% ccard_no = "1234567824683579";
   ccard_no.chunk(4).join
%]
```

It outputs the following:

```
1234 5678 2468 3579
```

If the size is specified as a negative number, the text will be chunked from right to left. This gives the correct grouping for numbers, for example:

```
[% number = 1234567;
   number.chunk(-3).join(',')
%]
```

and outputs the following:

```
1,234,567
```

3.3.1.2 defined

This returns true if the value is defined, even if it contains an empty string or the number zero. It returns false if the item is undefined.

```
foo [% foo.defined ? 'is' : 'is not' %] defined
```

3.3.1.3 hash

This returns a hash reference containing the original item as the single entry, indexed by the key value:

```
[% name = 'Slartibartfast' %]
[% user = name.hash %]
[% user.value %]        # Slartibartfast
```

3.3.1.4 length

This virtual method returns the number of characters in the string representation of the item:

```
[% IF password.length < 8 %]
  Your password is too short, please try again.
[% END %]
```

3.3.1.5 list

This returns the value as a single element list:

```
[% things = thing.list %]
```

The list virtual method can also be called against a list and will return the list itself, effectively doing nothing. Hence, if thing is already a list, thing.list will return the original list. Either way, things ends up containing a reference to a list.

Most of the time, you don't need to worry about the difference between scalars and lists. You can call a list virtual method against any scalar item and it will be treated as if it were a single element list. The FOREACH directive also works in a similar way. If you pass it a single scalar item instead of a reference to a list, it will behave as if you passed it a reference to a list containing that one item, and will iterate through the block just once.

The list vmethod is provided for those times when you really do want to be sure that you've got a list reference. For example, if you are calling a Perl subroutine that expects a reference to a list, adding the .list vmethod to the argument passed to it will ensure that it gets a list, even if the original argument is a scalar:

```
[% item = 'foo';
  mysub(item.list)  # same as mysub([item])
%]                  #  - item is a scalar

[% item = ['foo'];
  mysub(item.list)  # same as mysub(item)
%]                  #  - item is already a list
```

3.3.1.6 match(pattern)

The match virtual method performs a Perl regular expression match on the string using the pattern passed as an argument. Example 3-13 shows it being used to test whether the value of the serial variable matches the regular expression pattern ^\w{3}-\d{4}$. This pattern requires the string to be composed of exactly three alphanumeric "word" characters (\w{3}), followed by a dash (-), and then exactly four digits (\d{4}). The ^ and $ characters anchor the pattern to the start and end of the string, respectively. Without them, the pattern could match anywhere in what might be a much longer string. In this case, we want to make sure that the serial number is exactly eight characters long—no more, no less.

Example 3-13. serial

```
[% FOREACH serial IN ['ABC-1234', 'FOOD-4567', 'WXYZ-789'];
  IF serial.match('^\w{3}-\d{4}$');
    "GOOD serial number: $serial\n";
  ELSE;
    "BAD serial number: $serial\n";
  END;
END
%]
```

Example 3-13 outputs the following:

```
GOOD serial number: ABC-1234
BAD serial number: FOOD-4567
BAD serial number: WXYZ-789
```

The pattern can contain parentheses to capture parts of the matched string. If the entire pattern matches, the vmethod returns a reference to a list of the captured strings:

```
[% name = 'Arthur Dent' %]
[% matches = name.match('(\w+) (\w+)') %]
[% matches.1 %], [% matches.join('') %]  #  Dent, ArthurDent
```

In this example, the match vmethod returns a list of the two strings matched by the parenthesized patterns, (\w+). Here they are the values Arthur and Dent.

Remember that match returns false if the pattern does not match. It does *not* return a reference to an empty list, which both Perl and the Template Toolkit would treat as a true value, regardless of how many entries it contains. This allows you to test the value returned by match to determine whether the pattern matched.

The following example shows how the results of the match vmethod can be saved in the matches variable, while also testing that the pattern matched. The assignment statement is enclosed in parentheses and used as the expression for an IF directive.

```
[% IF (matches = name.match('(\w+) (\w+)')) %]
   pattern matches: [% matches.join(', ') %]
[% ELSE %]
   pattern does not match
[% END %]
```

Any regular expression modifiers can be embedded in the pattern using the (?imsx-imsx) syntax. For example, a case-insensitive match can be specified by using the (?i) construct at the start of the pattern:

```
[% matched = name.match('(?i)arthur dent') %]
```

In the following fragment, the (?x) flag is set to have whitespace and comments in the pattern ignored:

```
[% matched = name.match(
     '(?x)
        (\w+)   # match first name
         \s+    # some whitespace
        (\w+)   # match second name
     '
   )
%]
```

The details of Perl's regular expressions are described in the perlre(1) manpage. For a complete guide to learning and using regular expressions, see *Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly).

3.3.1.7 repeat(n)

This virtual method returns a string containing the original item repeated a number of times. The repeat value should be passed as an argument.

```
[% name = 'foo ' %]

[% name.repeat(3) %]              # foo foo foo
```

3.3.1.8 replace(search, replace)

This virtual method performs a global search and replace on the input string. The first argument provides a Perl regular expression to match part of the text. The second argument is the replacement value. Each occurrence of the pattern in the input string will be replaced (hence the "global" part of "global search and replace").

```
[% name = 'foo, bar & baz' %]

[% name.replace('\W+', '_') %]    # foo_bar_baz
```

The replace vmethod returns a copy of the string with the appropriate values replaced. The original string is not modified.

3.3.1.9 size

This virtual method always returns 1 for scalar values. It is provided for consistency with the hash and list virtual methods of the same name.

3.3.1.10 split(pattern)

This virtual method splits the input text into a list of strings that is then returned. It uses the regular expression passed as an argument as the delimiter, or whitespace as the default if an explicit delimiter is not provided.

```
[% path  = '/here:/there:/every/where';

   paths = path.split(':');

   paths.join;    # /here /there /every/where

%]
```

3.3.2 List Virtual Methods

The following virtual methods operate on a reference to a list and on scalar items that are treated as if they were single item lists. They can also be called against objects that are implemented as a blessed reference to a list. If the object defines a method—say, size—it will take precedence over the list virtual method of the same name. If the object does not define that method explicitly, the virtual method will instead be called.

```
[% mylistobj.size %]  # object method or list virtual method
```

3.3.2.1 first(n)

This virtual method returns the first item in the list without removing it from the list:

```
[% list  = [10, 20 30] %]

[% list.first      %]   # 10

[% list.join(', ') %]   # 10, 20, 30
```

A number can be provided as an argument. In this case, the vmethod returns a reference to a list containing that many items copied from the start of the list:

```
[% list.first(2).join(', ') %]   # 10, 20
```

3.3.2.2 grep(pattern)

The grep vmethod returns a list of the items in the list that match the regular expression pattern passed as an argument. For example, you can use it to select all the files in a directory listing, files, that have a *.txt* ending:

```
[% txtfiles = files.grep('\.txt$') %]
```

3.3.2.3 join(delimiter)

This virtual method returns the items in the list joined into a single string. By default it uses a single space to join the items.

```
[% list  = [10, 20 30] %]

[% list.join %]        # 10 20 30
```

An alternate delimiter can be provided as an argument:

```
[% list.join(', ') %]  # 10, 20, 30
```

3.3.2.4 last(n)

The last virtual method returns the last item in the list without removing it from the list:

```
[% list  = [10, 20 30] %]

[% list.last      %]  # 30

[% list.join(', ') %]  # 10, 20, 30
```

As with first, an argument can be provided indicating the number of items that should be returned from the end of the list:

```
[% list.last(2).join(', ') %]   # 20, 30
```

3.3.2.5 max

The max virtual method returns the index number for the last element in the list. It is always one less than the value returned by the size virtual method.

```
[% list  = [10, 20 30] %]

[% list.max %]  # 2
```

3.3.2.6 merge(list)

The merge virtual method returns a list composed of the original items in the list plus those from any additional lists passed as arguments:

```
[% list_a = [ 1 2 3 ];

   list_b = [ 4 5 6 ];

   list_c = [ 7 8 9 ];

   list_d = list_a.merge(list_b, list_c);

%]
```

The new list, list_d, contains the items merged from list_a, list_b, and list_c. The original lists are left unmodified.

```
[% list_a.join(', ') %]  # 1, 2, 3

[% list_b.join(', ') %]  # 4, 5, 6

[% list_c.join(', ') %]  # 7, 8, 9

[% list_d.join(', ') %]  # 1, 2, 3, 4, 5, 6, 7, 8, 9
```

3.3.2.7 pop

This virtual method removes the last item from the list and returns it:

```
[% list = [10, 20 30] %]

[% list.pop  %]  # 30
```

3.3.2.8 reverse

The reverse virtual method returns a reference to a new list containing the items in the original list, but in reverse order:

```
[% list  = [10, 20 30] %]

[% list.reverse.join(', ') %]   # 30, 20, 10
```

3.3.2.9 shift

This vmethod removes the first item from the list and returns it:

```
[% list = [10, 20 30] %]

[% list.shift %]  # 10
```

3.3.2.10 size

This virtual method returns the number of elements in the list:

```
[% list = [10, 20 30] %]

[% list.size %]  # 3
```

3.3.2.11 slice(from, to)

This virtual method returns the items in the list between the bounds passed as arguments. If the second argument is not specified, it defaults to the last item in the list. The original list is not modified.

```
[% list = [10, 20 30] %]

[% list.slice(0, 1).join(', ') %]   # 10, 20

[% list.join(', ') %]               # 10, 20, 30
```

The arguments can also be negative numbers, in which case they are counted from the end of the list:

```
[% list.slice(-2, -1).join(', ') %] # 20, 30
```

3.3.2.12 sort, nsort

The sort vmethod returns a list of the items in alphabetical order:

```
[% list = ['foo', 'bar', 'baz'] %]

[% list.sort.join(', ') %]  # bar baz foo
```

The nsort vmethod is similar, but sorts the items in numerical order. The following example illustrates the difference between the two:

```
[% list = ['0.1', '1', '02', '3', '010', '11'] %]

[% list.sort.join(', ')  %]  # 0.1, 010, 02, 1, 11, 3

[% list.nsort.join(', ') %]  # 0.1, 1, 02, 3, 010, 11
```

When the items in the list are references to hash arrays, an optional argument can be used to specify a sort key. This corresponds to an entry in each hash array, the value of which is used to sort the items. This is shown in Example 3-14, where the id and name keys as specified as arguments to the sort virtual method.

Example 3-14. products

```
[% products = [
     { id = 'xyz789', name = 'Foo Widget' }
     { id = 'def456', name = 'Bar Widget' }
     { id = 'abc123', name = 'Baz Widget' }
   ]
-%]
Products sorted by id:
[% FOREACH product IN products.sort('id') -%]
  * [% product.id %] [% product.name %]
[% END -%]

Products sorted by name:
[% FOREACH product IN products.sort('name') -%]
  * [% product.id %] [% product.name %]
[% END -%]
```

The output generated by Example 3-14 is as follows:

```
Products sorted by id:
  * abc123 Baz Widget
  * def456 Bar Widget
  * xyz789 Foo Widget

Products sorted by name:
  * def456 Bar Widget
  * abc123 Baz Widget
  * xyz789 Foo Widget
```

3.3.2.13 splice(offset, length, list)

This virtual method behaves just like Perl's splice function, allowing you to selectively remove or replace part of a list. The first argument defines the offset in the list of the part to be removed, starting at 0 for the first item. With just one argument provided, the vmethod removes everything from that element onward, returning the removed items in a new list.

```
[% primes = [2, 3, 5, 7, 11, 13];
  others = primes.splice(2);
  primes.join(', ');      # 2, 3
  others.join(', ');      # 5, 7, 11, 13
%]
```

The offset can also be specified as a negative number, in which case it is counted backward from the end of the list:

```
[% primes = [2, 3, 5, 7, 11, 13];
   others = primes.splice(-2);
   primes.join(', ');      # 2, 3, 5, 7
   others.join(', ');      # 11, 13
%]
```

A second optional argument can be provided to specify the length of the section to be removed:

```
[% primes = [2, 3, 5, 7, 11, 13];
   others = primes.splice(2, 3);
   primes.join(', ');      # 2, 3, 13
   others.join(', ');      # 5, 7, 11
%]
```

A third optional argument can be used to provide a list of items that will be inserted into the list in place of the removed section. This can be specified as a reference to a list or as a list of items.

```
[% primes1 = [2, 3, 5, 7, 11];
   primes2 = [13, 17, 19];

   # pass reference to list
   primes3 = primes1.splice(1, 2, primes2);
   primes1.join(', ');     # 2, 13, 17, 19, 7, 11
   primes2.join(', ');     # 13, 17, 19
   primes3.join(', ');     # 3, 5

   # pass list of items
   primes4 = primes1.splice(1, 3, 3, 5);
   primes1.join(', ');     # 2, 3, 5, 7, 11
   primes4.join(', ');     # 13, 17, 19
%]
```

3.3.2.14 unique

This vmethod returns a copy of the list with any duplicate values removed:

```
[% mylist = [ 1 2 3 2 3 4 1 4 3 4 5 ];
   numbers = mylist.unique;
   numbers.join(', ');    # 1, 2, 3, 4, 5
%]
```

3.3.2.15 unshift(item)

This virtual method adds an item to the start of a list:

```
[% numbers = [ 2.718, 3.142 ];
   numbers.unshift(1.414);
   numbers.join(', ');    # 1.414, 2.718, 3.142
%]
```

3.3.2.16 push(item)

The push vmethod is similar to unshift, but adds the item to the end of the list:

```
[% numbers = [ 1.414, 2.718 ];
   numbers.push(3.142);
   numbers.join(', ');    # 1.414, 2.718, 3.142
%]
```

3.3.3 Hash Virtual Methods

The following virtual methods operate on hash references. They can also be called against objects that are implemented as blessed hash arrays. As with list virtual methods, any method explicitly provided by the object will take precedence over a hash virtual method of the same name.

```
[% myhashobj.keys %]  # object method or hash virtual method
```

3.3.3.1 defined(key)

The defined virtual method returns true or false to indicate whether a particular item is defined in the hash. A key for the item in question should be passed as an argument:

```
foo [% hash.defined('foo') ? 'is' : 'is not' %] defined
```

3.3.3.2 each

The each virtual method, as shown in Example 3-15, returns a list of the keys and values in the hash, interleaved as key1, value1, key2, value2, etc.

Example 3-15. each

```
[% product = {
     id    = 'ABC-123'
     name  = 'ABC Widget #123',
     price = 7.99,
   }

   keyvals = product.each;

   WHILE (keyvals.size);
     key = keyvals.shift;
     val = keyvals.shift;
```

```
    "$key => $val\n";
  END
%]
```

Example 3-15 outputs the following:

```
id => ABC-123
price => 7.99
name => ABC Widget #123
```

Hash arrays do not maintain any particular order for the items in them, so the each virtual method (and also keys and values, covered later in this section) returns the items in what appears to be a random order.[2] This ensures that key and value return their items in a corresponding order, even if we're not sure what that order will be.

[2] Strictly speaking, it is nondeterministic rather than truly random, although Perl does, of course, have an idea how to determine the "correct" traversal order for a hash array.

3.3.3.3 exists(key)

The exists virtual method performs a similar function to defined, but indicates whether the item exists in the hash. If it does exist, the exists vmethod will return true, even if it is set to an undefined value. In contrast, the defined vmethod returns false if an item exists in the hash but is set to an undefined value.

```
foo [% hash.exists('foo') ? 'does' : 'does not' %] exist
```

3.3.3.4 import(hash)

The import virtual method can be called against a hash array to have it import the elements of another hash array:

```
[% hash1 = {
      foo = 'Foo'
      bar = 'Bar'
  }
  hash2 = {
      wiz = 'Wiz'
      woz = 'Woz'
  }
%]

[% hash1.import(hash2) %]
[% hash1.wiz %]                    # Wiz
```

You can also call the import vmethod by itself to import the items in a hash array into the current variable namespace. In effect, the items in the hash array become new template variables.

```
[% user = { id = 'dent' name = 'Arthur Dent' } %]
[% import(user) %]
[% id %]: [% name %] # dent: Arthur Dent
```

3.3.3.5 item(key)

This vmethod performs a simple lookup in the hash, returning the value for the key passed as an argument:

```
[% hash.item('foo') %]
```

This has the same effect as retrieving an item directly:

```
[% hash.foo %]
```

The item virtual method can be used to fetch an item from the hash that might otherwise be confused for a hash virtual method. In the following example, the size item is fetched from the font hash using the item virtual method:

```
[% size = font.item('size') %]  # hash item
```

If the font hash does not contain a size key, it will return an undefined value. If instead we access it directly using the dot operator, the size virtual method will automatically be called if the hash does not contain a defined value for size.

```
[% size = font.size %]          # hash item or vmethod
```

In this case, we would end up with a value defined for size, even if the hash doesn't contain a size item.

3.3.3.6 keys

This virtual method performs the same task as the equivalent Perl function. It returns a reference to a list containing the keys of the hash. As with each, these are returned in no particular order, although it is guaranteed to be the same order as the corresponding values returned by the values vmethod.

```
[% product = {
     id    = 'widget2k'
     name  = "Widget 2000"
     about = "Ultra-fast dynamic widget"
     price = 4.99
  }
%]
[% FOREACH key = product.keys -%]
  [% key %] => [% product.$key %]
[% END %]
```

This generates the following output:

```
about => Ultra-fast dynamic widget
id => widget2k
price => 4.99
name => Widget 2000
```

3.3.3.7 list

The list virtual method returns the contents of the hash as a reference to a list. An argument can be passed to indicate the desired items required in the list: keys to return a list of the keys (same as hash.keys), values to return a list of the values (same as hash.values), or each to return as list of key/value pairs (same as hash.each). When called without an argument, it returns a list of hash references, each of which contains a key and value item representing a single key/value pair in the hash.

Consider the following hash:

```
[% hash = {
      one   = 1
      two   = 2
      three = 3
  };

%]
```

Calling hash.list('keys'):

```
[% FOREACH key IN hash.list('keys') -%]
   [% key %]
[% END %]
```

generates this output:

```
one
three
two
```

Calling hash.list('values'):

```
[% FOREACH key IN hash.list('values') -%]
   [% key %]
[% END %]
```

generates this output:

```
1
3
2
```

Calling hash.list('each'):

```
[% FOREACH key IN hash.list('each') -%]
   [% key %]
[% END %]
```

generates this output:

```
one
1
three
3
two
2
```

Calling hash.list:

```
[% FOREACH keyval IN hash.list -%]
   [% keyval.key %] => [% keyval.value %]
[% END %]
```

generates this output:

```
one => 1
three => 3
two => 2
```

3.3.3.8 size

This virtual method returns the number of key/value pairs in the hash.

3.3.3.9 sort, nsort

The sort virtual method returns a list of the keys sorted alphabetically:

```
[% FOREACH term IN terms.sort %]
  [% term %] means '[% terms.$term %]',
[% END %]
```

The nsort vmethod performs a similar function but returns the keys sorted by their numerical value. See the sort and nsort list virtual methods for an example.

3.3.3.10 values

The values virtual method returns a list of the values in a hash array. They are returned in the same apparently random order as for each and keys.

```
[% keys = product.keys;
  vals = product.vals;

  WHILE keys.size;
    key = keys.shift;
    val = vals.shift;
    "$key => $val\n";
  END
%]
```

3.3.4 Defining New Virtual Methods

You can define your own virtual methods for scalars, lists, and hash arrays. You might do this to add useful functionality not provided by the Template Toolkit itself, or to add methods specific to your data. For example, if you want to offer template designers a way to format a number as a dollar-and-cents string, you might do this with a new virtual method on numbers.

To add a new virtual method from Perl, manipulate package variables yourself to add the new method to the stash:

```
# load Template::Stash to make method tables visible
use Template::Stash;

# define list method to return a new list of palindromic strings only
$Template::Stash::LIST_OPS->{ palindromes } = sub {
    my $list = shift;
    return [ grep { $_ eq reverse($_) } @$list ];
};
```

Alternatively, use the define_vmethod( ) method on the Template Toolkit's context:

```
# locate the context
use Template;
my $template = Template->new( );
my $context  = $tt->context( );

# define list method to return a new list of palindromic strings only
```

```
$context->define_vmethod('list', 'palindromes', sub {
    my $list = shift;
    return [ grep { $_ eq reverse($_) } @$list ];
};
```

3.3.4.1 Stash package variables

The Template::Stash package variables $SCALAR_OPS, $LIST_OPS, and $HASH_OPS are references to hash arrays that define these virtual methods. The HASH_OPS and LIST_OPS virtual methods are implemented as subroutines that accept a hash or list reference as the first item, respectively. The SCALAR_OPS virtual methods are subroutines that accept a scalar value as the first item.

Any other arguments specified when the method is called will also be passed to the subroutine. Any named arguments will be collated into a single hash reference and passed as the last argument, as for any subroutine or method call. This example, therefore:

```
# load Template::Stash to make method tables visible
use Template::Stash;


# define list method to return new list of odd numbers only
$Template::Stash::LIST_OPS->{ odd } = sub {
    my $list = shift;
    return [ grep { $_ % 2 } @$list ];
};
```

creates this template:

```
[% primes = [ 2, 3, 5, 7, 9 ] %]
[% primes.odd.join(', ') %]        # 3, 5, 7, 9
```

New virtual methods can perform arbitrarily complex actions, or very simple actions:

```
$Template::Stash::SCALAR_OPS->{ int } = sub { int($_[0]) };


use Digest::MD5 qw(md5_hex);
$Template::Stash::SCALAR_OPS->{ md5 } = sub { md5_hex($_[0]) };
```

Here is a vmethod to pick an element randomly from a list (courtesy of Slash):

```
$Template::Stash::LIST_OPS->{ rand } = sub {
    my $list = shift;
    return $list->[rand @$list];
};
```

Implementing delete for hashes is straightforward:

```
$Template::Stash::HASH_OPS->{ delete } = sub {
    my ($hash, $key) = @_;
    delete $hash->{ $key } if (defined $key);
}
```

It can be used as you would expect:

```
[% hash.delete('key') %]
```

delete returns the deleted element, just like Perl's delete. This can be chained with other vmethods:

```
[% hash.delete('ccard_no').md5 %]
```

3.3.4.2 Stash and context methods

The Template::Stash and Template::Context modules both implement define_vmethod( ) methods that handle the installation of new virtual methods into the stash package variables. In the case of Template::Context, it simply delegates the task to the current Template::Stash object in use.

The internal architecture of the Template Toolkit is described in painful detail in Chapter 7, but you don't need to know too much about it to be able to define your own virtual methods. The Template object implements a context( ) method that returns the current Template::Context object (the internal template processing engine) that it is using:

```
my $template = Template->new( );

my $context  = $tt->context( );
```

The define_vmethod( ) method can then be called against the $context object. The first argument denotes the data type and should be one of the values scalar, list, or hash. For convenience, item is provided as an alias for scalar, and array as an alias for list. The second argument is the name of the virtual method. The third argument is a reference to the subroutine implementing it.

Here is an example showing another way of adding the odd list virtual method:

```
$context->define_vmethod('list', 'odd', sub {

    my $list = shift;

    return [ grep { $_ % 2 } @$list ];

};
```

This example shows a hash virtual method being added to print a Perl representation of the hash array in sorted order. Here we are using Perl's => operator, which acts just like a comma but saves us from having to quote the hash and dump values.

```
$context->define_vmethod(hash => dump => sub {

    my $hash = shift;

    return '{ '

        . join( ', ',

                map { "$_ => '$hash->{$_}'" }

                sort keys %$hash )

        . ' }';

});
```

If you enable the EVAL_PERL configuration option, you can also define virtual methods in a PERL block from within a template. The $context variable is automatically available for use in PERL blocks.

```
[% PERL %]

$context->define_vmethod(hash => dump => sub {

    my $hash = shift;

    return '{ '

        . join( ', ',

                map { "$_ => '$hash->{$_}'" }

                sort keys %$hash )

        . ' }';

});

[% END %]
```

It is also possible to write a plugin that defines virtual methods. This is covered in Chapter 8.

Chapter 4. Template Directives

Templates consist of a combination of fixed text and template directives. The template directives are recognized by the Template Toolkit and are expanded in a processor's output. In this chapter, we will take a close look at all of the directives that the Template Toolkit provides. We've already seen examples of many of them in previous chapters, but now we'll go back and fill in all of the details.

The Template Toolkit has directives for common presentation tasks. There are directives for accessing and setting variables, loading and using both external and local templates, repetition, conditional processing, flow control, and exception handling. Directives are also provided to define macros and access template metadata. If that's not enough for you, you can extend the functionality of the Template Toolkit using filters, plugins, or even inline Perl code.

4.1 Accessing Variables

The Template Toolkit allows you to define variables in your templates. In this section, we will look at the various directives that the Template Toolkit provides for manipulating template variables.

4.1.1 GET

The GET directive retrieves and outputs the value of the named variable:

```
[% GET foo %]
```

The GET keyword is optional. A variable can be specified in a directive tag by itself:

```
[% foo %]
```

The variable name can have an unlimited number of elements, each separated by a . (dot). Each element can have arguments specified within parentheses:

```
[% foo %]

[% bar.baz %]

[% biz.baz(10) %]
```

See Chapter 3 for a full discussion of template variables.

The GET directive can also take an expression as an argument:

```
[% GET total + tax %]


[% GET length * breadth * height %]
```

Expressions can use any of the mathematical operators +, -, *, /, mod, div, and %. They can be combined using the logical operators and, or, and not. &&, ||, and ! are provided as aliases for and, or, and not.

```
[% GET golgafrincham.answer or 42 %]
```

The mod, div, and % operators carry out integer division. div returns the result of the division and mod returns the modulus (or remainder) from the division:

```
[% SET people = 4

   pies = 10 %]


[% pies %] pies shared between [% people %] people

is [% pies div people %] pies each

(and [% pies mod people %] pies left over)
```

The % operator is a synonym for mod.

The logical operator ?: is also available:

```
[% pies > people * 2 ? 'everyone happy' : 'not enough pies' %]
```

This operator works by evaluating the expression that comes before the question mark to see if it is true or false. If it is true, the operator returns the expression that comes before the : character. If it is false, the operator returns the expression that follows the : character. In the example, status is set to either everyone happy or not enough pies depending on whether we have at least two pies for everyone.

The comparison operators = =, !=, <, <=, >, and >= are also provided. Note that they always compare their operands as strings.

```
[% GET name = = 'Zaphod' ?

        'Greetings Mr. President' :

        'Hello Monkey' %]
```

4.1.2 SET

The SET directive allows you to assign new values to existing variables or to create new temporary variables:

```
[% SET title = 'Hello World' %]
```

The SET keyword is optional when it is unambiguous:

```
[% title = 'Hello World' %]
```

Variables may be assigned the values of other variables, unquoted numbers (digits), literal text (single quotes), or quoted text (double quotes). In the latter case, any variable references within the text will be interpolated when the string is evaluated. Variables should be prefixed by $, using curly braces to explicitly scope the variable name where necessary.

```
[% foo  = 'Foo'  %]             # literal value 'Foo'
[% bar  =  foo   %]             # value of variable 'foo'
[% cost = '$100' %]             # literal value '$100'
[% item = "$bar: ${cost}.00" %]   # value "Foo: $100.00"
```

Multiple variables may be assigned in the same directive and are evaluated in the order specified. Thus, the previous example could have been written:

```
[% foo  = 'Foo'
   bar  = foo
   cost = '$100'
   item = "$bar: ${cost}.00"
%]
```

Simple expressions can also be used, as they can with GET:

```
[% ten    = 10
   twenty = 20
   thirty = twenty + ten
   forty  = 2 * twenty
   fifty  = 100 div 2
   six    = twenty mod 7
%]
```

You can concatenate strings together using the underscore (_) operator. In Perl 5, the . is used for string concatenation, but in Perl 6, as in the Template Toolkit, the . will be used as the method-calling operators and the underscore (_) operator will be used for string concatenation.[1] Note that the operator must be specified with surrounding whitespace that, as Larry says, is construed as a feature:

> [1] Larry has since changed his mind and it looks as if the ~ will be the Perl 6 string concat operator. As always, this is all subject to change.

```
[% copyright = '(C) Copyright ' _ year _ ' ' _ author %]
```

You can, of course, achieve a similar effect with double-quoted string interpolation:

```
[% copyright = "(C) Copyright $year $author" %]
```

The SET directive can also take arguments that are expressions in exactly the same way as the GET directive:

```
[% total = price + (price * tax_rate) %]
```

4.1.3 CALL

The CALL directive is similar to GET in evaluating the variable named, but doesn't print the result returned. This can be useful when a variable is bound to a subroutine or object method that you want to call but whose returned value you aren't interested in.

```
[% CALL dbi.disconnect %]


[% CALL inc_page_counter(page_count) %]
```

4.1.4 DEFAULT

The DEFAULT directive is similar to SET but updates only variables that are currently undefined or have no "true" value (in the Perl sense):

```
[% DEFAULT
   name = 'John Doe'
   id   = 'jdoe'
%]
```

This can be particularly useful in common template components to ensure that some sensible default is provided for otherwise undefined variables. If a true value is provided for variables with DEFAULT values, the provided value will be used; otherwise, the default value will be used.

```
[% DEFAULT
   title = 'Hello World'
   bgcol = '#ffffff'
%]
<html>
<head>
<title>[% title %]</title>
</head>

<body bgcolor="[% bgcol %]">
```

DEFAULT can also take an expression as an argument in exactly the same way as GET:

```
[% DEFAULT pies = 3 * people %]
```

DEFAULT has no effect on variables that already have values.

4.2 Accessing External Templates and Files

Variables are for storing little bits of data. Templates are for writing larger chunks of content. As with variables, it is often useful to be able to reuse the contents of a template. For example, the output of a template will often actually be composed of the output of a number of lower-level templates. These lower-level templates can be reused in other templates. This is very similar to the modular approach to writing programs that encourages code reuse.

The Template Toolkit provides a number of directives for manipulating templates. The first three of these all work in a very similar way. INSERT, PROCESS, and INCLUDE all insert the contents of another named template into the current template. The basic syntax for these directives looks like this:

```
[% INCLUDE filename %]
```

You may optionally include arguments (in a name = value format) that define variables to use while processing the included template:

```
[% INCLUDE filename title = "la la la"

                moonphase = "waxing" %]
```

With all of these directives, the results of processing the template are included in the output in place of the directive. The WRAPPER directive works a little differently. It is a block directive and it allows you to define a template that is wrapped around the block of content. The content of the block is made available to the *wrapper* template in a special variable called content.

```
[% WRAPPER layout %]

  blah blah

[% END %]
```

We discuss the directives for manipulating templates in the next four sections.

4.2.1 INSERT

The INSERT directive is used to insert the contents of an external file at the current position:

```
[% INSERT myfile %]
```

No attempt to parse or process the file is made. The contents, possibly including any embedded template directives, are inserted intact.

The filename specified should be relative to one of the INCLUDE_PATH directories. Absolute (i.e., starting with /) and relative (i.e., starting with .) filenames may be used if the ABSOLUTE and RELATIVE options are set, respectively. Both of these options are disabled by default.

```
my $tt = Template->new({

    INCLUDE_PATH => '/here:/there:/every/where',

});



$tt->process('myfile');
```

The contents of *myfile* are:

```
[% INSERT foo %]            # looks for /here/foo then /there/foo

[% INSERT /etc/passwd %]    # file error: ABSOLUTE not set

[% INSERT ../secret %]      # file error: RELATIVE not set
```

For convenience, the filename does not need to be quoted as long as it contains only alphanumeric characters, underscores, dots, or forward slashes. Names containing any other characters should be quoted.

```
[% INSERT misc/legalese.txt            %]

[% INSERT 'dos98/Program Files/foobar' %]
```

To evaluate a variable to specify a filename, you should explicitly prefix it with a $ or use double-quoted string interpolation:

```
[% language = 'en'
   legalese = 'misc/legalese.txt'
%]

[% INSERT $legalese %]              # 'misc/legalese.txt'
[% INSERT "$language/$legalese" %]  # 'en/misc/legalese.txt'
```

Multiple files can be specified using + as a delimiter. All files should be unquoted names or quoted strings. Any variables should be interpolated into double-quoted strings.

```
[% INSERT legalese.txt + warning.txt %]
[% INSERT  "$legalese" + warning.txt %]  # requires quoting
```

4.2.2 INCLUDE

The INCLUDE directive is used to process and include the output of another template file or block:

```
[% INCLUDE header %]
```

If a BLOCK of the specified name is defined in the same file or in a file from which the current template has been called (i.e., a parent template), it will be used in preference to any file of the same name.

```
[% INCLUDE table %]             # uses BLOCK defined below

[% BLOCK table %]
   <table>
   ...
   </table>
[% END %]
```

If a BLOCK definition is not currently visible, the template name should be a file relative to one of the INCLUDE_PATH directories, or an absolute or relative filename if the ABSOLUTE / RELATIVE options are appropriately enabled. The INCLUDE directive automatically quotes the filename specified, as per INSERT described earlier. When a variable contains the name of the template for the INCLUDE directive, it should be explicitly prefixed by $ or double-quoted:

```
[% myheader = 'my/misc/header' %]
[% INCLUDE   myheader  %]            # 'myheader'
[% INCLUDE  "myheader" %]            # 'myheader'
[% INCLUDE  $myheader  %]            # 'my/misc/header'
[% INCLUDE "$myheader" %]            # 'my/misc/header'
```

Any template directives embedded within the file will be processed accordingly. All variables currently defined will be visible and accessible from within the included template.

```
[% title = 'Hello World' %]
[% INCLUDE header %]
<body>
...
```

Therefore, this *header* template:

```
<html>
<title>[% title %]</title>
```

provides the following output:

```
<html>

<title>Hello World</title>

<body>

...
```

Local variable definitions may be specified after the template name, temporarily masking any existing variables. Insignificant whitespace is ignored within directives, so you can add variable definitions on the same line, on the next line, or split across several lines with comments interspersed, if you prefer.

```
[% INCLUDE table %]



[% INCLUDE table title="Active Projects" %]



[% INCLUDE table

     title   = "Active Projects"

     bgcolor = "#80ff00"    # chartreuse

     border  = 2

%]
```

The INCLUDE directive localizes (i.e., copies) all variables before processing the template. Any changes made within the included template will not affect variables in the including template.

```
[% foo = 10 %]



foo is originally [% foo %]

[% INCLUDE bar %]

foo is still [% foo %]



[% BLOCK bar %]

   foo was [% foo %]

   [% foo = 20 %]

   foo is now [% foo %]

[% END %]
```

The preceding example produces the following output:

```
foo is originally 10

   foo was 10

   foo is now 20

foo is still 10
```

> The localization of the stash (that is, the process by which variables are copied before an INCLUDE to prevent being overwritten) is only skin-deep. The top-level variable namespace (hash) is copied, but no attempt is made to perform a deep-copy of other structures (hashes, arrays, objects, etc.). Therefore, a foo variable referencing a hash will be copied to create a new foo variable that points to the same hash array. Thus, if you update compound variables (e.g., foo.bar), you will change the original copy, regardless of any stash localization. If you're not worried about preserving variable values, or you trust the templates you're including, you might prefer to use the PROCESS directive, which is faster by virtue of not performing any localization.

You can specify dotted variables as "local" variables to an INCLUDE directive. However, be aware that because of the localization issues explained earlier (if you skipped the previous Note, you might want to go back and read it, or else skip this section too), the variables might not actually be "local." If the first element of the variable name already references a hash array, the variable update will affect the original variable.

```
[% foo = {

       bar = 'Baz'

   }

%]



[% INCLUDE somefile foo.bar='Boz' %]



[% foo.bar %]           # Boz
```

This behavior can be a little unpredictable (and may well be improved upon in a future version). If you know what you're doing with it and you're sure that the variables in question are defined (nor not) as you expect them to be, you can rely on this feature to implement some powerful "global" data-sharing techniques. Otherwise, you might prefer to steer clear and always pass simple (undotted) variables as parameters to INCLUDE and other similar directives.

If you want to process several templates simultaneously, you can specify each of their names (quoted or unquoted names only, no unquoted $variables) joined together by +. The INCLUDE directive will then process them in order.

```
[% INCLUDE html/header + "site/$header" + site/menu

     title = "My Groovy Web Site"

%]
```

The variable stash is localized once and then the templates specified are processed in order, all within that same variable context. This makes it slightly faster than specifying several separate INCLUDE directives (because you clone the variable stash only once instead of *n* times), but it's not quite as "safe" because any variable changes in the first file will be visible in the second, third, and so on. This might be what you want, of course, but then again, it might not.

4.2.3 PROCESS

The PROCESS directive is similar to INCLUDE but does not perform any localization of variables before processing the template. Any changes made to variables within the included template will be visible in the including template. For example, this code:

```
[% foo = 10 %]



foo is [% foo %]

[% PROCESS bar %]

foo is [% foo %]



[% BLOCK bar %]

   [% foo = 20 %]

   changed foo to [% foo %]

[% END %]
```

produces this output:

```
foo is 10

   changed foo to 20

foo is 20
```

Parameters may be specified in the PROCESS directive, but these too will become visible changes to current variable values. As such, the following code:

```
[% foo = 10 %]

foo is [% foo %]

[% PROCESS bar

   foo = 20

%]

foo is [% foo %]



[% BLOCK bar %]

   this is bar, foo is [% foo %]

[% END %]
```

produces the following output:

```
foo is 10

   this is bar, foo is 20

foo is 20
```

The PROCESS directive is slightly faster than the INCLUDE directive because it avoids the need to localize (i.e., copy) the variable stash before processing the template. As with INSERT and INCLUDE, the first parameter does not need to be quoted as long as it contains only alphanumeric characters, underscores, periods, or forward slashes. A $ prefix can be used to explicitly indicate a variable that should be interpolated to provide the template name:

```
[% myheader = 'my/misc/header' %]

[% PROCESS  myheader %]            # 'myheader'

[% PROCESS $myheader %]            # 'my/misc/header'
```

As with INCLUDE, multiple templates can be specified, delimited by +, and are processed in order:

```
[% PROCESS html/header + my/header %]
```

4.2.4 WRAPPER

It's not unusual to find yourself adding common headers and footers to pages or sub-sections within a page. For example:

```
[% INCLUDE section/header

   title = 'Quantum Mechanics'

%]

   Quantum mechanics is a very interesting subject which

   should prove easy for the layman to fully comprehend.

[% PROCESS section/footer %]



[% INCLUDE section/header

   title = 'Desktop Nuclear Fusion for Under $50'

%]

   This describes a simple device that generates significant

   sustainable electrical power from common tap water via the process
```

```
  of nuclear fusion.

[% PROCESS section/footer %]
```

The individual template components being included might look like the folowing examples:

section/header:

```
<p>

<h2>[% title %]</h2>
```

section/footer:

```
</p>
```

The WRAPPER directive provides a way of simplifying this a little. It encloses a block to a matching END directive, which is first processed to generate some output. This is then passed to the named template file or BLOCK as the content variable.

```
[% WRAPPER section

  title = 'Quantum Mechanics'

%]

  Quantum mechanics is a very interesting subject which

  should prove easy for the layman to fully comprehend.

[% END %]



[% WRAPPER section

  title = 'Desktop Nuclear Fusion for Under $50'

%]

  This describes a simple device that generates significant

  sustainable electrical power from common tap water via the process

  of nuclear fusion.

[% END %]
```

The single *section* template can then be defined as:

```
<p>

<h2>[% title %]</h2>

[% content %]

</p>
```

Like other block directives, it can be used in side-effect notation:

```
[% INSERT legalese.txt WRAPPER big_bold_table %]
```

It's also possible to specify multiple templates to a WRAPPER directive. The specification order indicates outermost to innermost wrapper templates. For example, given the following template block definitions:

```
[% BLOCK bold   %]<b>[% content %]</b>[% END %]

[% BLOCK italic %]<i>[% content %]</i>[% END %]
```

the directive:

```
[% WRAPPER bold + italic %]Hello World[% END %]
```

would generate the following output:

```
<b><i>Hello World</i></b>
```

4.3 Defining Local Template Blocks

Sometimes, particularly in a project that involves a large number of small templates, it doesn't seem very efficient to create an external file for every template that you need. The BLOCK ... END construct can be used to avoid this. It allows you to define template component blocks that can be processed with the INCLUDE, PROCESS, and WRAPPER directives.

```
[% BLOCK tabrow %]

<tr><td>[% name %]<td><td>[% email %]</td></tr>

[% END %]



<table>

[% PROCESS tabrow  name='Fred'  email='fred@nowhere.com' %]

[% PROCESS tabrow  name='Alan'  email='alan@nowhere.com' %]

</table>
```

A BLOCK definition can be used before it is defined, as long as the definition resides in the same file. The block definition itself does not generate any output.

```
[% PROCESS tmpblk %]



[% BLOCK tmpblk %] This is OK [% END %]
```

You can use an anonymous BLOCK to capture the output of a template fragment:

```
[% julius = BLOCK %]

   And Caesar's spirit, ranging for revenge,

   With Ate by his side come hot from hell,

   Shall in these confines with a monarch's voice

   Cry  'Havoc', and let slip the dogs of war;

   That this foul deed shall smell above the earth

   With carrion men, groaning for burial.

[% END %]
```

Like a named block, an anonymous block can contain any other template directives that are processed when the block is defined. The output generated by the block is then assigned to the variable julius.

Anonymous BLOCKs can also be used to define block macros. The enclosing block is processed each time the macro is called.

```
[% MACRO locate BLOCK %]

   The [% animal %] sat on the [% place %].

[% END %]



[% locate(animal='cat', place='mat') %]    # The cat sat on the mat

[% locate(animal='dog', place='log') %]    # The dog sat on the log
```

4.4 Loops

It is very common to want to repeat parts of a template. You might want to produce similar output for every item in a list, or you might want to repeat a piece of content a set number of times. The Template Toolkit provides two loop directives that deal with both of these situations—FOREACH (also spelled FOR) and WHILE.

Use FOREACH in cases where you know the size of the data set over which you are iterating, or in cases where you need access to loop metadata, such as the next or previous element, the index of the iteration, or the size of the data set. WHILE is useful for performing an action until a condition is true, for looping over a very large data set, or when termination of the loop depends on a condition external to the data set. Both directives are discussed in the sections that follow.

4.4.1 FOREACH

The FOREACH directive defines a block, up to the corresponding END tag, that is processed repeatedly for each item in a list. The basic syntax is:

```
[% FOREACH item IN list %]
   # content of block
[% END %]
```

You can also use = in place of IN if you find that more natural:

```
[% FOREACH item = list %]
   # content of block
[% END %]
```

FOREACH loops over each element in a list and creates an alias to the current item:

```
[% numbers = [ 1 .. 5 ] %]

[% FOREACH num IN numbers %]
   * [% num %]
[% END %]
```

In this example, numbers is an array of five elements, the numbers 1 through 5. In the FOREACH loop, these elements are assigned to num, one at a time, in the order that they occur in numbers:

```
* 1
* 2
* 3
* 4
* 5
```

4.4.1.1 Complex data

The elements of the array can be any kind of complex data:

```
[% fabfour = [
    {
      name       = "John Lennon"
      instrument = "guitar"
```

```
    }
    {
      name       = "Paul McCartney"
      instrument = "bass guitar"
    }
    {
      name       = "George Harrison"
      instrument = "lead guitar"
    }
    {
      name       = "Ringo Starr"
      instrument = "drums"
    }
  ]
%]
[% FOREACH beatle IN fabfour -%]
  [% beatle.name %] played [% beatle.instrument %].
[% END %]
```

The beatle variable is aliased to each hash in the fabfour list, and through it we can access the various elements:

```
John Lennon played guitar.
Paul McCartney played bass guitar.
George Harrison played lead guitar.
Ringo Starr played drums.
```

The original array is not modified, but the elements of the array can be modified within the FOREACH loop.

4.4.1.2 Importing hash array items

When the FOREACH directive is used without specifying a target variable, any iterated values that are hash references will be automatically imported:

```
[% FOREACH fabfour -%]
  [% name %] played [% instrument %].
[% END %]
```

This particular usage creates a localized variable context to prevent the imported hash keys from overwriting any existing variables. The imported definitions and any other variables defined in such a FOREACH loop will be lost at the end of the loop, when the previous context and variable values are restored.

4.4.1.3 Iterating over entries in a hash array

The FOREACH directive can also be used to iterate over the entries in a hash array. Each entry in the hash is returned in sorted order (based on the key) as a hash array containing "key" and "value" items.

```
[% users = {
    tom   = 'Thomas'
    dick  = 'Richard'
```

```
    larry = 'Lawrence'
  }
%]

[% FOREACH user IN users %]
  * [% user.key %] : [% user.value %]
[% END %]
```

The previous example generates the following output:

```
* dick : Richard
* larry : Lawrence
* tom : Thomas
```

To iterate over the keys of a hash, use the keys virtual method on the hash:

```
[% FOREACH key IN hash.keys %]
   * [% key %] : [% hash.$key %]
[% END %]
```

4.4.1.4 The loop iterator object

The underlying implementation of the FOREACH directive involves the creation of a special object called an *iterator*, which maintains metadata about the data set being processed. This object can be accessed within the body of the FOREACH using the special variable loop:

```
[% FOREACH item IN items %]
   [% IF loop.first %]
     <ul>
   [% END %]
       <li>[% item %] ([% loop.count %] of [% loop.size %])</li>
   [% IF loop.last %]
     </ul>
   [% END %]
[% END %]
```

The iterator defines several useful methods that return information about the current loop:

size

Returns the size of the data set, or returns undef if the dataset has not been defined

max

Returns the maximum index number (i.e., the index of the last element), which is equivalent to size - 1

index

Returns the number of the current item, in the range 0 to max

count

Returns the current iteration count in the range 1 to size, equivalent to index + 1 *first*

Returns a Boolean value to indicate whether the iterator is currently on the first iteration of the set

last

Returns a Boolean value to indicate whether the iterator is currently on the last iteration of the set

prev

Returns the previous item in the data set, or returns undef if the iterator is on the first item

next

Returns the next item in the data set, or undef if the iterator is on the last item

An iterator plugin is available that enables you to control how an iterator is created; if an iterator object is passed to a FOREACH loop, it is used as is (a new iterator is not created).

```
[% USE all_data = iterator(list_one.merge(list_two)) %]

[% FOREACH datum = all_data %]

   ...

[% END %]
```

4.4.1.5 Nested FOREACH loops

Nested loops will work as expected, with the loop variable correctly referencing the innermost loop and being restored to any previous value (i.e., an outer loop) at the end of the loop:

```
[% FOREACH group IN grouplist;

     # loop => group iterator

     "Groups:\n" IF loop.first;



     FOREACH user IN group.userlist;

        # loop => user iterator

        "$loop.count: $user.name\n";

     END;



     # loop => group iterator

     "End of Groups\n" IF loop.last;

  END

%]
```

The iterator plugin can also be used to explicitly create an iterator object. This can be useful within nested loops where you need to keep a reference to the outer iterator within the inner loop. The iterator plugin effectively allows you to create an iterator by a name other than loop. See the manpage for Template::Plugin::Iterator for further details.

```
[% USE giter = iterator(grouplist) %]



[% FOREACH group IN giter %]
```

```
  [% FOREACH user IN group.userlist %]
      user #[% loop.count %] in
      group [% giter.count %] is
      named [% user.name %]
  [% END %]
[% END %]
```

4.4.2 WHILE

WHILE loops are used to repeatedly process a template block. This block is enclosed within [% WHILE (test) %] ... [% END %] blocks and can be arbitrarily complex. The test condition follows the same rules as those for IF blocks.

```
[%  total = 0;
    WHILE total <= 100 %]
       Total: [% total;
       total = total + 1;
    END;
%]
```

An assignment can be enclosed in parentheses to evaluate the assigned value:

```
[% WHILE (user = next_user) %]
  [% user.name %]
[% END %]
```

The Template Toolkit uses a fail-safe counter to limit the number of loop iterations to prevent runaway loops that never terminate. If the loop exceeds 1,000 iterations, an undef exception will be thrown, reporting the error:

```
WHILE loop terminated (> 1000 iterations)
```

This number can be adjusted from within Perl by setting the $Template::Directive::WHILE_MAX variable.

4.4.2.1 Flow control: NEXT and LAST

The NEXT directive starts the next iteration in a FOREACH or WHILE loop:

```
[% FOREACH user IN userlist %]
  [% NEXT IF user.isguest %]
  Name: [% user.name %]    Email: [% user.email %]
[% END %]
```

The LAST directive can be used to prematurely exit the loop. BREAK is also provided as an alias for LAST.

```
[% FOREACH match IN results.nsort('score').reverse %]
  [% LAST IF match.score < 50 %]
  [% match.score %] : [% match.url %]
[% END %]
```

See the section titled Section 4.11 later in this chapter for more details.

4.5 Conditionals

Often you don't know exactly what output is required until you process the template. Perhaps your web site should be orange on certain days of the week, or maybe negative numbers should be displayed in red. The Template Toolkit has a number of conditional directives that allow your template to make decisions about what path to take.

A *conditional* controls execution of a block of code, based on the value of a variable. In the Template Toolkit, there are two main conditional directives: IF and SWITCH. In addition, there is the UNLESS directive, which is a negated IF.

4.5.1 IF, ELSIF, ELSE, and UNLESS

The primary directive for *conditional execution* is the IF statement. The basic syntax is:

```
[% IF test %]
   action
[% END %]
```

where action is executed only if test is true (the Template Toolkit's definition of "truth" is explained later in this section). IF statements allow for an optional ELSE clause, which is executed if test is not true. There can be multiple test/action pairs as well; these are written using the ELSIF statement:

```
[% IF today = = "friday" %]
   Yay! It's Friday!
[% ELSIF today = = "monday" %]
   Yuck.  It's Monday.
[% ELSE %]
   ...
[% END %]
```

There can be any number of ELSIF clauses, including none. The ELSE clause is also optional. Because the IF directive defines a block, the END token is not optional.

The test clause can be any statement, even just a single variable name; the extreme case is a test clause of 1—i.e., always true. If the result of this statement is 0 or "" (the empty string), test is considered to be false; everything else is true. Variables that have not been assigned a value, either with DEFAULT or SET, are considered to be false (the value of an undefined variable is an empty string).

More complex statements are possible, such as the earlier example. test can be arbitrarily complex. Other than simple variable value, another common test is equality or comparison: what value does a variable contain? The notation = = is used to compare strings because = is used for assignment—it is an error to try to assign to a variable in an IF statement, to prevent subtle errors and hard-to-diagnose problems. Comparison operators include:

```
= =        Test for equality
!=         Test for inequality
<          Less than
<=         Less than or equal to
>          Greater than
>=         Greater than or equal to
&&, AND    grouping
||, OR     grouping
!, NOT     negation
```

Some of these make sense only for numbers, such as >, >=, <, and <=. NOT is used to reverse the meaning of a test:

```
[% IF NOT today %]

   Error! 'today' not defined!

[% END %]
```

There is a special version of IF that does exactly this: UNLESS.

```
[% UNLESS today %]

   ...
```

UNLESS is exactly equivalent to IF NOT, and often clarifies the intent of the condition (but can be more confusing when combined with ELSIF clauses, even though this is a syntactically legal thing to do).

AND and OR can be used to construct compound statements that might otherwise require nested IF blocks:

```
[% IF today =  = "Friday" AND time >= 1700 %]

   Go home!  It's the weekend!

[% END %]
```

Without grouping, this would need to be:

```
[% IF today =  = "Friday" %]

   [% IF time >= 1700 %]

      Go home!  It's the weekend!

   [% END %]

[% END %]
```

As you can imagine, this would get very tedious for blocks with many options.

4.5.2 SWITCH and CASE

The SWITCH directive makes writing long IF / ELSIF / ELSE statements easier when the test condition needs to be compared to a number of possible outcomes. SWITCH consists of a single statement, which is evaluated once, and a number of CASE statements, against which the evaluated value is compared. For example:

```
[% SWITCH today %]

   [% CASE "Monday" %]

   Hi ho, hi ho, it's off to work we go.


   [% CASE "Friday" %]

   Friday's here, almost time for the weekend!


   [% CASE [ "Saturday" "Sunday" ] %]

   It's the weekend! Party!


   [% CASE %]

   Ho hum, just another workday...

[% END %]
```

The value in today is compared against each successive CASE statement until a match is found; the contents of the matching CASE statement are processed, or the contents of the default CASE statement are processed if no match is found (if there is a default CASE statement, of course). An equivalent IF / ELSIF / ELSE block would look like this:

```
[% IF today =  = "Saturday" OR today =  = "Sunday" %]
```

```
    It's the weekend! Party!
[% ELSIF today =  = "Monday" %]
    Hi ho, hi ho, it's off to work we go.
[% ELSIF today =  = "Friday" %]
    Friday's here, almost time for the weekend!
[% ELSE %]
    Ho hum, just another workday...
[% END %]
```

The SWITCH statement is cleaner and there is less syntax to maintain. Most important, however, is that if the *test* statement requires computation instead of just variable comparison, the SWITCH will be more efficient and has less potential for side effects.

4.6 Filters

One of the problems with templates is that you can never be completely sure what content will be produced at the end. This is particularly true if you are pulling in some of your data from an external source. Perhaps you are producing an HTML page from news stories that have been entered into a database by reporters. You can't be sure the stories don't contain characters such as < or & that should be plain text but will be interpreted as HTML. Or perhaps you have room for only a certain number of characters and you don't know how long a story will be.

The Template Toolkit provides filters to deal with these cases. A filter can be applied to part of a template and will postprocess those parts in a defined manner. For example, the html filter converts troublesome characters to their equivalent HTML entities, and the truncate filter will truncate text to a given length.

The FILTER directive introduces a filter, which operates on a block:

```
[% FILTER html %]

  HTML text may have < and > characters embedded

  that you want converted to the correct HTML entities.

[% END %]
```

The previous example produces the following output:

```
HTML text may have &lt; and &gt; characters embedded

that you want converted to the correct HTML entities.
```

The FILTER directive can also follow various other nonblock directives. For example:

```
[% INCLUDE mytext FILTER html %]
```

The | character can also be used as an alias for FILTER:

```
[% INCLUDE mytext | html %]
```

Multiple filters can be chained together and will be called in sequence:

```
[% INCLUDE mytext FILTER html FILTER html_para %]
```

or:

```
[% INCLUDE mytext | html | html_para %]
```

A number of standard filters are provided with the Template Toolkit; these are detailed in Chapter 5.

4.7 Plugins

It is obviously impossible for the Template Toolkit to do everything that everyone might want to do with it. For one thing, we haven't heard of every possible piece of software that you might want to talk to, and for another, no one would want a template processor that is infinite in size! Instead, we provided the plugin mechanism, which makes it possible to write extensions to the Template Toolkit. This is a far saner solution.

Plugins are externally defined extensions that can be dynamically loaded into templates to provide functionality. A plugin is a regular Perl module that conforms to a particular object-oriented interface, allowing it to be loaded into and used automatically by the Template Toolkit. The next subsections discuss directives for working with plugins.

4.7.1 USE

The *USE* directive loads and initializes "plugin" extension modules:

```
[% USE date %]
```

This makes a date plugin object available to the template, which can be used by referencing the variable date:

```
Today is [% date.format(date.now, "%A") %].
```

which might return:

```
Today is Monday.
```

The plugin name is case sensitive and will be appended to the PLUGIN_BASE value (which defaults to *Template::Plugin*) to construct a full module name. Any periods (i.e., .), in the name will be converted to ::.

```
[% USE MyPlugin %]     #  => Template::Plugin::MyPlugin

[% USE Foo.Bar  %]     #  => Template::Plugin::Foo::Bar
```

Various standard plugins are included with the Template Toolkit (see Chapter 6). These can be specified in lowercase and are mapped to the appropriate name:

```
[% USE cgi   %]       # => Template::Plugin::CGI

[% USE table %]       # => Template::Plugin::Table
```

Any additional parameters supplied in parentheses after the plugin name also will be passed to the *new( )* constructor. A reference to the current Template::Context object is always passed as the first parameter. Thus:

```
[% USE MyPlugin('foo', 123) %]
```

is equivalent to:

```
Template::Plugin::MyPlugin->new($context, 'foo', 123);
```

Named parameters may also be specified. These are collated into a hash that is passed by reference as the last parameter to the constructor, as per the general code-calling interface. Thus:

```
[% USE url('/cgi-bin/foo', mode='submit', debug=1) %]
```

is equivalent to:

```
Template::Plugin::URL->new($context, '/cgi-bin/foo',

                           { mode => 'submit', debug => 1 });
```

The plugin may represent any data type—a simple variable, hash, list, or code reference—but in general it will be an object reference. Methods can be called on the object (or on the relevant members of the specific data type) in the usual way:

```
[% USE table(mydata, rows=3) %]



[% FOREACH row = table.rows %]

   <tr>

  [% FOREACH item = row %]

     <td>[% item %]</td>
```

```
  [% END %]

  </tr>

[% END %]
```

A plugin can be referenced by an alternative name:

```
[% USE scores = table(myscores, cols=5) %]



[% FOREACH row = scores.rows %]

  ...

[% END %]
```

You can use this approach to create multiple plugin objects with different configurations. This example shows how the format plugin is used to create subroutines bound to variables for formatting text as per *printf( )*.

```
[% USE bold = format('<b>%s</b>') %]

[% USE ital = format('<i>%s</i>') %]



[% bold('This is bold')   %]

[% ital('This is italic') %]
```

The previous example generates the following output:

```
<b>This is bold</b>

<i>This is italic</i>
```

This next example shows how the *URL* plugin can be used to build dynamic URLs from a base part and optional query parameters:

```
[% USE mycgi = URL('/cgi-bin/foo.pl', debug=1) %]

<a href="[% mycgi %]">...

<a href="[% mycgi(mode='submit') %]"...
```

The previous example generates the following output:

```
<a href="/cgi-bin/foo.pl?debug=1">...

<a href="/cgi-bin/foo.pl?mode=submit&debug=1">...
```

The *LOAD_PERL* option (disabled by default) provides a further way by which external Perl modules may be loaded. If a regular Perl module (i.e., not a Template::Plugin::* or other module relative to some PLUGIN_BASE) supports an object-oriented interface and a *new( )* constructor, it can be loaded and instantiated automatically. The following trivial example shows how the IO::File module might be used:

```
[% USE file = IO.File('/tmp/mydata') %]



[% WHILE (line = file.getline) %]

  <!-- [% line %] -->

[% END %]
```

Chapter 6 discusses plugins in excruciating detail.

4.8 Macros

Sometimes Template Toolkit code can get very complicated. You can often have complex pieces of code that get repeated a number of times throughout your template. One solution to this problem is to extract the code into another template and call it with PROCESS whenever it is needed:

```
[% PROCESS my/gnarly/code day='Monday' %]

...later...

[% PROCESS my/gnarly/code day='Tuesday' %]
```

This idea works well for larger chunks of code, but it can be a little unwieldy if used often. A far better idea is to define a macro. A macro is a piece of arbitrary Template Toolkit code that is given a name, enabling you to call it later in the template. For example:

```
[% USE date -%]

[% MACRO now GET date.format(date.now, '%H:%M:%S') -%]

[% MACRO today GET date.format(date.now, '%Y-%m-%d') -%]
```

This defines two macros called now and today that will output the current time and date whenever they are called in the template:

```
[% now %] [% today %]
```

The following subsection introduces the directive for working with macros.

4.8.1 MACRO

The MACRO directive allows you to define a directive or directive block that is evaluated each time the macro is called:

```
[% MACRO header INCLUDE header %]
```

Calling the macro as:

```
[% header %]
```

is then equivalent to:

```
[% INCLUDE header %]
```

Macros can be passed named parameters when called. These values remain local to the macro. Therefore, calling the macro as:

```
[% header(title='Hello World') %]
```

is equivalent to:

```
[% INCLUDE header title='Hello World' %]
```

A *MACRO* definition may include parameter names. Values passed to the macros are then mapped to these local variables. Other named parameters may follow these.

```
[% MACRO header(title) INCLUDE header %]



[% header('Hello World') %]

[% header('Hello World', bgcol='#123456') %]
```

There are equivalent to:

```
[% INCLUDE header title='Hello World' %]

[% INCLUDE header title='Hello World' bgcol='#123456' %]
```

Here's another example, defining a macro for display numbers in comma-delimited groups of three, using the *chunk* and *join* virtual method:

```
[% MACRO number(n) GET n.chunk(-3).join(',') %]


[% number(1234567) %]    # 1,234,567
```

A *MACRO* may precede any directive, including block directives, but must conform to the structure of the directive:

```
[%  terms = {
        sass  = 'know, be aware of, meet, have sex with',
        hoopy = 'really together guy',
        frood = 'really, amazingly together guy'
    };

    MACRO explain(term)
        IF (explanation = terms.$term);
            "$term ($explanation)";
        ELSE;
            term;
        END;
%]
```

Here we define the explain(term) macro as an IF / ELSE directive. It consults a hash table to locate an explanation for the term passed as an argument. It generates a string containing the term and explanation, or the term by itself if no explanation is found.

```
Hey you [% explain('sass') %] that
[% explain('hoopy') %] Ford Prefect?
There's a [% explain('frood') %]
who really knows where his towel is.
```

This generates the following output:

```
Hey you sass (know, be aware of, meet, have sex with) that
hoopy (really together guy) Ford Prefect?
There's a frood (really, amazingly together guy)
who really knows where his towel is.
```

A *MACRO* can also be defined as an anonymous *BLOCK*. The block will be evaluated each time the macro is called.

```
[%  MACRO translate(text)
      BLOCK;
        words = [ ];
        FOREACH word IN text.split;
          IF (explanation = terms.$word);
            words.push("$word ($explanation)");
          ELSE;
            words.push(word);
          END;
```

```
    END;
    words.join(' ');
   END
%]
```

This macro splits the text passed as an argument into words, attempts to explain them, and then joins them back up into a single piece of text:

```
[% translate(
    "Hey you sass that hoopy Ford Prefect?
     There's a frood who really knows where
     his towel is."
  )
%]
```

This is the output generated by the previous template fragment:

```
Hey you sass (know, be aware of, meet, have sex with)
that hoopy (really together guy)
Ford Prefect? There's a frood (really, amazingly together guy)
who really knows where his towel is.
```

A MACRO can also be defined as a PERL block, but will require the EVAL_PERL option to be set:

```
[% MACRO triple(n) PERL %]
    my $n = $stash->get('n');
    print $n * 3;
[% END -%]
```

The PERL and RAWPERL directives are covered at the end of this chapter.

4.9 Template Metadata

The Template Toolkit compiles a template into a Perl object (an instance of the class Template::Document). This object contains Perl code that reproduces the required behavior of the template. You can access the data in this object via the template variable.

The Template::Document has access to various items of metadata about the template that you can access via template. This always includes the name of the template and the last modification time, so it is always possible to include things such as this in your template:

```
[% USE date(format => '%Y-%m-%d %H:%M:%S') %]

[% template.name %]

Last modified: [% date.format(template.modtime) %]
```

Further metadata items can be added using the META directive, discussed next. These new items will also be available through the template variable.

```
[% META moon_phase = 'first quarter' -%]

Phase of moon: [% template.moon_phase %]
```

4.9.1 META

The *META* directive allows simple metadata items to be defined within a template. These are evaluated when the template is parsed, and as such may contain only simple values (e.g., it's not possible to interpolate other variable values into *META* variables).

```
[% META

   title   = 'The Cat in the Hat'

   author  = 'Dr. Seuss'

   version = 1.23

%]
```

The *template* variable contains a reference to the main template being processed. These metadata items may be retrieved as attributes of the template.

```
<h1>[% template.title %]</h1>

<h2>[% template.author %]</h2>
```

The *name* and *modtime* metadata items are automatically defined for each template to contain its name and modification time in seconds since the epoch:

```
[% USE date %]              # use Date plugin to format time

...

[% template.name %] last modified

at [% date.format(template.modtime) %]
```

The *PRE_PROCESS* and *POST_PROCESS* options allow common headers and footers to be added to all templates. The *template* reference is correctly defined when these templates are processed, allowing headers and footers to reference metadata items from the main template:

```
$tt = Template->new({

    PRE_PROCESS  => 'header',

    POST_PROCESS => 'footer',

});


$tt->process('cat_in_hat');
```

header:

```
<html>

<head>

<title>[% template.title %]</title>

</head>

<body>
```

cat_in_hat:

```
[% META

   title   = 'The Cat in the Hat'

   author  = 'Dr. Seuss'

   version = 1.23

   year    = 2000

%]


The cat in the hat sat on the mat.
```

footer:

```
<hr />

&copy; [% template.year %] [% template.author %]

</body>

</html>
```

The output generated from the preceeding example is:

```
<html>

<head>

<title>The Cat in the Hat</title>

</head>

<body>


The cat in the hat sat on the mat.


<hr />

&copy; 2000 Dr. Seuss

</body>

</html>
```

4.10 Exception Handling

No matter how careful you are, things always go wrong. Errors are a fact of life. Your templates could contain bad code and fail to compile. Or you could get an error thrown from the Template Toolkit—maybe it can't find the header file you asked for. Or your back-end code could raise an error—you failed to connect to the required database. The Template Toolkit wouldn't be of much use if common errors such as these caused it to keel over and die. That's why it provides an exception-handling mechanism in the form of TRY...CATCH.

Exceptions are just a fancy way of saying errors. They're structured as objects so that an error can have a type (just a word to identify the kind of error that occurred, such as database, user, or file) and an info field that provides further information about the specifics of the error. They get thrown just like regular errors, via Perl's *die*, but rather than saying die 'bad apple', we say THROW bad apple.

You don't have to explicitly add code to handle errors. If you don't and an error occurs, it gets reported in the usual way. But if you know that errors might occur and you have a sensible way of recovering from them, it's good to add *TRY...CATCH* to do that.

Using the exception mechanism doesn't force you to worry about all errors that might occur. You can filter on the type of error and just look out for your one custom error code to catch, letting everything else pass through. Exceptions can also be nested, so you can catch them at the most appropriate level in your template.

4.10.1 TRY / THROW / CATCH / FINAL

The Template Toolkit supports fully functional, nested exception handling. The *TRY* directive introduces an exception-handling scope that continues until the matching *END* directive. Any errors that occur within that block will be caught and can be handled by one of the *CATCH* blocks defined.

```
[% TRY %]
  ...blah...blah...
  [% CALL somecode %]
  ...etc...
  [% INCLUDE someblock %]
  ...and so on...
[% CATCH %]
  An error occurred!
[% END %]
```

Errors are raised as exceptions (objects of the Template::Exception class) and contain two fields, *type* and *info*. The exception *type* can be any string containing letters, numbers, "_" or ".", and is used to indicate the kind of error that occurred. The *info* field contains an error message indicating what actually went wrong. Within a *CATCH* block, the exception object is aliased to the *error* variable. You can access the *type* and *info* fields directly.

```
[% mydsn = 'dbi:MySQL:foobar' %]
...

[% TRY %]
  [% USE DBI(mydsn) %]
[% CATCH %]
  ERROR! Type: [% error.type %]
         Info: [% error.info %]
[% END %]
```

The previous example generates the following output (assuming a nonexistent database called foobar):

```
ERROR!  Type: DBI

        Info: Unknown database "foobar"
```

The error variable can also be specified by itself and will return a string of the form $type error - $info:

```
...

[% CATCH %]

ERROR: [% error %]

[% END %]
```

The previous example generates the following output:

```
ERROR: DBI error - Unknown database "foobar"
```

Each *CATCH* block may be specified with a particular exception type denoting the kind of error that it should catch. Multiple *CATCH* blocks can be provided to handle different types of exceptions that may be thrown in the *TRY* block. A *CATCH* block specified without any type, as in the previous example, is a default handler that will catch any otherwise uncaught exceptions. This also can be specified as [% CATCH DEFAULT %].

```
[% TRY %]

  [% INCLUDE myfile %]

  [% USE DBI(mydsn) %]

  [% CALL somecode %]

  ...

[% CATCH file %]

  File Error! [% error.info %]

[% CATCH DBI %]

  [% INCLUDE database/error.html %]

[% CATCH %]

  [% error %]

[% END %]
```

Remember that you can specify multiple directives within a single tag, each delimited by ;. Thus, you might prefer to write your simple *CATCH* blocks more succinctly as:

```
[% TRY %]

  ...

[% CATCH file; "File Error! $error.info" %]

[% CATCH DBI;  INCLUDE database/error.html %]

[% CATCH; error %]

[% END %]
```

or even:

```
[% TRY %]

  ...

[% CATCH file ;

      "File Error! $error.info" ;

  CATCH DBI ;

      INCLUDE database/error.html ;
```

```
  CATCH ;
    error ;
  END
%]
```

The *DBI* plugin throws exceptions of the *DBI* type (in case that wasn't already obvious). The other specific exception caught here is of the *file* type.

A *file* error is automatically thrown by the Template Toolkit when it can't find a file, or fails to load, parse, or process a file that has been requested by an *INCLUDE*, *PROCESS*, *INSERT*, or *WRAPPER* directive. If myfile can't be found in the previous example, the [% INCLUDE myfile %] directive will raise a file exception, which is then caught by the [% CATCH file %] block, generating the output:

```
File Error! myfile: not found
```

Note that the *DEFAULT* option (disabled by default) allows you to specify a default file to be used any time a template file can't be found. This will prevent file exceptions from ever being raised when a nonexistent file is requested (unless, of course, the DEFAULT file doesn't exist). Errors encountered once the file has been found (i.e., read error, parse error) will be raised as file exceptions as per usual.

Uncaught exceptions (i.e., the *TRY* block doesn't have a type-specific or default *CATCH* handler) may be caught by enclosing *TRY* blocks that can be nested indefinitely across multiple templates. If the error isn't caught at any level, processing will stop and the Template *process( )* method will return a false value to the caller. The relevant Template::Exception object can be retrieved by calling the *error( )* method.

```
[% TRY %]
  ...
  [% TRY %]
    [% INCLUDE $user.header %]
  [% CATCH file %]
    [% INCLUDE header %]
  [% END %]
  ...
[% CATCH DBI %]
  [% INCLUDE database/error.html %]
[% END %]
```

In this example, the inner *TRY* block is used to ensure that the first *INCLUDE* directive works as expected. We're using a variable to provide the name of the template we want to include, *user.header*, and it's possible this contains the name of a nonexistent template, or perhaps one containing invalid template directives. If the *INCLUDE* fails with a file error, we *CATCH* it in the inner block and *INCLUDE* the default header file instead. Any DBI errors that occur within the scope of the outer *TRY* block will be caught in the relevant *CATCH* block, causing the *database/error.html* template to be processed. Note that included templates inherit all currently defined template variables, so these error files can quite happily access the error variable to retrieve information about the currently caught exception. For example:

database/error.html:

```
<h2>Database Error</h2>

A database error has occurred: [% error.info %]
```

You can also specify a *FINAL* block. This is always processed regardless of the outcome of the *TRY* and/or *CATCH* block. If an exception is uncaught, the *FINAL* block is processed before jumping to the enclosing block or returning to the caller.

```
[% TRY %]
  ...
[% CATCH this %]
  ...
```

```
[% CATCH that %]

   ...

[% FINAL %]

   All done!

[% END %]
```

The output from the *TRY* block is left intact up to the point where an exception occurs. For example, this template:

```
[% TRY %]

   This gets printed

   [% THROW food 'carrots' %]

   This doesn't

[% CATCH food %]

   culinary delights: [% error.info %]

[% END %]
```

generates the following output:

```
This gets printed

culinary delights: carrots
```

The *CLEAR* directive can be used in a *CATCH* or *FINAL* block to clear any output created in the *TRY* block. For example, this template:

```
[% TRY %]

   This gets printed

   [% THROW food 'carrots' %]

   This doesn't

[% CATCH food %]

   [% CLEAR %]

   culinary delights: [% error.info %]

[% END %]
```

generates the following output:

```
culinary delights: carrots
```

Exception types are hierarchical, with each level being separated by the familiar dot operator. A DBI.connect exception is a more specific kind of *DBI* error. Similarly, a myown.error.barf is a more specific kind of myown.error type, which itself is also a myown error. A *CATCH* handler that specifies a general exception type (such as *DBI* or myown.error) will also catch more specific types that have the same prefix as long as a more specific handler isn't defined. Note that the order in which *CATCH* handlers are defined is irrelevant; a more specific handler will always catch an exception in preference to a more generic or default one.

```
[% TRY %]

   ...

[% CATCH DBI ;

     INCLUDE database/error.html ;

   CATCH DBI.connect ;

     INCLUDE database/connect.html ;

   CATCH ;
```

```
    INCLUDE error.html ;
  END
%]
```

In this example, a DBI.connect error has its own handler, a more general *DBI* block is used for all other *DBI* or *DBI.** errors, and a default handler catches everything else.

Exceptions can be raised in a template using the *THROW* directive. The first parameter is the exception type, which doesn't need to be quoted (but can be, it's the same as *INCLUDE*), followed by the relevant error message, which can be any regular value such as a quoted string, variable, etc.

```
[% THROW food "Missing ingredients: $recipe.error" %]


[% THROW user.login 'no user id: please login' %]


[% THROW $myerror.type "My Error: $myerror.info" %]
```

It's also possible to specify additional positional or named parameters to the *THROW* directive if you want to pass more than just a simple message back as the error info field:

```
[% THROW food 'eggs' 'flour' msg='Missing Ingredients' %]
```

In this case, the error info field will be a hash array containing the named arguments—in this case msg => 'Missing Ingredients'—and an args item that contains a list of the positional arguments—in this case eggs and flour. The error type field remains unchanged; here it is set to food.

```
[% CATCH food %]
  [% error.info.msg %]
  [% FOREACH item = error.info.args %]
     * [% item %]
  [% END %]
[% END %]
```

This produces the output:

```
Missing Ingredients
  * eggs
  * flour
```

In addition to specifying individual positional arguments as [% error.info.args.n %], the info hash contains keys directly pointing to the positional arguments, as a convenient shortcut:

```
[% error.info.0 %]   # same as [% error.info.args.0 %]
```

Exceptions can also be thrown from Perl code that you've bound to template variables, or defined as a plugin or other extension. To raise an exception, call *die( )* passing a reference to a Template::Exception object as the argument. This will then be caught by any enclosing *TRY* blocks from where the code was called.

```
use Template::Exception;
...


my $vars = {
    foo => sub {
        # ... do something ...
        die Template::Exception->new('myerr.naughty',
                                     'Bad, bad error');
```

```
    },
};
```

Therefore, this template:

```
[% TRY %]
   ...
   [% foo %]
   ...
[% CATCH myerr ;
     "Error: $error" ;
   END
%]
```

produces the following output:

```
Error: myerr.naughty error - Bad, bad error
```

The info field can also be a reference to another object or data structure, if required:

```
die Template::Exception->new('myerror', {
    module => 'foo.pl',
    errors => [ 'bad permissions', 'naughty boy' ],
});
```

Later, it can be used in a template:

```
[% TRY %]
   ...
[% CATCH myerror %]
   [% error.info.errors.size or 'no';
      error.info.errors.size =  = 1 ? ' error' : ' errors' -%]
   in [% error.info.module %]:
      [% error.info.errors.join(', ') %].
[% END %]
```

generating the output:

```
2 errors in foo.pl:
   bad permissions, naughty boy.
```

You can also call *die( )* with a single string, as is common in much existing Perl code. This will automatically be converted to an exception of the *undef* type (that's the literal string `undef', not the undefined value). If the string isn't terminated with a newline, Perl will append the familiar at $file line $line message.

```
sub foo {
    # ... do something ...
    die "I'm sorry, Dave, I can't do that\n";
}
```

If you're writing a plugin, or some extension code that has the current Template::Context in scope (you can safely skip this section if this means nothing to you), you can also raise an exception by calling the *context throw( )* method. You can pass it a Template::Exception object reference, a pair of *($type, $info)* parameters, or just a *$info* string to create an exception of *undef* type.

```
$context->throw($e);            # exception object

$context->throw('Denied');      # 'undef' type

$context->throw('user.passwd', 'Bad Password');
```

4.10.2 CLEAR

The *CLEAR* directive can be used to clear the output buffer for the current enclosing block. It is most commonly used to clear the output generated from a *TRY* block up to the point where the error occurred.

```
[% TRY %]

   blah blah blah            # this is normally left intact

   [% THROW some 'error' %]  # up to the point of error

   ...

[% CATCH %]

   [% CLEAR %]               # clear the TRY output

   [% error %]               # print error string

[% END %]
```

4.11 Flow Control

Flow control is about making unexpected changes to the execution order of a template. This can be as simple as ending a FOREACH loop early, or as significant as ending the entire template processing process. These are generally exceptional cases, so you probably won't need to use flow-control directives that often, but we discuss them here just in case.

4.11.1 RETURN

The *RETURN* directive can be used to stop processing the current template and return to the template from which it was called, resuming processing at the point immediately after the *INCLUDE*, *PROCESS*, or *WRAPPER* directive. If there is no enclosing template, the Template *process( )* method will return to the calling code with a true value.

```
Before
[% INCLUDE half_wit %]
After

[% BLOCK half_wit %]
This is just half...
[% RETURN %]
...a complete block
[% END %]
```

The previous example produces the following output:

```
Before
This is just half...
After
```

4.11.2 STOP

The *STOP* directive can be used to indicate that the processor should stop gracefully without processing any more of the template document. This is a planned stop, and the Template *process( )* method will return a true value to the caller. This indicates that the template was processed successfully according to the directives within it.

```
[% IF something.terrible.happened %]
  [% INCLUDE fatal/error.html %]
  [% STOP %]
[% END %]

[% TRY %]
  [% USE DBI(mydsn) %]
  ...
[% CATCH DBI.connect %]
  <p>Cannot connect to the database: [% error.info %]</p>
  <br>
  We apologize for the inconvenience.  The cleaning lady
```

```
  has removed the server power to plug in her vacuum cleaner.

  Please try again later.

  </p>

  [% INCLUDE footer %]

  [% STOP %]

[% END %]
```

4.11.3 NEXT

The *NEXT* directive can be used to start the next iteration of a *FOREACH* or *WHILE* loop:

```
[% FOREACH user = userlist %]

  [% NEXT IF user.isguest %]

  Name: [% user.name %]    Email: [% user.email %]

[% END %]
```

4.11.4 LAST

The *LAST* directive can be used to prematurely exit a *FOREACH* or *WHILE* loop:

```
[% FOREACH user = userlist %]

  Name: [% user.name %]    Email: [% user.email %]

  [% LAST IF some.condition %]

[% END %]
```

BREAK can also be used as an alias for *LAST*.

4.12 Debugging

It's possible that you won't get everything just right in your templates the first time you write them. If you have problems working out what exactly is going on as the Template Toolkit is processing your template, the DEBUG directive can help you.

The DEBUG directive enables and disables directive debug messages within a template. It is used with an on or off parameter to enable or disable directive debugging messages from that point forward. When enabled, the output of each directive in the generated output will be prefixed by a comment indicating the file, line, and original directive text.

```
[% DEBUG on %]

directive debugging is on (assuming DEBUG option is set to true)

[% DEBUG off %]

directive debugging is off
```

The *format* parameter can be used to change the format of the debugging message:

```
[% DEBUG format '<!-- $file line $line : [% $text %] -->' %]
```

The DEBUG configuration option must be set to include DEBUG_DIRS for the DEBUG directives to have any effect. If DEBUG_DIRS is not set, the parser will automatically ignore and remove any DEBUG directives.

4.13 Perl Blocks

The Template Toolkit directives that we have seen up to now together define a presentation language that allows you to do just about anything you need to in order to control the display of your data. This is in keeping with the Template Toolkit philosophy of separating processing from presentation.

However, there may be times when you want to go beyond what Template Toolkit offers you. Very occasionally you might need the power of a full programming language within your templates. When nothing else will do, the Template Toolkit also gives you the option of embedding Perl directly in your templates in PERL and RAWPERL directive blocks.

Using PERL and RAWPERL blocks isn't something that is widely encouraged because it tends to make templates messy and hard to read. It also leads to a poor separation of concerns when you mix application code with presentation templates. However, the Template Toolkit doesn't enforce this separation, so you can embed Perl code inside your templates if you really want to. Because we don't encourage it, this feature is disabled by default. You will have to enable the EVAL_PERL configuration option to embed Perl code.

4.13.1 PERL

The PERL directive allows you to embed a block of Perl code in a template. It looks like this:

```
[% PERL %]
  print "Hello world\n"
[% END %]
```

The EVAL_PERL configuration option must be enabled in order to use PERL blocks. If you try to use a PERL block when EVAL_PERL is disabled, a perl exception will be thrown with the message `EVAL_PERL not set':

```
my $template = Template->new({
    EVAL_PERL => 1,
});
```

The Template Toolkit evaluates Perl code in the Template::Perl package. A number of special variables are predefined, providing access to various Template Toolkit objects.

The $context package variable contains a reference to the current Template::Context object. This can be used to access the functionality of the Template Toolkit to process other templates, and load plugins, filters, etc.:

```
[% PERL %]
  print $context->include('myfile');
[% END %]
```

The $stash variable contains a reference to the top-level stash object, which manages template variables. Through this, variable values can be retrieved and updated.

```
[% PERL %]
  $stash->set(foo => 'bar');
  print "foo value: ", $stash->get('foo');
[% END %]
```

The previous example generates the following output:

```
foo value: bar
```

Output is generated from the PERL block by calling print. Before evaluating the code, a filehandle called Template::Perl::PERLOUT is set up and selected as the default output filehandle. This will be connected to whatever output device was defined in the call to process. Your code should use this filehandle instead of STDOUT.

```
[% PERL %]
   print "foo\n";                          # OK
   print PERLOUT "bar\n";                  # OK, same as above
   print Template::Perl::PERLOUT "baz\n";   # OK, same as above
   print STDOUT "qux\n";                   # WRONG!
[% END %]
```

The PERL block may contain other template directives. These are processed before the Perl code is evaluated.

```
[% name = 'Fred Smith' %]

[% PERL %]
   print "[% name %]\n";
[% END %]
```

Thus, the Perl code in the previous example is evaluated as:

```
print "Fred Smith\n";
```

Exceptions may be thrown from within PERL blocks via die, and will be correctly caught by enclosing *TRY* blocks:

```
[% TRY %]
   [% PERL %]
      die "nothing to live for\n";
   [% END %]
[% CATCH %]
   error: [% error.info %]
[% END %]
```

The previous example generates the following output:

```
error: nothing to live for
```

4.13.2 RAWPERL

The Template Toolkit parser reads a source template and generates the text of a Perl subroutine as output. It then uses *eval( )* to evaluate it into a subroutine reference. This subroutine is then called to process the template, passing a reference to the current Template::Context object through which the functionality of the Template Toolkit can be accessed. The subroutine reference can be cached, allowing the template to be processed repeatedly without requiring any further parsing.

For example, a template such as:

```
[% PROCESS header %]
The [% animal %] sat on the [% location %]
[% PROCESS footer %]
```

is converted into the following Perl subroutine definition:

```
sub {
    my $context = shift;
    my $stash   = $context->stash;
    my $output  = '';
```

```
    my $error;

    eval { BLOCK: {
        $output .=  $context->process('header');
        $output .=  "The ";
        $output .=  $stash->get('animal');
        $output .=  " sat on the ";
        $output .=  $stash->get('location');
        $output .=  $context->process('footer');
        $output .=  "\n";
    } };
    if ($@) {
        $error = $context->catch($@, \$output);
        die $error unless $error->type eq 'return';
    }

    return $output;
}
```

To examine the Perl code generated, such as in the previous example, set the $Template::Parser::DEBUG package variable to any true value. You can also set the $Template::Directive::PRETTY variable to true to have the code formatted in a readable manner for human consumption. The source code for each generated template subroutine will be printed to STDERR on compilation (i.e., the first time a template is used).

```
$Template::Parser::DEBUG = 1;
$Template::Directive::PRETTY = 1;

...

$tt->process($file, $vars)
    || die $tt->error( ), "\n";
```

The PERL ... END construct allows Perl code to be embedded into a template (when the *EVAL_PERL* option is set), but it is evaluated at "runtime" using *eval( )* each time the template subroutine is called. This is inherently flexible but not as efficient as it could be, especially in a persistent server environment where a template may be processed many times.

The *RAWPERL* directive allows you to write Perl code that is integrated directly into the generated Perl subroutine text. It is evaluated once at compile time and is stored in cached form as part of the compiled template subroutine. This makes *RAWPERL* blocks more efficient than *PERL* blocks.

The downside is that you must code much closer to the metal. Within *PERL* blocks, you can call *print( )* to generate some output. *RAWPERL* blocks don't afford such luxury. The code is inserted directly into the generated subroutine text and should conform to the convention of appending to the $output variable.

```
[% PROCESS  header %]

[% RAWPERL %]
   $output .= "Some output\n";
```

```
   ...
   $output .= "Some more output\n";
[% END %]
```

The critical section of the generated subroutine for this example would then look something like this:

```
...
eval { BLOCK: {
    $output .=  $context->process('header');
    $output .=  "\n";
    $output .= "Some output\n";
    ...
    $output .= "Some more output\n";
    $output .=  "\n";
} };
...
```

As with *PERL* blocks, the $context and $stash references are predefined and available for use within *RAWPERL* code.

Only very advanced Template Toolkit users will ever need to use a *RAWPERL* block.

Chapter 5. Filters

Filters are a powerful feature of the Template Toolkit that allow you to postprocess parts of the output of your template in many different ways. A number of filters for carrying out common tasks are included with the standard Template Toolkit distribution, and it is possible to extend this set by writing your own.

A good example of a filter that comes with the Template Toolkit is the html filter. In an HTML document, a number of characters have special meanings, so if you want these characters to appear in your document they need to be converted to *HTML Entities*. The html filter converts the characters <, >, ", and & to <, >, ", and &, respectively.[1]

> [1] There is also another filter called html_entity, which converts far more characters.

Example 5-1 shows the html filter in action. Without the filter, the JavaScript section in the example would be treated as actual JavaScript code and executed. The filter converts the < characters, thereby changing the JavaScript to text that would be displayed by a browser rather than being executed.

Example 5-1. Filtering Javascript

```
<p>Here is what the JavaScript should look like:</p>

<pre>

[% FILTER html %]

<script language="JavaScript" type="text/javascript">

<!--

document.writeln("Hello, world");

//-->

</script>

[% END %]

</pre>
```

The processed document looks like this:

```
<p>Here is what the JavaScript should look like:</p>

<pre>



&lt;script language=&quot;JavaScript&quot; type=&quot;text/javascript&quot;&gt;

&lt;!--

    document.writeln(&quot;Hello, world&quot;);

//--&gt;

&lt;/script&gt;

</pre>
```

This example also demonstrates a good reason for using filters. The kinds of transformations that a filter makes might well be appropriate only for a particular output medium. For example, the html filter will be used only on HTML documents that are being sent to a browser. If you were printing out the document for some reason, the html filter would only make it harder to follow. Having the *FILTER* functionality available as a postprocessing option makes it easy to decide whether to use it in certain circumstances, and easy to add it to certain parts of a template without changing the way that most of the template works.

In Example 5-1, we used the block syntax for using the *FILTER* directive. This is useful for filtering large parts of a template. If you are filtering the output from a single tag, there is an inline version of the syntax, as shown in Example 5-2.

Example 5-2. Formatting numbers

```
[% pi = 3.1415926536;

 pi FILTER format('%0.3f')

%]
```

This example uses the format filter, which reformats data using format definitions such as those used by the printf function common in many programming languages. In the example, we reformat a decimal number to display only two decimal places (note also that the last digit displayed is rounded up).

The processed output looks like this:

```
3.142
```

It is possible to abbreviate this even further. The pipe character (|) can be used as a synonym for FILTER, as shown in Example 5-3.

Example 5-3. Filtering using the pipe symbol

```
[% pi = 3.1415926536;

 pi | format('%0.3f')

%]
```

These two examples also demonstrate the differences between the two types of filters. The html filter is an example of a *static filter*, whereby the filter has the same effect each time it is used. The format filter is an example of a *dynamic filter*, whereby the exact transformation is controlled by a parameter that is passed to the filter on each use.

In this chapter, we look at the different ways you can use filters in your own templates, and also look at the standard filters that are part of the Template Toolkit.

5.1 Using Filters

As we have seen, a filter is used to postprocess the text from a template. The filter acts after any other template processing on the text and transforms the text before the output phase. Example 5-1 shows the format filter being used to put HTML comment characters around a piece of text.

Example 5-4. Using the format filter to add comments

```
[% text = "The white zone is for loading and unloading only." %]


[% FILTER format("<!-- %s -->");
  text;
END
%]
```

Example 5-1 generates the following output:

```
<!-- The white zone is for loading and unloading only. -->
```

Filters can be invoked in two different ways—either by enclosing a block of template markup between the FILTER and END directives, as in:

```
[% FILTER html %] ... [% END %]
```

or in side-effect notation with the FILTER coming after the item to be filtered:

```
[% text FILTER html %]
```

In the second form, the pipe symbol (|) can be used as an alias for the FILTER keyword to give a more Unix-like pipeline feel:

```
[% text | truncate(30) | format("<!-- %s -->") %]
```

As the previous example shows, a number of FILTERs can be chained together. The filters are applied from left to right.

Filters can be applied to many Template Toolkit expressions other than plain strings and scalar variables, including any block directive:

```
[% FILTER indent("> ") %]
[% INSERT "mail.txt" %]
[% END %]
```

Or, more concisely:

```
[% INSERT "mail.txt" | indent("> ") %]
```

5.2 Standard Template Toolkit Filters

The Template Toolkit comes with a large number of preinstalled filters. In this section, we will take a look at these standard filters and see examples of their usage.

5.2.1 collapse

The collapse filter replaces any amount of whitespace with a single space character. It uses Perl's definition of whitespace, which includes spaces, tabs, carriage returns, newlines, and a few more esoteric characters. Example 5-2 gives an example of using this filter.

Example 5-5. The collapse filter

```
[% FILTER collapse %]
You'll   love
  it,   it's            a    way
       of      life.
[% END %]
```

The output is nice and clean:

```
You'll love it, it's a way of life.
```

5.2.2 eval / evaltt

The eval filter evaluates the block as template text, processing any directives embedded within it. This allows template variables to contain template fragments, or for some method to be provided for returning template fragments from an external source such as a database, which can then be processed in the template as required.

```
my $vars  = {
    fragment => "The cat sat on the [% place %]",
};
$tt->process($file, $vars);
    || die $tt->error( );
```

The following example:

```
[% fragment | eval %]
```

is therefore equivalent to:

```
The cat sat on the [% place %]
```

The evaltt filter is provided as an alias for eval.

5.2.3 format(fmt)

The format filter takes a *sprintf*-style format string and applies it to the input, line by line. It can be used to preface blocks with comment markers, truncate lines, or do numeric conversions.

The *format* filter can be used for commenting out sections of text, as shown in Example 5-3.

Example 5-6. The format filter used to comment out code

```
[% FILTER format("<!-- %s -->") -%]

<script language="VBScript" type="text/vbscript">

   // evil vbscript here...

</script>

[% END %]
```

Example 5-3 produces the following output:

```
<!--     <script language="VBScript" type="text/vbscript"> -->

<!--        // evil vbscript here... -->

<!--     </script> -->
```

Because format passes its arguments to sprintf, any sprintf format strings can be used, including the field width and padding modifiers, as shown in Example 5-4.

Example 5-7. Left- and right-justified text

```
[% string = "Hello, I must be going." %]

Space padded, right justified: '[% string | format("% 32s") %]'

Space padded, left justified: '[% string | format("%- 32s") %]'
```

Example 5-4 produces the following output:

```
Space padded, right justified: '         Hello, I must be going.'

Space padded, left justified: 'Hello, I must be going.         '
```

The format filter also handles numerical transformations. Example 5-5 shows the same number being displayed in a number of different formats.

Example 5-8. Number formats

```
[% num = 42 %]

Unfiltered: [% num %]

Decimal: [% num | format("%d") %]

Binary: [% num | format("%b") %]

Hex: [% num | format("%x") %]

Hex, 0x-padded: [% num | format("%#x") %]

Octal: [% num | format("%o") %]

Octal, 0-padded: [% num | format("%#o") %]

Floating point: [% num | format("%f") %]

Scientific Notation: [% num | format("%e") %]
```

Example 5-5 produces the following output:

```
Unfiltered: 42

Decimal: 42

Binary: 101010

Hex: 2a

Hex, 0x-padded: 0x2a

Octal: 52
```

```
Octal, 0-padded: 052

Floating point: 42.000000

Scientific Notation: 4.200000e+01
```

Example 5-6 demonstrates the use of the %f format definition to control the number of decimal places displayed by a floating-point number.

Example 5-9. Controlling the number of decimal places

```
[% pi = 3.1415926536 %]

[% pi | format('%3.1f') %]

[% pi | format('%4.2f') %]

[% pi | format('%5.3f') %]
```

Its output is shown here:

```
3.1

3.14

3.142
```

Example 5-7 shows that variable interpolation works as you'd expect.

Example 5-10. Variable interpolation in format definitions

```
[% pi = 3.1415926536 %]

[% FOREACH dp = [ 1 .. 10 ] -%]

[% pi | format("%.${dp}f") %]

[% END %]
```

Here is its output:

```
3.1

3.14

3.142

3.1416

3.14159

3.141593

3.1415927

3.14159265

3.141592654

3.1415926536
```

In this example, the { } around dp is required so that the Template Toolkit knows to interpolate dp and not dpf, which is undefined (at least from the earlier snippet).

5.2.4 html

The html filter does very basic HTML encoding: it replaces the most commonly troublesome characters (<, >, &, and ") with their encoded counterparts. This is enough for many encoding jobs, and this filter is very lightweight. More complex encoding will need to use the html_entity filter, which implements a more general-purpose and extended encoding filter, but which is slower and more involved. Example 5-8 shows this filter in action.

Example 5-11. Using the html filter

```
<p>Creating an HTML anchor is simple:</p>

<pre>

[% FILTER html %]

<a href="http://www.template-toolkit.org/docs/">

  Read the documentation!

</a>

[% END %]

</pre>
```

The output from Example 5-8 is as follows:

```
<p>Creating an HTML anchor is simple:</p>

<pre>


 &lt;a href=&quot;http://www.template-toolkit.org/docs/&quot;&gt;

    Read the documentation!

 &lt;/a&gt;


</pre>
```

5.2.5 html_break / html_para_break

The html_break filter looks for sequences of two or more newlines in the text and replaces them with the HTML tag sequence

 (see Example 5-9).

Example 5-12. Using the html_break filter

```
[% FILTER html_break %]

The cat sat on the mat.


Mary had a little lamb.

[% END %]
```

This example outputs the following:

```
The cat sat on the mat.

<br />

<br />

Mary had a little lamb.
```

5.2.6 html_entity

The html filter is fast and simple, but it doesn't encode the full range of HTML entities that your text may contain. The html_entity filter uses the Apache::Util module if it can be loaded (it is written in C and is therefore faster) or the HTML::Entities module (written in Perl but equally as comprehensive) to perform the encoding. If the *Apache::Util* or the *HTML::Entities* module is installed on your system, the text will be encoded (via the escape_html or encode_entities subroutines, respectively) to convert all extended characters into their appropriate HTML entities (e.g., converting é to é). If neither module is available on your system, an html_entity exception will be thrown reporting an appropriate message.

Example 5-10 gives one example of a character that is converted to an HTML entity by this filter. The British £ symbol is converted to £.

Example 5-13. Using the html_entity filter

```
[% price = '£19.99' -%]
<p>
  The book cost [% price | html_entity %].
</p>
```

Example 5-10 produces the following output:

```
<p>
  The book cost £19.99.
</p>
```

For further information on HTML entity encoding, see http://www.w3.org/TR/REC-html40/sgml/entities.html.

5.2.7 html_line_break

The html_line_break filter replaces any newlines with
 HTML tags, thus preserving the line breaks of the original text in the HTML output. Example 5-11 shows its use.

Example 5-14. Using the html_line_break filter

```
[% FILTER html_line_break -%]
The cat sat on the mat.
Mary had a little lamb.
[% END %]
```

The example produces the following output:

```
The cat sat on the mat.<br />
Mary had a little lamb.<br />
```

5.2.8 html_para

The html_para filter formats a block of text into HTML paragraphs. A sequence of two or more newlines is used as the delimiter for paragraphs, which are then wrapped in HTML <p> ... </p> tags (see Example 5-12).

Example 5-15. Using the html_para filter

```
[% FILTER html_para -%]
The cat sat on the mat.

Mary had a little lamb.
[% END %]
```

This example produces the following output:

```
<p>
The cat sat on the mat.
</p>

<p>
Mary had a little lamb.
</p>
```

5.2.9 indent(pad)

The indent filter prefixes each line of input with a fixed string or number of spaces (defaults to four). If the supplied argument is a number, then that many spaces are used; otherwise it is taken to be a string and used literally.

This filter can be used to create bulleted lists, as shown in Example 5-13.

Example 5-16. Creating bullet points with the indent filter

```
[% FILTER indent("  * ") -%]
Item one
Item two
Item three
[%- END %]
```

Example 5-13 produces the following output:

```
* Item one
* Item two
* Item three
```

This filter also can be used to quote emails, as shown in Example 5-14.

Example 5-17. Quoting emails with the indent filter

```
[% quote = "> " %]
[% FILTER indent(quote) -%]
Dear Darren, Dave, and Andy,

You guys rock. The Template Toolkit book is fantastic.

Thanks for writing it.

A Fan
[% END %]
```

Example 5-14 produces the following output:

```
> Dear Darren, Dave, and Andy,
>
> You guys rock. The Template Toolkit book is fantastic.
```

```
>

> Thanks for writing it.

>

> A Fan
```

It also can be used to add a prefix to debugging messages, as shown in Example 5-15.

Example 5-18. Adding the template name to debug output

```
[% debug_msg | indent("[$template.name] ") | stderr %]
```

This example produces the following output:

```
[src/header] Some useful debug info (which goes to stderr)
```

If you give the indent filter no arguments, it indents by four spaces, as shown in Example 5-16.

Example 5-19. Default indent

```
[% FILTER indent -%]

A sample piece of text

that will be indented

[%- END %]

This isn't indented
```

Example 5-16 produces the following output:

```
    A sample piece of text

    that will be indented

This isn't indented
```

5.2.10 latex(outputType)

The latex filter passes the text block to LaTeX[2] and produces either PDF, DVI, or PostScript output. The outputType argument determines the output format, and it should be set to one of the following strings: "pdf" (default), "dvi", or "ps".

[2] If you have it installed on your system.

The text block should be a complete LaTeX source file. Example 5-17 shows the latex filter in action.

Example 5-20. Using the latex filter

```
[% FILTER latex("pdf") -%]

\documentclass{article}


\begin{document}


\title{A Sample TT2 \LaTeX\ Source File}

\author{Craig Barratt}

\maketitle
```

```
\section{Introduction}

This is some text.


\end{document}

[% END -%]
```

The output will be a PDF file. You should be careful not to prepend or append any extraneous characters or text outside the FILTER block because this text will wrap the (binary) output of the latex filter. Notice the - character placed before the %] end tag to remove the trailing newline.

One instance in which you might prepend text is in a CGI script, where you might include the Content-Type before the latex output, as shown in Example 5-18.

Example 5-21. Using the latex filter in a CGI program

```
Content-Type: application/pdf


[% FILTER latex("pdf") -%]

\documentclass{article}

\begin{document}

...

\end{document}

[% END -%]
```

In other cases, you might use the redirect filter to put the output into a file, rather than delivering it to STDOUT. This might be suitable for batch scripts, as shown in Example 5-19.

Example 5-22. Redirecting output from the latex filter

```
[% output = FILTER latex("pdf") -%]

\documentclass{article}

\begin{document}

...

\end{document}

[% END; output | redirect("document.pdf", 1) -%]
```

(Notice the second argument to redirect to force binary mode.)

The *latex* filter runs one or two external programs, so it isn't very fast. But for modest documents, the performance is adequate, even for interactive applications.

An error of type latex will be thrown if an error is reported by *latex*, *pdflatex*, or *dvips*.

5.2.11 lcfirst

The lcfirst filter folds the first character of the input to lowercase, as shown in Example 5-20.

Example 5-23. Using the lcfirst filter

```
[% "FIREHOSE" FILTER lcfirst %]
```

Example 5-20 produces the following output:

```
fIREHOSE
```

The lcfirst filter can be chained to the upper, ucfirst, and lower filters (described later in this chapter). In Example 5-21 the first letter of the sentence is folded to uppercase, with the remaining letters folded to lowercase.

Example 5-24. Combining the lower and ucfirst filters

```
[% sentence = "sOmE tExT iN rAnDoM cAsE" -%]

[% sentence | lower | ucfirst %]
```

Example 5-21 produces the following output:

```
Some text in random case
```

This sequence of filters would make a very useful macro, as shown in Example 5-22.

Example 5-25. The sentence_case macro

```
[% MACRO sentence_case(str) str | lower | ucfirst %]
```

The upper, lower, ucfirst, and lcfirst filters are subject to Perl's normal locale considerations. The perllocale documentation, which came with your copy of Perl, has all the details.

5.2.12 lower

The lower filter folds all the characters in the input text to lowercase (see Example 5-23).

Example 5-26. Using the lower filter

```
[% "Hello World" | lower %]
```

Example 5-23 produces the following output:

```
hello world
```

5.2.13 null

The null filter prints nothing. This is useful for plugins whose methods return values that you don't want to appear in the output. You can use CALL on each plugin method call to ignore the value returned, or you can wrap the block in a *null* filter (see Example 5-24).

Example 5-27. Using the null filter

```
[% FILTER null;

  USE im = GD.Image(100,100);

  black = im.colorAllocate(0,   0, 0);

  red   = im.colorAllocate(255,0,  0);

  blue  = im.colorAllocate(0,  0,  255);

  im.arc(50,50,95,75,0,360,blue);

  im.fill(50,50,red);

  im.png | stdout(1);

END;

-%]
```

Notice the use of the stdout filter to ensure that a particular expression generates output to STDOUT (in this case, in binary mode).

5.2.14 perl / evalperl

The perl filter evaluates the block as Perl code. The EVAL_PERL option must be set to a true value or a perl exception will be thrown (see Example 5-25).

Example 5-28. Using the perl filter

```
[% my_perl_code | perl %]
```

In most cases, the PERL ... END directive block should suffice for evaluating Perl code. Thus, Example 5-25 could have been written in the more verbose forms shown in Example 5-27.

Example 5-29. Using a PERL block in place of the perl filter

```
[% PERL %]
[% my_perl_code %]
[% END %]
```

Example 5-30. Using the perl filter in block form

```
[% FILTER perl %]
[% my_perl_code %]
[% END %]
```

The evalperl filter is provided as an alias for perl for backward compatibility.

5.2.15 redirect(file, options)

The redirect filter redirects the output of the block to the named file, relative to a location defined in the OUTPUT_PATH configuration option.

The redirect( ) filter will throw a file exception if the file specified cannot be opened. The filter should be used in a TRY ... CATCH block if you want to trap these kind of errors (see Example 5-28).

Example 5-31. Using the redirect filter

```
[% USE translate("src" = "en");
 FOREACH language = languages;
  file = "index.html.$language";
  TRY;
   text | $translate("dest" => language) | redirect(file);
   msg  = " + Successfully translated $file to $language.";
  CATCH file;
   msg = " - Cannot open $file: $error";
  CATCH;
   msg = " - Error: $error";
  END;
  emsg | stderr;
 END;
%]
```

5.2.16 remove(string)

The remove filter removes parts of the text block, based on the regular expression specified by the string. The regular expression is passed directly to Perl, and can contain anything regular Perl regexes can contain. Example 5-29 removes every occurence of the letter "e" from a string:

Example 5-32. Using the remove filter

```
[% string = "Hello, I must be going.";

  string | remove("e") %]
```

Example 5-29 produces the following output:

```
Hllo, I must b going.
```

Example 5-30 shows a more complex example that removes all occurences of "e" preceeded by an "H" and followed by "ll", but without removing the "H" or "ll". It combines a zero-width positive lookbehind assertion (?<=) with a zero-width positive lookahead assertion (?=).

Example 5-33. Using the remove filter with a regular expression

```
[% string = "Hello, I must be going.";

  string | remove("(?x)   # whitespace is not important

    (?<=H)              # an 'H'

      e                 # strip the 'e'!

    (?=ll)              # followed by 'll'

") %]
```

Example 5-30 produces the following output:

```
Hllo, I must be going.
```

5.2.17 repeat(iterations)

The repeat filter repeats the text iteration number of times. The default for iterations is 1 and the text is printed only once (see Example 5-31).

Example 5-34. Using the repeat filter

```
[% FILTER repeat(5) %]

All work and no play make Jack a dull boy.

[% END %]
```

Example 5-31 produces the following output:

```
All work and no play make Jack a dull boy.


All work and no play make Jack a dull boy.


All work and no play make Jack a dull boy.


All work and no play make Jack a dull boy.
```

```
All work and no play make Jack a dull boy.
```

5.2.18 replace(search, replace)

The replace filter is similar to the remove filter, but also takes a replacement string. Example 5-32 replaces every "e" in the input text with a "u".

Example 5-35. Using the replace filter

```
[% string = "Hello, I must be going.";

   string | replace("e", "u") %]
```

Example 5-32 produces the following output:

```
Hullo, I must bu going.
```

5.2.19 stderr

The stderr filter, shown in Example 5-33, prints the input text to STDERR. The binmode argument can be used as described in the stdout filter, explained next.

Example 5-36. Using the stderr filter

```
[% PROCESS something/cool | stderr(binmode=1) %]
```

5.2.20 stdout(options)

The stdout filter prints the output generated by the enclosing block to STDOUT. Currently, the only supported option is binmode, which can be passed as either a named parameter or a single argument to set STDOUT to binary mode (see Example 5-34).

Example 5-37. Using the stdout filter

```
[% PROCESS something/cool

   FILTER stdout(binmode=1) # recommended %]



[% PROCESS something/cool

   FILTER stdout(1)        # alternate %]
```

Setting binmode is mostly of use for Win32 and VMS users; see the *perlfunc(1)* manpage for all the gory details.

The stdout filter can be used to force binmode on STDOUT, or inside redirect, null, or stderr blocks to make sure that particular output goes to standard output. See Example 5-24 earlier in this chapter for an example of this usage.

5.2.21 trim

The trim filter removes any leading and trailing whitespace from the input text. Example 5-35 shows a string with leading and trailing whitespace, both of which are removed when passed through the trim filter.

Example 5-38. Using the trim filter

```
[% text = "  some text with leading and trailing spaces   " %]

+[% text | trim %]+
```

Example 5-35 produces the following output:

+some text with leading and trailing spaces+

This filter is particularly useful when working with BLOCK definitions. In Example 5-36, the foo block will be defined as *\nLine 1 of foo\n*. The surrounding newlines will also be introduced whenever the template is loaded using INCLUDE or PROCESS.

Example 5-39. Extra newlines when processing blocks

```
[% BLOCK foo %]

between

[% END %]



before-[% PROCESS foo %]-after
```

Example 5-36 produces the following output:

```
before-

between

-after
```

When run through the trim filter, leading and trailing newlines (which count as whitespace) will be removed from the output of the BLOCK (see Example 5-37).

Example 5-40. Using the trim filter to remove the extra newlines

```
[% BLOCK foo %]

between

[% END %]



before-[% PROCESS foo | trim %]-after
```

Example 5-37 produces the following output:

```
before-between-after
```

5.2.22 truncate(length)

The truncate filter returns the first length characters of the input text. The default value for length is 32. The text will actually be truncated three characters short of this, to make room for an ellipsis (. . .) to be appended to it. The returned text will be exactly length characters long, or less.

Example 5-38 shows it being used in a search results page.

Example 5-41. Using the truncate filter

```
[% FOREACH result = results %]

  * [% result.description | truncate(24) %]

    <a href="[% result.link %]">Read more</a>

[% END %]
```

When using truncate from within HTML, there is a danger that simply truncating the text will leave hanging HTML tags, as demonstrated in Example 5-39.

Example 5-42. Hanging HTML tags

```
[%- result.description = "Hello, <blink>world</blink>!" %]

Description: [% result.description | truncate(20) %]
```

Example 5-39 produces the following output:

```
Description: Hello, <blink>wor...
```

Using the remove filter in conjunction with the truncate filter, we get the desired results, as shown in Example 5-40.

Example 5-43. Using the remove filter to fix the hanging HTML tags

```
[%- result.description = "Hello, <blink>world</blink>!" %]

Description: [% result.description | remove("<[^>]*?>") | truncate(20) %]
```

Example 5-40 produces the following output:

```
Description: Hello, world!
```

5.2.23 ucfirst

The ucfirst filter folds the first character of the input to uppercase, as shown in Example 5-41.

Example 5-44. Using the ucfirst filter

```
[% "hello" | ucfirst %]
```

Example 5-41 produces the following output:

```
Hello
```

5.2.24 upper

The upper filter uppercases the input, similar to Perl's uc function (see Example 5-42).

Example 5-45. Using the upper filter

```
[% 'do not leave it is not real' | upper %]
```

Example 5-42 produces the following output:

```
DO NOT LEAVE IT IS NOT REAL
```

5.2.25 uri

The uri filter performs *URI-escaping*, which is the transformation of a URI string into a specific set of characters that are guaranteed not to cause any clients to do funny things. As defined by RFC 2396, a URI may consist of a limited number of "safe" characters; all others must be escaped using hexadecimal equivalents in the format %nn, where nn is the hex number that represents the ASCII code for the character. This is demonstrated in Example 5-43.

Example 5-46. Using the uri filter

```
[% filename = 'C:\My Documents\My Web Page.html' %]

Visit <a href="[% filename | uri %]">My Web Page</a>!
```

Example 5-43 produces the following output:

```
Visit <a href="C:%5CMy%20Documents%5CMy%20Web%20Page.html">My Web Page</a>!
```

Escaping a URI that doesn't need it cannot hurt, although escaping a URI that has already been escaped can lead to bugs that are difficult to track down. For example, the % character by itself is always escaped because it marks the beginning of an escaped sequence. Because an escaped URI is not necessarily HTML-safe, many URIs will also need to be passed through the html filter. A good rule of thumb is to escape anything that might need escaping immediately, as shown in Example 5-44.

Example 5-47. Using the uri filter with the html filter

```
[% url = "this page.cgi";
   prev = "$url?page=1&search=1" | uri | html;
   next = "$url?page=3&search=1" | uri | html;
%]
<a href="[% prev %]">Previous</a>
<a href="[% next %]">Next</a>
```

Example 5-44 produces the following output:

```
<a href="this%20page.cgi?page=1&amp;search=1">Previous</a>
<a href="this%20page.cgi?page=3&amp;search=1">Next</a>
```

For more information about URI escaping, see RFC 2396 and 2732.

Chapter 6. Plugins

A templating system that allow only minimal interaction with the outside world would become boring pretty quickly—most of the interesting stuff is going to be outside our templates, not inside. This chapter covers the Template Toolkit plugin system, designed to make interfacing with the outside world as simple as possible.

In the Template Toolkit, a *plugin* provides extra functionality that is otherwise not possible using only the core language. Many plugins create template-facing interfaces between external resources, such as a database or mail server, while some plugins provide tidy interfaces for complex formatting operations. Plugins allow developers to add functionality without having to modify or override core Template Toolkit components.

To a large extent, plugins are what give the Template Toolkit its power and flexibility: if the basic toolkit lacks the functionality you desire, it is very straightforward to add the functionality by creating plugins. External modules, designed without the Template Toolkit in mind, can be subverted for use within templates with just a little glue code. At the same time, however, a plugin can be used to enforce privacy within a module, and to make methods inaccessible, ensuring that the modules get used only as anticipated.

Unlike filters, which exist primarily to postprocess text, a plugin is unlimited in scope. The most popular use for plugins is to integrate other Perl modules—many, if not most, of the thousands of modules found on CPAN can be wrapped in a plugin and made available to a template designer.

6.1 Using Plugins

As we saw in Chapter 2, using plugins from a template is done with the USE directive:

```
[% USE date %]
```

This makes a date plugin object available to the template, which can be used by referencing the variable date. Many plugins accept arguments as part of the USE directive, to control the initial configuration. For example, to tell the date plugin to use GMT as the default time zone, instead of the local time zone, you would use:

```
[% USE date(gmt = 1) %]
```

Once a plugin has been initialized, it can be treated like any other variable:

```
Today is [% date.format %].
```

The preceeding example might return:

```
Today is 09:31:55 11-Aug-2003.
```

A plugin reference can be optionally assigned to a variable:

```
[% USE today = date %]
```

and accessed as today, rather than date. This has the potential to make for less confusing templates, but, more importantly, it means that you can have multiple instances of a plugin in the same template:

```
[% USE here = Directory '.' %]

[% USE there = Directory '/etc' %]
```

The Template Toolkit ships with a large number of useful, general-purpose plugins, which we will examine here, and provides a supporting framework for creating your own plugins (see Chapter 8).

Many of the standard plugins are Template Toolkit wrappers around general-purpose modules. In order to use these plugins, the wrapped module must be installed. The general installation techniques discussed in this chapter are applicable for all CPAN modules; in particular, the CPAN shell is very useful, as it will decline to reinstall modules that are up-do-date, and can be used to automatically fetch new versions from your favorite CPAN mirror.

In addition to the standard plugins, a number of plugins are available on CPAN, at http://www.cpan.org/modules/by-module/Template.

6.2 Standard Template Toolkit Plugins

As of Version 2.10, the Template Toolkit ships with a large number of plugins. The functionality these plugins add varies from trivial helper wrappers to full-blown reformatting utilities.

Some of these plugins are of interest only to developers, such as the Template::Plugin::Procedural and Template::Plugin::Filter; these will not be covered here (see Chapter 8 for treatment of these).

6.2.1 Autoformat

The Autoformat plugin provides an interface to Damian Conway's Text::Autoformat Perl module, which provides advanced text wrapping and formatting. Text::Autoformat is designed to be intelligent about wrapping lines; in addition to doing basic text wrapping, it can handle unusual text, such as mail or news text with quoting, or text with bullets or numbering. The Autoformat plugin provides a simple plugin/filter interface to the module.

Configuration options may be passed to the plugin constructor via the USE directive:

```
[% USE autoformat %]
```

The Autoformat plugin can then be called like a function, passing in text items that will be wrapped and formatted according to the current configuration (see Example 6-1).

Example 6-1. Autoformatting a Martin Gardner quote

```
[% USE autoformat right = 42 %]
[% autoformat('
Biographical history, as taught in our public schools, is still
largely a history of boneheads:  ridiculous kings and queens, paranoid
political leaders, compulsive voyagers, ignorant generals -- the
flotsam and jetsam of historical currents.  The men who radically
altered history, the great scientists and mathematicians, are seldom
mentioned, if at all.


    -- Martin Gardner
')
%]
```

Output of Example 6-1:

```
Biographical history, as taught in our
public schools, is still largely a history
of boneheads: ridiculous kings and queens,
paranoid political leaders, compulsive
voyagers, ignorant generals -- the flotsam
and jetsam of historical currents. The men
who radically altered history, the great
```

```
scientists and mathematicians, are seldom

mentioned, if at all.


    -- Martin Gardner
```

Additional configuration items can be passed to the autoformat subroutine and will be merged with any existing configuration specified via the constructor.

In addition to the functional interface, the Autoformat plugin also provides a filter interface, which works identically, as shown in Example 6-2.

Example 6-2. Using autoformat in filter mode

```
[% FILTER autoformat justify = 'center' -%]

Programming is a Dark Art, and it will always be. The programmer is

fighting against the two most destructive forces in the universe:

entropy and human stupidity. They're not things you can always

overcome with a "methodology" or on a schedule.


    -- Damian Conway

[% END %]
```

Output of Example 6-2:

```
      Programming is a Dark Art, and it will

       always be. The programmer is fighting

    against the two most destructive forces in

    the universe: entropy and human stupidity.

    They're not things you can always overcome

      with a "methodology" or on a schedule.


                  -- Damian Conway
```

Configuration options are passed directly to Text::Autoformat; see the Text::Autoformat documentation for all of the available options.

The Text::Autoformat module is available from CPAN at http://search.cpan.org/dist/Text-Autoformat/.

6.2.2 CGI

The CGI plugin is a wrapper around Lincoln Stein's CGI module, which is included with Perl. CGI provides a simple way of interacting with form parameters and cookies without having to understand the messy details of the CGI interface.

The CGI plugin provides access to all of CGI's functionality, including parameter and cookie support, access to file uploads, and access to HTML generation methods.

All the usual methods of the CGI module are available when using the CGI plugin, including the ever-popular param:

```
[% USE q = CGI %]


Hello, [% q.param('name') OR 'Mr. Unnamed' %]!
```

When called without an argument, param returns a list of all the defined parameter names, which can then be iterated over in a FOREACH loop:

```
[% FOREACH param IN q.param %]
   [% param %] -> [% q.param(param) %]
[% END %]
```

The plugin adds another method, params, that returns all CGI parameters as a hash:

```
[% params = q.params;
   IF params.exists('story_id');
      PROCESS story id = params.story_id;
   END;
%]
```

This hash can be used like any other hash. For example, to import this hash so that the parameters can be accessed directly, use import:[1]

[1] This takes advantage of the fact that the *stash* is a hash; see Chapter 8 for an explanation of why this works.

```
[% USE q = CGI('uid=18&name=Dave+Cross&nick=davorg') %]
[% params = q.params %]
[% import(params) %]


UID: [% uid %]
Nick: [% nick %]
Name: [% name %]
```

Without calling import, these variables would have to be qualified:

```
UID: [% params.uid %]
Nick: [% params.nick %]
Name: [% params.name # or q.param('name') -- same thing %]
```

Cookies are available via the aptly named cookie method:

```
[% SessionID = q.cookie('SessionID') %]
```

The CGI module's HTML generation methods work as expected, for the most part:

```
[% q.start_ol;
  FOREACH param IN q.param;
    q.start_li;
    q.start_b;
    param;
    q.end_b;
    ": ";
    q.param(param);
    q.end_li;
  END;
    q.end_ol;
%]
```

CGI methods that return a list, such as checkbox_group, need to be explicitly joined into a string (using the join vmethod, for example), or iterated over (using a FOREACH loop). Otherwise, the unsightly (and most likely unintended!) stringified array reference will be the result, as shown in Example 6-3.

Example 6-3. Stringified array

```
[% USE  q = CGI %]

[% q.checkbox_group(name = 'modules'

                 label = 'Modules to install'

                 values = [ 'Template-Toolkit',

                            'DBD::Google',

                            'Calendar::Simple'

                          ])

%]
```

Output of Example 6-3:

```
ARRAY(0x859eab4)
```

When joined with the join vmethod, the results are a little more natural, as shown in Example 6-4.

Example 6-4. Joined array

```
[% USE q = CGI %]

[% q.checkbox_group(name = 'modules'

                 label = 'Modules to install'

                 values = [ 'Template-Toolkit',

                            'DBD::Google',

                            'Calendar::Simple'

                          ]).join("\n")

%]
```

Output of Example 6-4:

```
<input type="checkbox" name="modules" value="Template-Toolkit" label="Modules to install" />Template-Toolkit

<input type="checkbox" name="modules" value="DBD::Google" label="Modules to install" />DBD::Google

<input type="checkbox" name="modules" value="Calendar::Simple" label="Modules to install" />Calendar::Simple
```

The CGI module is available with all recent versions of Perl, or from CPAN at http://search.cpan.org/dist/CGI/.

6.2.3 Datafile

The Datafile plugin provides a simple interface to tabular file-based data, such as Comma Separated Value (CSV) files. It provides a simple facility to construct a list of hashes, each of which represents a data record of known structure, from the datafile.

Pass a file to USE:

```
[% USE datafile(filename, delim = ':') %]
```

The file specified by filename will be read and split on delim into an array of hashes. delim is optional, and defaults to :. Currently, no INCLUDE_PATH search is performed for the file, so an absolute path should be used (this may change in a future version of the plugin, however).

delim can be used to specify an alternate delimiter character, such as the Tab or comma keys:

```
[% USE machines = datafile('machine-list.txt', delim = ",") %]
```

The format of the file is intentionally simple. The first line defines the field names, delimited by $delim with optional surrounding whitespace. Subsequent lines then define records containing data items, also delimited by $delim.

The first line of the file contains the field definitions. Blank lines and lines beginning with the comment character (#) will be ignored.

Each line is read, split into composite fields, and then used to initialize a hash array containing the field names as relevant keys.

The Datafile plugin is ideal for mostly static data that may need to be reused in many places—for example, storing information about computers, as shown in the following datafile called *machine-list.txt*:

```
name,    os,          ip
apollo,  RedHat 7.3,  10.100.5.100
hera,    RedHat 7.2,  10.100.33.227
juno,    Solaris 8,   10.100.6.41
artemis, RedHat 7.3,  10.100.6.42
hermes,  Solaris 9,   10.100.55.182
zeus,    RedHat 7.3,  10.100.6.78
```

Creating reports from this datafile is very simple, as Example 6-5 shows.

Example 6-5. Turning machine-list.txt into XML

```
[% USE machines = datafile('example/machine-list.txt',
                           delim = ',') -%]
<machines>
[% FOREACH machine IN machines.sort('name') -%]
  <machine name="[% machine.name %]"
           os="[% machine.os %]"
           ip="[% machine.ip %]" />
[% END -%]
</machines>
```

When Example 6-5 is run, we get:

```
<machines>
  <machine name="apollo"
           os="RedHat 7.3"
           ip="10.100.5.100" />
  <machine name="artemis"
           os="RedHat 7.3"
           ip="10.100.6.42" />
  <machine name="hera"
           os="RedHat 7.2"
           ip="10.100.33.227" />
  <machine name="hermes"
           os="Solaris 9"
           ip="10.100.55.182" />
  <machine name="juno"
           os="Solaris 8"
```

```
        ip="10.100.6.41" />
  <machine name="zeus"
        os="RedHat 7.3"
        ip="10.100.6.78" />
</machines>
```

6.2.4 Date

The Date plugin provides an easy way to manipulate dates and times, including generating formatted dates and times based on the formats defined by your system's strftime library (see the sidebar). The Date plugin also provides methods to perform date calculations using Date::Calc, and to perform general date manipulations using Date::Manip. (These modules, which are available from CPAN, must be installed in order to use this functionality. The rest of the plugin will work just fine without them, though.)

strftime

strftime is a system library function that returns a formatted date according to a *format string*. These format strings are a sort of templating system on their own—they contain plain text and format strings (which begin with %). These format strings are like the Template Toolkit's variables, and are replaced with the appropriate values. The supported format strings vary from system to system, but they all support the same basic subset, a summary of which follows:

%a The abbreviated weekday name.

%A The full weekday name.

%b The abbreviated month name.

%B The full month name.

%d The day of the month as a decimal number (range 01 to 31).

%H The hour as a decimal number using a 24-hour clock (range 00 to 23).

%I The hour as a decimal number using a 12-hour clock (range 01 to 12).

%j The day of the year as a decimal number (range 001 to 366).

%m The month as a decimal number (range 01 to 12).

%M The minute as a decimal number (range 00 to 59).

```
%p Either "AM" or "PM" according to the given time value,

      or the corresponding  strings  for  the current locale.

Noon is treated as "pm" and midnight as "am".

%S The second as a decimal number (range 00 to 59).

%w The day of the week as a decimal, range 0 to 6, Sunday being 0.

%Y The year as a decimal number, including the century.

%Z     The time zone, name, or abbreviation.
```

The plugin provides the format method, which accepts a time value, a format string, and a locale name. All of these parameters are optional with the current system time, default format (%H:%M:%S %d-%b-%Y), and current locale being used, respectively, if undefined. Default values for the time, format, and/or locale may be specified as named parameters in the USE directive:

```
[% USE date(format = '%Y/%m/%d'

          locale = 'fr_FR')

%]
```

When called without any parameters, the format method returns a string representing the current system time, formatted by strftime according to the default format and for the default locale (which may not be the current one, if locale is set in the USE directive):

```
[% date.format %]
```

The plugin allows a time/date to be specified as seconds since the epoch, as is returned by time:

```
File last modified: [% date.format(template.modtime) %]
```

The time/date can also be specified as a string of the form h:m:s d/m/y. A space or any of the characters :, /, or -, may be used to delimit fields:

```
[% USE day = date(format = '%A' locale = 'en_GB') %]

[% day.format('09:31:56 11-08-2003') %]
```

The previous code generates the following output:

```
Monday
```

A format string can also be passed to the format method, and a locale specification may follow that:

```
[% date.format(template.modtime, '%d-%b-%Y') %]

[% date.format(template.modtime, '%d-%b-%Y', 'en_GB') %]
```

A fourth parameter allows you to force output in GMT, in the case of seconds-since-the-epoch input:

```
[% date.format(template.modtime, '%d-%b-%Y', 'en_GB', 1) %]
```

Any or all of these parameters may be named. Positional parameters should always be in the order ($time, $format, $locale):

```
[% date.format(format = '%H:%M:%S') %]

[% date.format(time = template.modtime format = '%H:%M:%S') %]

[% date.format(mytime format = '%H:%M:%S') %]

[% date.format(mytime format = '%H:%M:%S' locale = 'fr_FR') %]

[% date.format(mytime format = '%H:%M:%S' gmt = 1) %]
```

The now method returns the current system time in seconds since the epoch:

```
[% date.format(date.now, '%A') %]


It has been [% date.now - template.modtime %] seconds since

[% template.name %] was last modified.
```

The calc method can be used to create an interface to the Date::Calc module (if installed on your system):

```
[% calc = date.calc %]

[% calc.Monday_of_Week(22, 2001).join('/') %]
```

Date::Calc provides a number of useful date-related methods, including date math (adding dates together, for example).

The manip method can be used to create an interface to the Date::Manip module (if installed on your system):

```
[% USE q = CGI %]

[% manip = date.manip %]

[% time = manip.UnixDate(q.param('date'), "%s") %]

[% date.format(time) %]
```

See the strftime sidebar for details about common format strings. Many versions of strftime, most notably GNU strftime, include more format strings, so check your system's manpages for the complete story.

Date::Calc is available from CPAN at http://search.cpan.org/dist/Date-Calc/. Date::Manip is also available from CPAN, at http://search.cpan.org/dist/Date-Manip/.

6.2.5 Directory

The Directory plugin provides a simple interface to a directory and the files within it. It provides methods for iterating over all the contained files and subdirectories. This plugin is in cahoots with the File plugin, and in fact uses instances of the File plugin to represent files within a directory (all the methods available to the File plugin are also available here, such as uid and mtime). Subdirectories within a directory are represented by further instances of this plugin.

The Directory plugin can be used to create an instance with a directory name as an argument:

```
[% USE dir = Directory '/tmp' %]
```

It then provides access to the files and subdirectories contained within the directory via the files and dirs methods, respectively:

```
# regular files (not directories)

[% FOREACH file = dir.files %]

   [% file.name %]
```

```
[% END %]


# directories only

[% FOREACH file = dir.dirs %]

   [% file.name %]

[% END %]


# files and/or directories

[% FOREACH file = dir.list %]

   [% file.name %] ([% file.isdir ? 'directory' : 'file' %])

[% END %]
```

The plugin constructor will throw a Directory error if the specified path does not exist or is not a directory, or if there is some other error at the operating system level (such as NFS problems). Otherwise, it will scan the directory and create lists named files, containing files, dirs, containing directories, and list, containing both files and directories combined. The nostat option can be set to disable all file/directory checks and directory scanning; this speeds up the process of loading the plugin for large directories:

```
[% USE etc = directory '/etc/' nostat = 1 %]
```

Each file in the directory will be represented by an instance of the File plugin, and each directory will be represented by another Directory plugin. If the recurse flag is set, those directories will contain further nested entries, and so on. With the recurse flag unset, as it is by default, each is just a place marker for the directory and does not contain any further content unless its scan method is explicitly called. The isdir flag can be tested against files and/or directories, returning true if the item is a directory or false if it is a regular file:

```
[% FOREACH file = dir.list %]

   [% IF file.isdir %]

     * Directory: [% file.name %]

   [% ELSE %]

     * File: [% file.name %]

   [% END %]

[% END %]
```

6.2.6 DBI

The DBI plugin provides a template-level interface to Tim Bunce's DBI module. The DBI module provides a uniform database interface, and the DBI plugin ensures that it plays nicely with the Template Toolkit. The DBI plugin is covered extensively in Chapter 9.

6.2.7 Dumper

The Dumper plugin provides an interface to the Data::Dumper module. Data::Dumper will convert a complex variable into a human-readable structure.

The Dumper plugin provides the dump method, which is extremely useful for displaying the structure of a variable (see Example 6-6).

Example 6-6. Dumping a hash

```
[% USE dumper %]

[% terms = {

      sass  = 'know, be aware of, meet, have sex with'
```

```
        hoopy = 'really together guy'

        frood = 'really, amazingly together guy'

    } %]

[% dumper.dump(terms) %]
```

Coming out, terms looks almost exactly like it did going in, except for the order:[2]

[2] Perl's hashes are not stored in the order in which they are inserted, but rather in an order optimized for fast lookup by name. This is called *hash order*, and Data::Dumper doesn't attempt to reorder the keys of a hash as it dumps them.

```
$VAR1 = {

          'hoopy' => 'really together guy',

          'frood' => 'really, amazingly together guy',

          'sass' => 'know, be aware of, meet, have sex with'

        };
```

Although the Dumper plugin is not so useful for a variable we've defined ourselves, it is much more useful for data structures that you don't have direct control over, as Example 6-7 shows.

Example 6-7. Dumping the CGI plugin

```
[% USE CGI %]

[% USE dumper %]

[% dumper.dump(CGI) %]
```

Output of Example 6-7:

```
$VAR1 = bless( {

                 '.charset' => 'ISO-8859-1',

                 '.parameters' => [  ],

                 '.fieldnames' => {  },

                 'escape' => 1

               }, 'CGI' );
```

The dump_html method takes the output of dump and formats it for HTML. Example 6-7 is the same as Example 6-8, except for the call to dump_html:

Example 6-8. Dumping the CGI plugin with dump_html

```
[% USE CGI %]

[% USE dumper %]

[% dumper.dump_html(CGI) %]
```

The output is very similar:

```
$VAR1 = bless( {<br>

                 '.charset' =&gt; 'ISO-8859-1',<br>

                 '.parameters' =&gt; [  ],<br>

                 '.fieldnames' =&gt; {  },<br>

                 'escape' =&gt; 1<br>

               }, 'CGI' );<br>
```

The Data::Dumper Pad, Indent, and Varname options are supported as constructor arguments to affect the output generated. Example 6-9 shows all the details.

Example 6-9. Modifying Data::Dumper's output

```
[% USE CGI %]

[% USE dumper(Pad = '// ', Varname = 'CGI') %]

[% dumper.dump(CGI) %]
```

Output of Example 6-9:

```
// $CGI1 = bless( {

//              '.charset' => 'ISO-8859-1',

//              '.parameters' => [  ],

//              '.fieldnames' => {  },

//              'escape' => 1

//            }, 'CGI' );
```

Data::Dumper comes with all recent versions of Perl, and is also available from CPAN at http://search.cpan.org/dist/Data-Dumper/.

6.2.8 File

This plugin provides an abstraction of a file. It can be used to fetch details about files from the filesystem, or to represent abstract files (e.g., when creating an index page) that may or may not exist on a filesystem.

A filename or path should be specified as a constructor argument:

```
[% USE file 'foo.html' %]

[% USE file 'foo/bar/baz.html' %]

[% USE file '/foo/bar/baz.html' nostat = 1 %]
```

The file should exist on the current filesystem (unless the nostat option is set, which we discuss in a bit) as an absolute file when specified with a leading / as per */foo/bar/baz.html*, or otherwise as one relative to the current working directory. The constructor performs a stat on the file and makes the 13 elements returned available as the plugin items:

```
dev ino mode nlink uid gid rdev size

atime mtime ctime blksize blocks
```

For example:

```
[% USE baz = File '/foo/bar/baz.html' %]



[% baz.mtime %]

[% baz.mode %]
```

In addition, the user and group items are set to contain the user and group names as returned by calls to getpwuid and getgrgid for the file uid and gid elements, respectively (see Example 6-10). On Win32 platforms on which getpwuid and getgrid are not available, these values are undefined.

Example 6-10. user and uid

```
[% USE Makefile = file 'Makefile'  %]

uid: [% Makefile.uid %]

user: [% Makefile.user %]
```

Output of Example 6-10:

```
uid: 500

user: darren
```

This user/group lookup can be disabled by setting the noid option, as shown in Example 6-11.

Example 6-11. noid = 1

```
[% USE Makefile = file 'Makefile' noid = 1 %]

uid: [% Makefile.uid %]

user: [% Makefile.user %]
```

Output of Example 6-11:

```
uid: 500

user:
```

If the stat on the file fails (e.g., file doesn't exist, bad permission, etc.), the constructor will throw a File exception. This can be caught within a TRY...CATCH block:

```
[% TRY %]

   [% USE File '/tmp/myfile'  %]

   File exists!

[% CATCH File %]

   File error: [% error.info %]

[% END %]
```

Note the capitalization of the exception type, File, to indicate an error thrown by the File plugin, to distinguish it from a regular file exception thrown by the Template Toolkit. Like all plugins, the File plugin can be referenced by the lowercase name file; exceptions are always thrown of the File type, regardless of the capitalization of the plugin name used.

The nostat option can be specified to prevent the plugin constructor from performing a stat on the file specified. In this case, the file does not have to exist in the filesystem, no attempt will be made to verify that it does, and no error will be thrown if it doesn't. The entries for the items usually returned by stat will be set empty.

```
[% USE file '/some/where/over/the/rainbow.html', nostat = 1 %]

[% file.mtime %]     # nothing
```

All File plugins, regardless of the *nostat* option, have set a number of items relating to the original path specified:

path

The full, original file path specified to the constructor.

```
[% USE file '/foo/bar.html' %]

[% file.path %]     # /foo/bar.html
```

name

The name of the file without any leading directories.

```
[% USE file '/foo/bar.html' %]

[% file.name %]     # bar.html
```

dir

The directory element of the path with the filename removed.

```
[% USE file '/foo/bar.html' %]

[% file.name %]     # /foo
```

ext

The file extension, if any, appearing at the end of the path following a dot operator (.) (not included in the extension).

```
[% USE file '/foo/bar.html' %]

[% file.ext %]      # html
```

home

This contains a string of the form *../..* to represent the upward path from a file to its root directory.

```
[% USE file 'bar.html' %]

[% file.home %]     # nothing

[% USE file 'foo/bar.html' %]

[% file.home %]     # ..

[% USE file 'foo/bar/baz.html' %]

[% file.home %]     # ../..
```

root

The root item can be specified as a constructor argument, indicating a root directory in which the named file resides. This is otherwise set empty.

```
[% USE file 'foo/bar.html', root='/tmp' %]

[% file.root %]     # /tmp
```

abs

This returns the absolute file path by constructing a path from the root and path options.

```
[% USE file 'foo/bar.html', root='/tmp' %]

[% file.path %]     # foo/bar.html

[% file.root %]     # /tmp

[% file.abs %]      # /tmp/foo/bar.html
```

In addition, the following method is provided:

rel(path)

This returns a relative path from the current file to another path specified as an argument. It is constructed by appending the path to the home item.

```
[% USE file 'foo/bar/baz.html' %]

[% file.rel('wiz/waz.html') %]      # ../../wiz/waz.html
```

6.2.9 Format

The Format plugin provides a simple way to format text according to a specific format. The *format* is a text string, and can contain regular text interspersed with sprintf-style placeholders (the format string is passed to Perl's sprintf). Each %*x* token will be replaced with successive elements of the list provided to the function call. This plugin is very similar to the *format* filter, described in Chapter 5.

USE format creates a functionlike variable that can be used for formatting. Example 6-12 shows a simple way to wrap text in HTML comments.

Example 6-12. HTML comments

```
[% USE commented = format('<!-- %s -->') -%]

[% commented('The cat sat on the mat') %]
```

Output of Example 6-12:

```
<!-- The cat sat on the mat -->
```

Mutiple elements can be included as well, by passing multiple items. Format tokens of %s will be treated as strings, but tokens of %d will be treated as numbers, as shown in Example 6-13.

Example 6-13. image tag

```
[% USE img = format('<img src="%s" width="%d" height="%d" />') -%]

[% img('logo.png', '0088', 42) %]
```

Output of Example 6-13:

```
<img src="logo.png" width="88" height="42" />
```

All of the formatting rules and tricks that apply to the *format* filter also apply to the Format plugin. See Chapter 5 for some more examples.

As with the *format* filter, width, precision, and minimum and maximum lengths can be provided as part of the filter, as Example 6-14 shows.

Example 6-14. Using precision and width with format

```
[% USE fmt = format("%2.8f");

  USE Math;

  fmt(Math.pi)

%]
```

Output of Example 6-14:

```
3.14159265
```

6.2.10 GD

Lincoln Stein's GD modules provide access to the gd graphics library. gd is a small, fast graphics library that allows you to create color drawings using a large number of graphics primitives, and emits the drawings in a number of popular graphics formats, such as PNG or JPEG.

In the following example, a new image is created with the USE call. The plugin's contructor takes the same arguments as GD::Image itself:

```
[% USE img1 = GD.Image            # empty image of default size (64x64) %]

[% USE img2 = GD.Image(X, Y)       # empty image (X x Y) %]

[% USE img3 = GD.Image(filename)    # a preexisting image %]
```

To use an existing image, use the filename form of the constructor. The GD plugin will attempt to determine the type of image based on the first few bytes of the file, and then Do The Right Thing.

Once you have an image object, you can call methods on it. Colors are allocated using the colorAllocate method, which accepts a (red, green, blue) triplet as integers:

```
[% orange = img.colorAllocate(255, 165, 0) %]

[% red = img.colorAllocate(255, 0, 0) %]

[% blue = img.colorAllocate(0, 0, 255) %]
```

The first color allocated becomes the background color,[3] so choose wisely!

[3] There are plenty of example colors in your system's *rgb.txt.*

The getPixel method is used in conjunction with the rgb method to return the color of a particular pixel.[4]

[4] GD stores images in a bitmapped form internally; getPixel returns the index into the color table of the color at the specified pixel, and the rgb method turns that back into a triplet.

To get the color at pixel (42,24), you could use this:

```
[% index = img.getPixel(42, 42);

  rgb = img.rgb(index)

%]
```

Or, more succinctly:

```
[% rgb = img.getPixel(42, 42).rgb(index) %]
```

GD supports several output types, including PNG, JPEG, WBMP, and its own GD and GD2 formats. You are likely to use only PNG and JPEG on a regular basis, though the GD2 format is useful for storing images that will be manipulated primarily by GD.

Here are the GD.Image output methods:

```
[% img.png      # emit the image as a PNG... %]

[% img.jpeg     # ... or as a JPEG... %]

[% img.gd       # ... or in GD %]

[% img.gd2      # ... or GD2 formats %]
```

When combined with the OUTPUT_PATH and redirect filter, the GD plugins can be used to automate image creation.

Because these plugins are used to create binary output, it is very important that no extraneous template output appear before or after the image. Because some methods return values that would otherwise appear in the output, it is recommended that this plugin code be wrapped in a null filter. The methods that produce the final output (e.g., *png*, *jpeg*, *gd*, etc.) can then explicitly make their output appear by using the *stdout* filter, with a non-zero argument to force binary mode (see Example 6-15).

Example 6-15. Strange, pointless shapes made entirely with GD

```
[%  FILTER null;

    USE im   = GD.Image(100, 100);

    USE c    = GD.Constants;

    USE poly = GD.Polygon;


    # allocate some colors; white is the background

    white  = im.colorAllocate(255, 255, 255);

    black  = im.colorAllocate(0, 0, 0);

    orange = im.colorAllocate(255, 165, 0);

    blue   = im.colorAllocate(0, 0, 255);


    # Put a black-bordered orange square in the middle

    im.filledRectangle(10, 10, 90, 90, orange);

    im.rectangle(10, 10, 90, 90, black);


    # Draw a diamond in the middle

    poly.addPt(0, 50);
```

```
   poly.addPt(50, 100);
   poly.addPt(100, 50);
   poly.addPt(50, 0);
   im.filledPolygon(poly, blue);

   # Put a smaller black-bordered white square in the middle of that
   im.filledRectangle(30, 30, 70, 70, white);
   im.rectangle(30, 30, 70, 70, black);

   # Output binary image in PNG format
   im.png | stdout(1);
   END;
-%]
```

The GD.Constants plugin provides templates with access to the many GD constants that define font types, styles, and other image attributes.

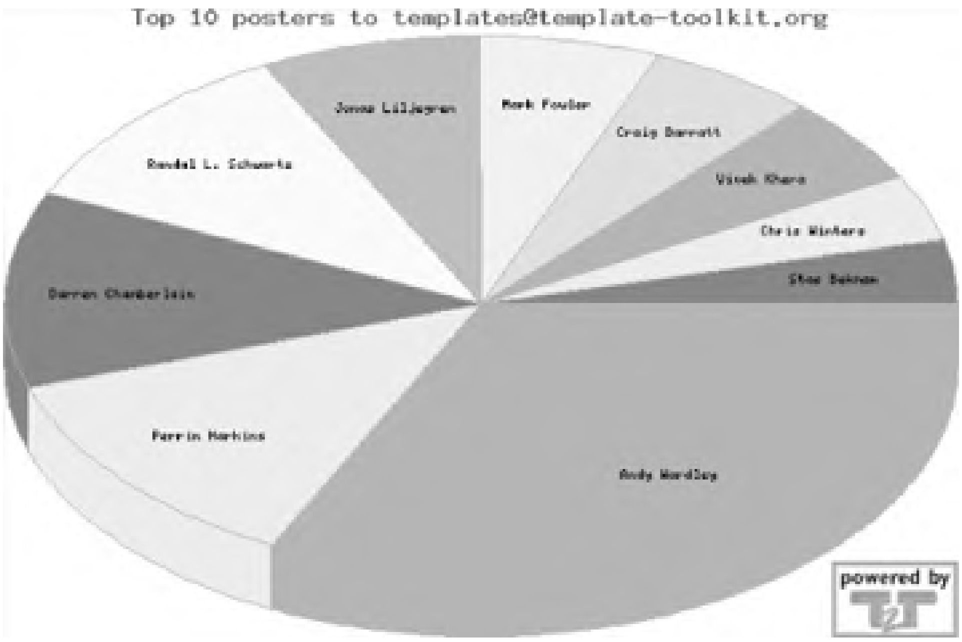
The GD.Graph plugins provide an interface to Martien Verbruggen's GD::Graph module. This module is built on top of GD and can generate graphs, plots, and charts.

The GD.Graph plugins are actually a group of several smaller plugins: GD.Graph.area, GD.Graph.bars, GD.Graph.bard3d, GD.Graph.lines, GD.Graph.lines3d, GD.Graph.linespoints, GD.Graph.mixed, GD.Graph.pie, GD.Graph.pie3d, and GD.Graph.points. All of the plugins have the same interface and differ mainly in the accepted arguments; see the GD::Graph documentation for a full API guide, including the differences between the types.

Figure 6-1 shows a pie chart generated from a datafile containing the top 10 posters to the Template Toolkit mailing list, generated from the single large *mbox* file that the mailman maintains.[5]

[5] At http://www.template-toolkit.org/pipermail/templates.mbox/templates.mbox.

Figure 6-1. Top 10 posters



This graph was generated using the simple template in Example 6-16.

Example 6-16. Generating a graph of the top 10 posters

```
[%
FILTER null;
   USE graph = GD.Graph.pie(600, 400);
   USE gdc = GD.Constants;
   USE posters = datafile('posters');

   data = [
      [ ]     # posters
      [ ]     # the count
   ];
   list = 'templates@template-toolkit.org';

   FOREACH poster IN posters;
      data.0.push(poster.name);
      data.1.push(poster.posts);
   END;

   graph.set(title       = "Top 10 posters to $list"
            transparent = 0
            logo        = 'tt2power.png'
            t_margin    = 4
            b_margin    = 4
            r_margin    = 4
            l_margin    = 4
            start_angle = -90 # aesthetics hack
   );

   # A big font for the title
   graph.set_title_font(gdc.gdGiantFont);

   graph.plot(data).png | stdout(1);
  END;
-%]
```

The GD.Text, GD.Text.Align, and GD.Text.Wrap plugins provide interfaces to the GD::Text module. GD::Text provides a font-independant way of dealing with text in GD and the GD plugins. This is useful primarily for aligning text on GD.Image objects; because positioning strings needs to be done based on pixel offsets, GD.Text's get('width') and get('height') functionality can be invaluable.

Here's an example of using GD.Text:

```
[%
   FILTER null;
   msg = 'Template Toolkit';
   USE gdc = GD.Constants;
   USE t = GD.Text(text = msg
                   font = gdc.gdGiantFont);

   width = t.get('width');     # width of the string in pixels
   height = t.get('height');   # height of the string in pixels

   imgwidth = width * 3;
   imgheight = height * 3;

   USE img = GD.Image(imgwidth, imgheight);
   black = img.colorAllocate(0, 0, 0);
   orange = img.colorAllocate(255, 165, 0);

   img.string(gdc.gdGiantFont, width, height, msg, orange);

   img.png | stdout(1);
END;
-%]
```

The GD module is available on CPAN at http://search.cpan.org/dist/GD/, and the underlying gd C library lives at http://www.boutell.com/gd/. The GD::Graph module is available on CPAN at http://search.cpan.org/dist/GD-Graph/, and the GD::Text module is available on CPAN at http://search.cpan.org/dist/GD-Text/.

6.2.11 HTML

The HTML plugin provides a simple mechanism for generating arbitrary HTML elements. It also provides utility methods for creating attribute lists and for HTML- and URL-escaping.

Generating a single element is done with the element method, as shown in Example 6-17.

Example 6-17. Generating a single element

```
[% USE HTML %]
[% HTML.element('html') %]
```

Output of Example 6-17:

```
<html>
```

Not very exciting. Any named parameters provided become attribute pairs, as shown in Example 6-18.

Example 6-18. Generating an element with attributes

```
[% USE HTML %]

[% HTML.element('img',

            src = 'logo.png'

            width = 88

            height = 38

            alt = 'Company Logo'

            name = 'logo')

%]
```

Output of Example 6-18:

```
<img alt="Company Logo" height="38" src="logo.png" width="88" name="logo">
```

If the plugin is used with the sorted option set, then the attributes are sorted in alphabetical order when the attribute list is produced, as shown in Example 6-19.

Example 6-19. Generating an element with sorted attributes

```
[% USE HTML(sorted=1) %]

[% HTML.element('img',

            src = 'logo.png'

            width = 88

            height = 38

            alt = 'Company Logo'

            name = 'logo')

%]
```

Output of Example 6-19:

```
<img alt="Company Logo" height="38" name="logo" src="logo.png" width="88">
```

The plugin also provides HTML- and URL-escaping utility methods, which can be used independently of the plugin's element generating methods. The escape method does HTML-escaping, as shown in Example 6-20.

Example 6-20. Using escape

```
[% USE HTML %]

[% HTML.escape("I'd prefer that you type that tag as <br />") %]
```

Output of Example 6-20:

```
I'd prefer that you type that tag as &lt;br /&gt;
```

And the url method does URL-escaping, as shown in Example 6-21.

Example 6-21. Using url

```
[% USE HTML %]

[% HTML.url("I'd prefer that you type that tag as <br />") %]
```

Output of Example 6-21:

```
I%27d%20prefer%20that%20you%20type%20that%20tag%20as%20%3Cbr%20%2F%3E
```

As Example 6-22 shows, the attribute method can be used to generate an attribute string from a hash (this method is used internally by element).

Example 6-22. Generating a string of attributes from a hash

```
[% USE HTML(sorted=1);
  attr = {
    type = 'submit'
    name = 'search'
    value = 'Go!'
  };
  HTML.attributes(attr);
%]
```

Output of Example 6-22:

```
name="search" type="submit" value="Go!"
```

6.2.12 Image

The Image plugin provides a wrapper for image files. This plugin makes available the wrapped image's size, type, and modification time as methods, and also provides methods for generating an HTML tag for the image:

```
[% USE image 'tt2power.png' %]
```

The Image plugin will use either the Image::Info or Image::Size modules (both are available from CPAN), or will throw a runtime error if neither is present on the system. Image::Info is used in preference to Image::Size because it provides more information about the image.

Regardless of which underlying module is used, the name, height, width, and modification time of the image will be available (see Examples Example 6-23 and Example 6-24).

Example 6-23. Basic image info

```
[% image.name %] was last modified on [% date.format(image.modtime) %].
```

Output of Example 6-23:

```
tt2power.png was last modified on 09:29:02 11-Aug-2003.
```

Example 6-24. Image height and width

```
Height: [% image.height %]
Width: [% image.width %]
```

Output of Example 6-24:

```
Height: 47
Width: 78
```

In addition, if the plugin uses Image::Info, several more methods are available, including the following:

file_media_type

Returns the media type in major/minor form and produces the following output:

```
Content-type: image/png
```

file_ext

Returns the extension of the image file and produces the following output:

png

resolution

The value of this field normally gives the physical size of the image on screen or paper. When the unit specifier is missing, this field denotes the squareness of pixels in the image.

The syntax of this field is:

```
<res> <unit>
```

```
<xres> "/" <yres> <unit>
```

```
<xres> "/" <yres>
```

The <res>, <xres>, and <yres> fields are numbers. The <unit> is a string such as dpi, dpm, or dpcm (denoting "dots per inch/meter/cm).

The previous example produces the following output:

```
Resolution: 1/1
```

In addition, several other attributes are available when using Image::Info that depend on the image type; for example, animated gifs have a GIF_loop attribute.

The Image plugin has two utility methods: attr, which returns the image's height and width as XHTML attributes; and tag, which returns a formatted XHTML string representing the image. For instance, this code:

```
[% image.attr %]
```

would produce this output:

```
width="78" height="47"
```

The tag method creates a full XHTML tag, with attributes (using the attr method). For instance, this code:

```
[% image.tag %]
```

produces this output:

```
<img src="tt2power.png" width="78" height="47" />
```

The tag method can also take arbitrary named parameters, and will Do The Right Thing with them:

```
[% image.tag(alt = 'Powered by TT', name = 'tt2power') %]
```

The previous code would output the following:

```
<img src="tt2power.png" width="78" height="47" alt="Powered by TT" name="tt2power" />
```

6.2.13 Iterator

The Iterator plugin provides a way to create a Template::Iterator object to iterate over a data set. An *iterator* is used for walking through the elements of a list; one is created automatically by the *FOREACH* directive and is aliased to the loop variable.

This plugin allows an iterator to be explicitly created with a given name, or with the default plugin name, iterator. Example 6-25 shows how to create your own iterator.

Example 6-25. Creating your own iterator

```
[% USE iterator(list) %]


[% FOREACH item IN iterator %]

   [% '<list>' IF iterator.first %]

   <item>[% item %]</item>

   [% '</list>' IF iterator.last %]

[% END %]
```

The Iterator plugin is useful when you want to use a portion of a list in a FOREACH loop, rather than the entire list, as shown in Example 6-26.

Example 6-26. Iterating over part of a list

```
[% days = [ 'Sunday'   'Monday' 'Tuesday' 'Wednesday'
            'Thursday' 'Friday' 'Saturday'
          ] %]
[% USE weekdays = iterator(days.slice(1,5)) %]
[% FOREACH weekday IN weekdays %]
  [% weekday %]
[% END %]
```

Because an iterator contains references to other objects and not copies of the objects themselves, this can be more efficient than simply creating a new list containing only the desired elements. This is especially when the list is large, true when it contains items other than simple data elements (such as objects), or when generating the data is expensive (as when generating database queries). So, in Example 6-26, weekdays persists beyond the FOREACH loop shown and can be reused.

Unlike the transient iterators created within FOREACH loops, specifically created iterators don't go out of scope at the end of their enclosing loop. This means that iterators can be reused. Example 6-27 illustrates this.

Example 6-27. Reusing iterators

```
[%  USE iterator([ 1 .. 3 ]);
    USE fmt = format("%02d => %02d/%02d\n");
    BLOCK iterate;
        fmt(i, it.count, it.size)
            FOREACH i IN it;
        "\n";
    END;
-%]

[% PROCESS iterate it = iterator FOREACH [ 1 .. 3 ] %]
```

Output of Example 6-27:

```
01 => 01/03
02 => 02/03
03 => 03/03

01 => 01/03
02 => 02/03
03 => 03/03

01 => 01/03
02 => 02/03
03 => 03/03
```

6.2.14 Pod

This plugin provides an interface to the Pod::POM module, which parses POD[6] documents into an internal object model that can then be traversed and presented through the Template Toolkit.

[6] POD, which stands for *Plain Old Documentation*, is Perl's internal documentation format. It is intentionally simple and extensible, and is designed to be readable without special processing.

You create a POD parser with USE:

```
[% USE pod %]
```

This parser can then be used to parse documents in POD format:

```
[% pom = pod.parse_file('Chapter6.pod') %]
```

Pod::POM presents POD documents as a tree, of which each branch represents successive =head1 tags in the document. =head2 elements form branches within these sections, and so on, down to the content nodes at the end. The Pod::POM documentation describes this *Pod Object Model* (that's what POM stands for) in great detail.

For more details on using the POD plugin, and on Pod::POM in general, please consult the Pod::POM documentation.

Pod::POM is available from CPAN at http://search.cpan.org/dist/Pod-POM/.

6.2.15 String

This is a plugin module for object-oriented string manipulation. A String object is created via the USE directive, adding any initial text value as an argument or as the named parameter text:

```
[% USE String %]

[% USE String 'initial text' %]

[% USE String text='initial text' %]
```

It's likely that there will be more than one string in a template, so assigning the plugin to a name is wise:

```
[% USE greeting = String 'Hello World' %]
```

Once you've got a String object, you can use it as a prototype to create other String objects with the new method:

```
[% USE String %]

[% greeting = String.new('Hello World') %]
```

The new method also accepts an initial text string as an argument or the named parameter text:

```
[% greeting = String.new( text => 'Hello World' ) %]
```

You can also call the copy method to create a new string as a copy of the original:

```
[% greet2 = greeting.copy %]
```

The String object has a text method to return the content of the string:

```
[% greeting.text %]
```

However, it is sufficient to simply print the string and let the overloaded stringification operator call the text method automatically for you:

```
[% greeting %]
```

Thus, you can treat String objects pretty much like any regular piece of text, interpolating it into other strings, for example:

```
[% msg = "It printed '$greeting' and then dumped core\n" %]
```

You also have the benefit of numerous other methods for manipulating the string:

```
[% msg.append("PS  Don't eat the yellow snow") %]
```

Note that all methods operate on and mutate the contents of the string itself. If you want to operate on a copy of the string, simply take a copy first:

```
[% msg.copy.append("PS  Don't eat the yellow snow") %]
```

These methods return a reference to the String object itself. This allows you to chain multiple methods together:

```
[% msg.copy.append('foo').right(72) %]
```

It also means that in the previous examples, the string is returned. This causes the text method to be called, which results in the new value of the string being printed. To suppress printing of the string, you can use the CALL directive:

```
[% foo = String.new('foo') %]


[% foo.append('bar') %]         # prints "foobar"


[% CALL foo.append('bar') %]    # nothing
```

There are several ways to create a new String object. Here is the "usual" way:

```
[% USE err = String text = 'Bad Things Happened' %]
```

Alternatively, calling the new method on an already initialized String object will create a new string:

```
[% msg = err.new('False alarm!') %]
```

Finally, copy will return a copy of the string object:

```
[% urgent_error = err.copy.append(' - lp1 on fire') %]
```

The plugin also implements many methods to inspect or modify the contents of the String object. Here is a list of the methods:

text

Returns the internal text value of the string. The stringification operator is overloaded to call this method. Thus, the following are equivalent:

```
[% msg.text %]

[% msg %]
```

length

Returns the length of the string.

```
[% USE String("foo") %]

[% String.length %]   # => 3
```

search($pattern)

Searches the string for the regular expression specified in $pattern, returning true if found, or returning false otherwise.

```
[% item = String.new('foo bar baz wiz waz woz') %]

[% item.search('wiz') ? 'WIZZY! :-)' : 'not wizzy :-(' %]
```

split($pattern, $limit)

Splits the string based on the delimiter $pattern and optional $limit. Delegates to Perl's internal split, so the parameters are exactly the same.

```
[% FOREACH item.split %]

    ...

[% END %]

[% FOREACH item.split('baz|waz') %]

    ...

[% END %]
```

The following methods modify the internal value of the string. For example:

```
[% USE str=String('foobar') %]


[% str.append('.html') %]       # str => 'foobar.html'
```

The value of the string str is now foobar.html. If you don't want to modify the string, simply take a copy first.

```
[% str.copy.append('.html') %]
```

These methods all return a reference to the String object itself. This has two important benefits. The first is that when used as shown earlier, the String object str returned by the append method will be stringified with a call to its text method. This will return the newly modified string content. In other words, a directive such as:

```
[% str.append('.html') %]
```

will update the string and also print the new value. If you just want to update the string but not print the new value, use CALL:

```
[% CALL str.append('.html') %]
```

The other benefit of these methods returning a reference to the string is that you can chain as many different method calls together as you like. For example:

```
[% String.append('.html').trim.format(href) %]
```

Here are the methods:

push($suffix, ...) / append($suffix, ...)

Appends all arguments to the end of the string. The append method is provided as an alias for push.

```
[% msg.push('foo', 'bar') %]

[% msg.append('foo', 'bar') %]
```

pop($suffix)

Removes the suffix passed as an argument from the end of the string.

```
[% USE String 'foo bar' %]

[% String.pop(' bar')   %]   # => 'foo'
```

unshift($prefix, ...) / prepend($prefix, ...)

Prepends all arguments to the beginning of the string. The prepend method is provided as an alias for unshift.

```
[% msg.unshift('foo ', 'bar ') %]

[% msg.prepend('foo ', 'bar ') %]
```

shift($prefix)

Removes the prefix passed as an argument from the start of the string.

```
[% USE String 'foo bar' %]

[% String.shift('foo ') %]   # => 'bar'
```

left($pad)

If the length of the string is less than $pad, the string is left-formatted and padded with spaces to $pad length.

```
[% msg.left(20) %]
```

right($pad)

As per left( ), but right-padding the string to a length of $pad.

```
[% msg.right(20) %]
```

center($pad) / centre($pad)

As per *left( )* and *right( )*, but formatting the string to be centered within a space-padded string of length $pad. The centre method is provided as an alias for center to account for misspellings.

```
[% msg.center(20) %]   # American spelling

[% msg.centre(20) %]   # European spelling
```

format($format)

Apply a format in the style of sprintf to the string.

```
[% USE String("world") %]

[% String.format("Hello %s\n") %]  # => "Hello World\n"
```

upper( )

Converts the string to uppercase.

```
[% USE String("foo") %]

[% String.upper %]  # => 'FOO'
```

lower( )

Converts the string to lowercase

```
[% USE String("FOO") %]

[% String.lower %]  # => 'foo'
```

capital( )

Converts the first character of the string to uppercase.

```
[% USE String("foo") %]

[% String.capital %]  # => 'Foo'
```

The remainder of the string is left untouched. To force the string to be all lowercase with only the first letter capitalized, you can do something like this:

```
[% USE String("FOO") %]

[% String.lower.capital %]  # => 'Foo'
```

chop( )

Removes the last character from the string:

```
[% USE String("foop") %]

[% String.chop %]       # => 'foo'
```

chomp( )

Removes the trailing newline from the string:

```
[% USE String("foo\n") %]

[% String.chomp %]       # => 'foo'
```

trim( )

Removes all leading and trailing whitespace from the string:

```
[% USE String("   foo   \n\n ") %]

[% String.trim %]       # => 'foo'
```

collapse( )

Removes all leading and trailing whitespace, and collapses any sequences of multiple whitespace to a single space:

```
[% USE String(" \n\r  \t  foo   \n \n bar  \n") %]

[% String.collapse %]   # => "foo bar"
```

truncate($length, $suffix)

Truncates the string to $length characters.

```
[% USE String('long string') %]

[% String.truncate(4) %]  # => 'long'
```

If $suffix is specified, it will be appended to the truncated string. In this case, the string will be further shortened by the length of the suffix to ensure that the newly constructed string, complete with suffix, is exactly $length characters long.

```
[% USE msg = String('Hello World') %]

[% msg.truncate(8, '...') %]   # => 'Hello...'
```

replace($search, $replace)

Replaces all occurrences of $search in the string with $replace.

```
[% USE String('foo bar foo baz') %]

[% String.replace('foo', 'wiz')  %]  # => 'wiz bar wiz baz'
```

remove($search)

Removes all occurrences of $search in the string.

```
[% USE String('foo bar foo baz') %]

[% String.remove('foo ')  %]  # => 'bar baz'
```

repeat($count)

Repeats the string $count times.

```
[% USE String('foo ') %]

[% String.repeat(3)  %]  # => 'foo foo foo '
```

6.2.16 Table

The Table plugin allows you to format a list of data items into a virtual table. When you create a Table plugin via the *USE* directive, simply pass a list reference as the first parameter and then specify a fixed number of rows or columns:

```
[% USE table list, rows = 5  %]
```

The plugin then presents a table-based view of the data set. The data isn't actually reorganized in any way, but is available via row, col, rows, and cols as if formatted into a simple two-dimensional table of n rows x n columns. Thus, if our sample alphabet list contained the letters a to z, the preceeding *USE* directives would create plugins that represent the views of the alphabet, as shown in Examples Example 6-28 and Example 6-29.

Example 6-28. rows

```
[% USE table alphabet, rows = 5 %]


[% FOREACH row IN table.row;
   FOREACH cell IN row;
      "$cell  ";
   END %]
[% END %]
```

Output of Example 6-28:

```
a f k p u z
b g l q v
c h m r w
d i n s x
e j o t y
```

Example 6-29. cols

```
[% USE table alphabet, cols = 5 %]


[% FOREACH col IN table.col;
   FOREACH cell IN col;
      "$cell  ";
   END %]
[% END %]
```

Output of Example 6-29:

```
a b c d e f
g h i j k l
m n o p q r
s t u v w x
y z
```

We can request a particular row or column using the row and col methods, as shown in Example 6-30.

Example 6-30. row(0)

```
[% USE table alphabet, rows = 5 %]
[% FOREACH item IN table.row(0);
   item %]
[% END %]
```

Output of Example 6-30:

```
a

f

k

p

u

z
```

Data in rows is returned from left to right, and in columns from top to bottom. The first row/column is 0. By default, rows or columns that contain empty values will be padded with the undefined value to fill it to the same size as all other rows or columns. For example, the last row (row 4) in the first example would contain the values [e j o t y undef]. The Template Toolkit will safely accept these undefined values and print an empty string. You can also use the *IF* directive to test whether the value is set.

You can explicitly disable the pad option when creating the plugin to returned shortened rows/columns where the data is empty, as shown in Example 6-31.

Example 6-31. pad = 0

```
[% USE table alphabet, cols=5, pad=0 %]

[% FOREACH item = table.col(4);

  item %]

[% END %]
```

The rows method returns all rows/columns in the table as a reference to a list of rows (themselves list references). The row method, when called without any arguments calls rows to return all rows in the table. cols and col behave analogously.

6.2.17 URL

The URL plugin provides a convenient way to construct URLs from a base stem and a hash of additional parameters, without having to worry about getting the syntax correct.

The constructor should be passed a base URL:

```
[% USE siteroot = url('http://www.template-toolkit.org') %]
```

The constructor can optionally be passed a hash reference of default parameters and values:

```
[% USE next = url('search.cgi', search = search, next = curpage + 1) %]
```

When the plugin is then called without any arguments, the default base and parameters are returned as a formatted URL, including any query parameters. Thus, one url object can be used as the base for another:

```
[% USE news = url("$siteroot/news") %]
```

Simply calling or interpolating the plugin is enough for the Template Toolkit to expand it, as shown in Example 6-32.

Example 6-32. url in action

```
[% USE tt = url('http://www.template-toolkit.org/') -%]

The <a href="[% tt %]">Template Toolkit</a> rules!
```

Output of Example 6-32:

```
The <a href="http://www.template-toolkit.org/">Template Toolkit</a> rules!
```

Any parameters passed into the call are combined with parameters specified when the plugin was created, and all become part of the resulting URL, as shown in Example 6-33.

Example 6-33. url + parameters

```
[% USE article = url('http://slashdot.org/article.pl'

                   mode = 'nested',

                   threshold = 1) %]

[% article(sid = 'xxx') %]
```

Output of Example 6-33:

```
http://slashdot.org/article.pl?mode=nested&amp;sid=xxx&amp;threshold=1
```

6.2.18 Wrap

The Wrap plugin provides a simple text wrapper, based on the Text::Wrap module. Paragraphs can be formatted using specific widths and leading indent, and can have padding applied to each line in the output.

The plugin defines a wrap subroutine that is called with the input text and further optional parameters to specify the page width (which defaults to 72) and tab characters for the first and subsequent lines (these have no defaults).

This plugin's simple wrapping is not aware of special prefixes and so forth; for more sophisticated wrapping, use the more complex autoformat plugin. For most simple wrapping jobs, however, wrap is capable enough (see Example 6-34).

Example 6-34. Basic wrapping

```
[% USE wrap %]


[% text = BLOCK -%]

First, attach the transmutex multiplier to the cross-wired quantum homogenizer.

[% END %]


[% wrap(text, 30) %]
```

Output of Example 6-34:

```
First, attach the transmutex

multiplier to the cross-wired

quantum homogenizer.
```

The plugin also registers a wrap filter that accepts the same three optional arguments, but takes the input text directly via the filter input (see Example 6-35).

Example 6-35. Wrap filter

```
[% FILTER bullet = wrap(40, '* ', '  ') -%]

First, attach the transmutex multiplier to the cross-wired quantum homogenizer.

[%- END %]


[% FILTER bullet -%]

Then remodulate the shield to match the harmonic frequency, taking care to correct the

phase difference.

[% END %]
```

Output of Example 6-35:

```
* First, attach the transmutex
  multiplier to the cross-wired quantum
  homogenizer.

* Then remodulate the shield to match
  the harmonic frequency, taking care
  to correct the phase difference.
```

Text::Wrap comes with recent versions of Perl, and is also available from CPAN at http://search.cpan.org/dist/Text-Wrap/.

6.2.19 XML::DOM

The XML::DOM plugin gives access to the XML Document Object Module via Clark Cooper and Enno Derksen's XML::DOM module. The following synopsis gives examples of some ways in which it can be used. See Chapter 10 for further details.

```
# load plugin
[% USE dom = XML.DOM %]

# also provide XML::Parser options
[% USE dom = XML.DOM(ProtocolEncoding => 'ISO-8859-1') %]

# parse an XML file
[% doc = dom.parse(filename) %]
[% doc = dom.parse(file => filename) %]

# parse XML text
[% doc = dom.parse(xmltext) %]
[% doc = dom.parse(text => xmltext) %]

# call any XML::DOM methods on document/element nodes
[% FOREACH node = doc.getElementsByTagName('report') %]
   * [% node.getAttribute('title') %]     # or just '[% node.title %]'
[% END %]

# define VIEW to present node(s)
[% VIEW report notfound='xmlstring' %]
   # handler block for a <report>...</report> element
   [% BLOCK report %]
      [% item.content(view) %]
   [% END %]
```

```
   # handler block for a <section title="...">...</section> element
   [% BLOCK section %]
   <h1>[% item.title %]</h1>
   [% item.content(view) %]
   [% END %]

   # default template block converts item to string representation
   [% BLOCK xmlstring; item.toString; END %]

   # block to generate simple text
   [% BLOCK text; item; END %]
[% END %]

# now present node (and children) via view
[% report.print(node) %]

# or print node content via view
[% node.content(report) %]
```

6.2.20 XML::RSS

The XML::RSS plugin is a simple interface to Jonathan Eisenzopf's XML::RSS module. A Rich Site Summary (RSS) file is typically used to store short news headlines describing different links within a site. This plugin allows you to parse RSS files and format the contents accordingly using templates.

```
[% USE news = XML.RSS(filename) %]

[% FOREACH item = news.items %]
   [% item.title %]
   [% item.link  %]
[% END %]
```

See Chapter 10 for more details.

6.2.21 XML::Style

This plugin defines a filter for performing simple stylesheet-based transformations of XML text.

Named parameters are used to define those XML elements that require transformation. These may be specified with the USE directive when the plugin is loaded and/or with the *FILTER* directive when the plugin is used.

This example shows how the default attributes border="0" and cellpadding="4" can be added to <table> elements:

```
[% USE xmlstyle
      table = {
         attributes = {
            border     = 0
```

```
            cellpadding = 4
          }
        }
%]

[% FILTER xmlstyle %]
<table>
  ...
</table>
[% END %]
```

This produces the output:

```
<table border="0" cellpadding="4">
  ...
</table>
```

Parameters specified within the *USE* directive are applied automatically each time the xmlstyle filter is used. Additional parameters passed to the *FILTER* directive apply only to that block.

```
[% USE xmlstyle
        table = {
          attributes = {
            border      = 0
            cellpadding = 4
          }
        }
%]

[% FILTER xmlstyle
        tr = {
          attributes = {
            valign="top"
          }
        }
%]
<table>
  <tr>
    ...
  </tr>
</table>
[% END %]
```

Of course, you may prefer to define your stylesheet structures once and simply reference them by name. Passing a hash reference of named parameters is just the same as specifying the named parameters as far as the Template Toolkit is concerned:

```
[% style_one = {
     table = { ... }
     tr    = { ... }
  }
  style_two = {
     table = { ... }
     td    = { ... }
  }
  style_three = {
     th = { ... }
     tv = { ... }
  }
%]

[% USE xmlstyle style_one %]

[% FILTER xmlstyle style_two %]
  # style_one and style_two applied here
[% END %]

[% FILTER xmlstyle style_three %]
  # style_one and style_three applied here
[% END %]
```

Any attributes defined within the source tags will override those specified in the stylesheet:

```
[% USE xmlstyle
      div = { attributes = { align = 'left' } }
%]

[% FILTER xmlstyle %]
<div>foo</div>
<div align="right">bar</div>
[% END %]
```

The output produced is:

```
<div align="left">foo</div>
<div align="right">bar</div>
```

The filter can also be used to change the element from one type to another:

```
[% FILTER xmlstyle
        th = {
            element = 'td'
```

```
        attributes = { bgcolor='red' }
    }
%]
<tr>
  <th>Heading</th>
</tr>
<tr>
  <td>Value</td>
</tr>
[% END %]
```

The output here is as follows (notice how the end tag </th> is changed to </td> as is the start tag):

```
<tr>
  <td bgcolor="red">Heading</td>
</tr>
<tr>
  <td>Value</td>
</tr>
```

You can also define text to be added immediately before or after the start or end tags. For example:

```
[% FILTER xmlstyle
    table = {
        pre_start = '<div align="center">'
        post_end  = '</div>'
    }
    th = {
        element    = 'td'
        attributes = { bgcolor='red' }
        post_start = '<b>'
        pre_end    = '</b>'
    }
%]
<table>
<tr>
  <th>Heading</th>
</tr>
<tr>
  <td>Value</td>
</tr>
</table>
[% END %]
```

The output produced is:

```
<div align="center">
<table>
<tr>
  <td bgcolor="red"><b>Heading</b></td>
</tr>
<tr>
  <td>Value</td>
</tr>
</table>
</div>
```

6.2.22 XML::XPath

The XML::XPath plugin provides an interface to Matt Sergeant's XML::XPath module. The following synopsis shows some examples of its use. See Chapter 10 and Chapter 11 for further examples of using this plugin.

```
[% USE xpath = XML.XPath(xmlfile) %]
[% USE xpath = XML.XPath(file => xmlfile) %]
[% USE xpath = XML.XPath(filename => xmlfile) %]

# load plugin and specify XML text to parse
[% USE xpath = XML.XPath(xmltext) %]
[% USE xpath = XML.XPath(xml => xmltext) %]
[% USE xpath = XML.XPath(text => xmltext) %]

# then call any XPath methods (see XML::XPath docs)
[% FOREACH page = xpath.findnodes('/html/body/page') %]
   [% page.getAttribute('title') %]
[% END %]

# define VIEW to present node(s)
[% VIEW repview notfound='xmlstring' %]
   # handler block for a <report>...</report> element
   [% BLOCK report %]
      [% item.content(view) %]
   [% END %]

   # handler block for a <section title="...">...</section> element
   [% BLOCK section %]
   <h1>[% item.getAttribute('title') | html %]</h1>
   [% item.content(view) %]
```

```
  [% END %]

  # default template block passes tags through and renders
  # out the children recursively
  [% BLOCK xmlstring;
     item.starttag; item.content(view); item.endtag;
  END %]

  # block to generate simple text
  [% BLOCK text; item | html; END %]
[% END %]

# now present node (and children) via view
[% repview.print(page) %]

# or print node content via view
[% page.content(repview) %]
```

Chapter 7. Anatomy of the Template Toolkit

Now that we've spent a great deal of time looking at what you can do with the Template Toolkit, let's take a look inside and get a feel for how it actually works. We'll follow the flow of processing a template from the frontend (such as Template or *ttree*), to getting the file from disk (Template::Provider), to compiling it (Template::Parser, Template::Grammar, and Template::Directive), and to executing it (Template::Context and Template::Document).

We'll be using pseudocode versions of the methods to illustrate the major thrust of each component, mainly to gloss over tedious details of error checking, parameter handling, file opening and closing, and syntax. Feel free to get a copy of each *.pm* file and follow along with the real code; however, the best way to understand any complex system is to look at the innards, and the Template Toolkit is no exception.

7.1 Template Modules

The Template module is simply a frontend that creates and uses a Template::Service object and then pipes the output wherever you want it to go (standard output by default, or maybe a file, scalar variable, etc.). The Apache::Template module is another frontend, which uses a Template::Service::Apache object under the hood and sends the output back to the relevant Apache object. The now-familiar *tpage* and *ttree* scripts are command line-based frontends; *tpage* simply connects standard input and output by way of the Template Toolkit, while *ttree* does the same for source and destination files (with the intelligence to detect when they haven't changed).

These frontend modules are really there only to handle any specifics of the environment in which they're being used. Apache::Template does web-specific things, such as making form parameters and client request headers available as template variables and allowing configuration via *httpd.conf*. The *ttree* program parses command-line arguments and a configuration file. The regular Template frontend deals with standard output and writing to files. Otherwise, it is Template::Service (or a subclass) that does all the work. The process method calls $service->process and then spends most of its time figuring out where to send the results. Example 7-1 shows the *process* method in action.

Example 7-1. Template::process

```
sub process($name, \%vars, $output, \%options) {
    $content = SERVICE->process($name, $vars);


    if type($output) = = 'code':
        &$output($content);


    elsif type($output) = = 'filehandle':
        print $output $content;


    elsif type($output) = = 'scalar reference':
        $$output = $content;


    elsif type($output) = = 'array reference':
        push @$output, $content;


    elsif $output->can('print'):
        $output->print($content);


    else:
        open OUT, $output;
        if $options->{'binmode'}:
            binmode OUT;
        print OUT $content;


}
```

Apache::Template behaves a little differently, but the basic idea is the same. Because it's an Apache handler, the entry point is called handler, not process (see Example 7-2).

Example 7-2. Apache::Template::handler

```
sub handler($r) {
    $template = SERVICE->template($r);
    $params = SERVICE->params($r);
    $content = SERVICE->process($r);

    SERVICE->headers($r, $template, $content);

    $r->print($content);

    return OK;
}
```

As you can see, the service object (Apache::Template uses a Template::Service::Apache instance, which is a Template::Service subclass) has a few more responsibilities: params and header handle the Apache-specific stuff (reading client headers and form parameters), and template calls upon a special provider to get a compiled template based on the filename requested (more on template later). Let's look at these modules in more detail.

7.1.1 Template::Service

The Template::Service module provides a consistent template-processing environment. In addition to processing the main template (passed by name to process), the service object processes any additional templates (PRE_PROCESS, PROCESS, POST_PROCESS), wrappers (WRAPPER), or error handlers (ERROR) defined by the frontend. For the most part, the job of the service object is really just one of scheduling, dispatching, and handling runtime errors.

Actually, that's a bit of a lie: the service object doesn't process the templates itself, but instead makes process calls against a Template::Context object. In pseudocode, process looks like the code shown in Example 7-3.

Example 7-3. Template::Service::process

```
sub process($template, \%vars) {
    $output = '';

    $compiled_template = CONTEXT->template($template);
    $vars->{'template'} = $compiled_template;

    eval {
        foreach $name in PRE_PROCESS:
            $output += CONTEXT->process($name, $vars);

        @process = PROCESS || $compiled_template;
        foreach $name in @process:
            $output += CONTEXT->process($name, $vars);

        @wrapper = reverse WRAPPER;
```

```
        foreach $name in @wrapper:
            $output += CONTEXT->process($name, $vars);

        foreach $name in POST_PROCESS:
            $output += CONTEXT->process($name, $vars);
    }

    if $EVAL_ERROR:
        $output = CONTEXT->process(ERROR);

    return $output;
}
```

7.1.2 Template::Context

Template::Context is the runtime engine for the Template Toolkit—the module that hangs everything together in the lower levels and that does most of the real work, albeit by crafty delegation to various other friendly helper modules.

Given a template name, the context's process method must first get a handle on the compiled template that represents that name. It does this by calling its template method.

Within template, the context calls fetch on each member of the list of Template::Provider objects (the contents of the LOAD_TEMPLATES array), stopping when one of them returns a Template::Document object. If none of them does, the context throws a Template::Exception object back to process via throw, as shown in Example 7-4.

Example 7-4. Template::Context::template

```
sub template($name) {
    $template = undef;

    foreach $p in LOAD_TEMPLATES:
        $template = $p->fetch($name);
        last if $template;

    $self->throw('file', "$name not found") unless $template;

    return $template;
}
```

The throw method takes an error type, such as file, and a descriptive string (*$name not found*), and creates a Template::Exception object out of them. This exception object is first passed back to the Template::Service object, which tries to handle it with any ERROR handlers the user specified; if that fails (i.e., if the user hasn't defined a handler for this exception type), it is passed into the template, where it is available via the error variable. Template::Context also implements a catch method, which attempts to handle a thrown error. The context's catch method ensures that the error caught is a Template::Exception rather than a simple string, and is primarily used within compiled templates. We'll see catch when we talk about Template::Directive and Template::Document.

Once the context has a compiled template, it updates the *stash* (the data engine where template variables are managed) to set any template variable definitions specified as the second argument by reference to a hash array.

Then, it calls the document's process method, passing a reference to itself (the context) as an argument. In doing this, it provides itself as an object against which template code can make callbacks to access runtime resources and Template Toolkit functionality: not only does the Template::Context object receive calls from the *outside* (those originating in user code calling the process method on a Template object), but it also receives calls from the *inside* (those originating in template directives of the form [% PROCESS template %]).

process looks something like the code shown in Example 7-5.

Example 7-5. Template::Context::process

```
sub process(\@names, \%vars) {

    foreach $name in $names:

        push @templates, $self->template($name);


    STASH->update($vars);


    eval {

        foreach $template in @templates:

            $output += &$template($self);

    }


    if $EVAL_ERROR:

        $self->throw($EVAL_ERROR);


    return $output;

}
```

As you can see, process can take an array of template names, so the following:

```
[% PROCESS copyright + footer %]
```

and:

```
$context->process([ 'copyright', 'footer' ]);
```

are equivalent.

The context is also responsible for loading plugins and filters via the cleverly named plugins and filters methods. The context maintains arrays of plugin and filter providers (stored in LOAD_PLUGINS and LOAD_FILTERS, respectively) that are consulted in order, until one of them returns the requested item. plugin is very similar to template, as you can see in Example 7-6.

Example 7-6. Template::Context::plugin

```
sub plugin($name, \@args) {

    $plugin = undef;


    foreach $p in @LOAD_PLUGINS:

        $plugin = $p->fetch($name, $args);

        last if $plugin;


    $self->throw('plugin', "$name not found") unless $plugin;
```

```
    return $plugin;
}
```

filter is slightly different; as shown in Example 7-7, the context can store filters in a local cache, if $alias is provided.

Example 7-7. Template::Context::filter

```
sub filter($name, \@args, $alias) {
    $filter = undef;

    $filter = $self->filter_cache->$name;
    return $filter if $filter;

    foreach $p in @LOAD_FILTERS:
        $filter = $p->fetch($name, $args);
        last if $filter;

    return undef unless $filter;

    $self->filter_cache->$alias = $filter;

    return $filter;
}
```

7.1.3 Template::Stash

The Template::Stash module defines the data engine that powers the Template Toolkit. The stash goes out of its way to ensure that all the data it contains can be accessed in the same way by making variable access "magical": scalars, arrays, hashes, subroutines, and objects are all accessed the same way, courtesy of the dot operator (.). We'll have a lot more to say about the stash shortly in Section 7.2.

7.1.4 Template::Provider

Template::Provider is responsible for locating templates, compiling them with Template::Parser, and handing Template::Document instances back to the context, all via the fetch method. The provider also handles the details of template caching and hides filesystem differences.

In pseudocode, fetch looks something like the code shown in Example 7-8.

Example 7-8. Template::Provider::fetch

```
sub fetch($name) {
    if $name =~ /^\//:
        if ABSOLUTE:
            $data, $error = $self->_fetch(name);
        else:
```

```
        $data = undef;

        $error = 'ABSOLUTE paths not allowed';


    elsif $name =~ /^\.+\//:

        if RELATIVE:

            $data, $error = $self->_fetch($name);

        else:

            $data = undef;

            $error = 'RELATIVE paths not allowed';


    else:

        $data, $error = $self->_fetch_path($name);


    return $data, $error;

}
```

There are two other helper methods here: _fetch and _fetch_path. The primary difference between the two is that _fetch is expecting a direct path to a file (either absolute or relative), while _fetch_path walks the INCLUDE_PATH to find the template. Each checks to see whether the user requested memory or disk-based caching, and uses these versions in preference to recompiling the template itself. If caching is enabled, the provider checks timestamps to ensure that the version on disk hasn't been modified since it was last compiled, and either hands back the cached version, or recompiles it and hands that back (being sure to cache this new version).

_fetch looks like Example 7-9 in pseudocode.

Example 7-9. Template::Provider::_fetch

```
sub _fetch($name) {

    $compiled_filename = $self->_compiled_filename;


    if CACHE_SIZE:

        $cached = $self->template_cache->$name

        if $cached:

            $self->_refresh($cached);

            $doc = $cached;

        else:

            $filedata = $self->_load($name);

            $doc = $self->_compile($filedata, $compiled_filename);

    else:

        if $compiled_filename:

            $doc = $self->_load_compiled($compiled_template);

            $self->store($name, $doc);

        else:

            $filedata = $self->_load($name);
```

```
        $doc = $self->_compile($filedata, $compiled_filename);

        $self->store($name, $doc);


    return $doc;

}
```

We're leaving out a lot of private methods here: _compiled_filename concatenates COMPILE_DIR, the template name, and COMPILE_EXT to figure out where a compiled template should be written to disk, and _refresh does timestamp comparisons between $name and $compiled_filename, calling _load and _compile as necessary. _load opens the file $name on disk and reads it into a scalar variable, and adds the special elements name and modtime to $filedata; these are $name and $name's timestamp (from (stat($name))[9]).

_compile bears a closer look because it is in _compile that the parser comes into play (see Example 7-10).

Example 7-10. Template::Provider::_compile

```
sub _compile($filedata, $compiled_filename) {

    $parsed = PARSER->parse($filedata->{'text'}, $filedata);


    $parsed->{'name'}    = $filedata->{'name'};

    $parsed->{'modtime'} = $filedata->{'time'};


    if $compiled_filename:

        DOCUMENT->write_perl_file($parsed, $compiled_filename);


    return DOCUMENT->new($parsed);

}
```

As mentioned earlier, Template::Provider objects are stored in an array; template iterates over these providers, giving each one a chance to respond. This means that it is possible to layer special-purpose providers (database-based, HTTP-based, and so on) on top of the default provider, or even instead of it.

Once the provider finds the template it is looking for, it passes the contents of the file to a Template::Parser, which tokenizes the templates, checks them for syntactical correctness, and returns a compiled data structure, which is fed to Template::Document.

7.1.5 Template::Parser

Template::Parser does most of the hard work. It accepts a string representation of a template, which it tokenizes based on the current TAGS settings, and uses a Template::Grammar instance to determine the actions associated with each token.

parse is the parser's primary interface, and looks something like the code in Example 7-11.

Example 7-11. Template::Parser::parse

```
sub parse($text, $info) {

    @tokens = $self->split_text($text);


    $block = $self->_parse(@tokens, $info);


    return {
```

```
        BLOCK     = $block
        DEFBLOCKS = $self->DEFBLOCK
        METADATA  = $self->METADATA
    }
}
```

split_text is the tokenizer. It uses START_TAG and END_TAG to break apart the text, and handles any whitespace-chomping specified by PRE_CHOMP, POST_CHOMP, or TRIM. _parse uses the grammar to determine whether the stream of tokens is syntactically valid, and if so, uses Template::Directive to generate Perl code ($block). DEFBLOCK and METADATA are accumulated in the parser as the document is parsed.

7.1.6 Template::Grammar

The Template::Grammar module contains a big list of parser states and their associated actions, which are generated from a yacc-like grammar using Parse::Yapp. The grammar calls upon the Template::Directive factory class to actually generate the code.

Ninety-nine percent of the grammar is generated from the file *parser/Parser.py* (part of the source distribution), which we'll see in more detail later in Chapter 8. The last 1% is part of the grammar skeleton, *parser/Grammar.pm.skel*, which defines reserved words and special tokens.

7.1.7 Template::Directive

The Template::Directive module defines the nitty-gritty details of the compilation process. The grammar calls a method against a Template::Directive instance (called a *factory*), passing along the tokens the parser found. The factory returns Perl code that implements the directives, which is evaled into live code by Template::Document.

By way of example, let's look at the code generated for an anonymous block, such as the one shown in Example 7-12.

Example 7-12. An example template

```
[% BLOCK %]
A city is like a large, complex rabbit.
[% END %]
```

This relatively simple block generates a bunch of code, as shown in Example 7-13.

Example 7-13. Code implementing an anonymous block

```
# BLOCK
$output .=  do {
    my $output  = '';
    my $error;

    eval { BLOCK: {
        $output .=  "\nA city is like a large, complex rabbit.\n";
    } };
    if ($@) {
        $error = $context->catch($@, \$output);
        die $error unless $error->type eq 'return';
```

```
    }

    $output;
};
```

The nested calls to eval are necessary because the user can do pretty much anything in a block, such as attempt to load nonexistent plugins or process a file with syntax errors, as shown in Example 7-14.

Example 7-14. A malformed template

```
[% BLOCK %]
[% USE %]
[% END %]
```

Template::Directive makes use of compile-time constants, as specified by the CONSTANTS configuration directive. When generating the code for *GET* directives, the factory checks to see whether any constants are defined, and if so, calls upon a Template::Namespace::Constants object to do the interpolation then and there. This means that the compiled templates contain static strings for these variables, and not calls to the stash. We'll see the code generation process in much more detail in the later Section 7.2.

7.1.8 Template::Namespace::Constants

The Template::Namespace::Constants module is a specialized factory class (like a slimmed-down Template::Directive) that handles compile-time constant folding. A Template::Namespace::Constants object has its own stash, which is initialized with the contents of the CONSTANTS configuration directive (if it was specified). These variables are accessed in the templates using a special prefix (which is constants by default, but can be set to something else using the CONSTANT_NAMESPACE configuration option). We'll see when constant folding comes into play in the Section 7.2; also see the Appendix for more details about CONSTANTS and CONSTANT_NAMESPACE.

7.1.9 Template::Document

A Template::Document module is a thin object wrapper around a compiled template subroutine. The object implements a process method that performs a little bit of housekeeping and then calls the template subroutine. The object also defines template metadata (defined in [% META ... %] directives), and has a blocks method that returns a hash of any additional [% BLOCK xxxx %] definitions found in the template source.

The context processes a Template::Document instance by invoking its process method, passing itself as a parameter; within process, the document executes its main subroutine (which it gets via the block method) and returns a string of output. If there is an error, the context intercepts it with the catch method, which ensures that the error is a Template::Exception object and not a string, and then rethrows it via dia (which is caught by the context in its own process method). Example 7-15 shows this module in action.

Example 7-15. Template::Document::process

```
sub process($context) {
    $output = '';

    eval {
        $block = $self->block;
        $output = &$block($context);
    }

    if $EVAL_ERROR:
```

```
        die $context->catch($EVAL_ERROR);


    return $output;

}
```

7.2 The Runtime Engine

All of this has been building up to one big secret: there is no Template Toolkit runtime. The Template Toolkit uses Perl as its runtime environment. So far, all of the modules we've discussed have been a complex way of turning non-Perl (the templates) into code that the Perl interpreter can execute (compile subroutines).

To see exactly what this means, we need to see what a compiled template looks like. In fact, a compiled template is just a regular Perl subroutine. Here's a very simple one:

```
sub my_compiled_template {

   return "This is a compiled template.\n";

}
```

You're unlikely to see a compiled template this simple unless you wrote it yourself, but it is entirely valid. All a template subroutine is obliged to do is return some output (which may be an empty string, of course). If it can't for some reason, it should raise an error via die:

```
sub my_todo_template {

   die "This template not yet implemented\n";

}
```

If it wants to get fancy, it can raise an error as a Template::Exception object. An exception object is really just a convenient wrapper for the type and info fields.

```
sub my_solilique_template {

   die (Template::Exception->new('yorrick', 'Fellow of infinite jest'));

}
```

Templates generally need to do a lot more than just generate static output or raise errors. They may want to inspect variable values, process another template, load a plugin, run a filter, and so on. Whenever a template subroutine is called, it gets passed a reference to a Template::Context object. It is through this context object that template code can access the features of the Template Toolkit.

We described earlier how the Template::Service object calls on Template::Context to handle a process request from the *outside*. We can make a similar request on a context to process a template, but from within the code of another template. This is a call from the *inside*:

```
sub my_process_template {

   my $context = shift;



   my $output = $context->process('header', { title => 'Hello World' })

            . "\nsome content\n"

            . $context->process('footer');

}
```

This is then roughly equivalent to a source template something like this:

```
[% PROCESS header

   title = 'Hello World'

%]

some content

[% PROCESS footer %]
```

Template variables are stored in and managed by a Template::Stash object. This is a blessed hash array in which template variables are defined. The object wrapper provides get and set methods that implement all the magical variable features of the Template Toolkit.

Each context object has its own stash, a reference to which is returned by the appropriately named stash method. So to print the value of some template variable, or, for example, to represent the following source template:

```
<title>[% title %]</title>
```

we might have a subroutine definition something like this:

```
sub {

    my $context = shift;

    my $stash = $context->stash( );

    return '<title>' . $stash->get('title') . '</title>';

}
```

The stash get method hides the details of the underlying variable types, automatically calling code references, checking return values, and performing other such tricks. If title happens to be bound to a subroutine, we can specify additional parameters as a list reference passed as the second argument to get:

```
[% title('The Cat Sat on the Mat') %]
```

This translates to the stash get call:

```
$stash->get([ 'title' => [ 'The Cat Sat on the Mat' ] ]);
```

Dotted compound variables can be requested by passing a single list reference to the get method in place of the variable name. Each pair of elements in the list should correspond to the variable name and reference to a list of arguments for each dot-delimited element of the variable. Therefore, this:

```
[% foo(1, 2).bar(3, 4).baz(5) %]
```

is equivalent to:

```
$stash->get([ foo => [1,2], bar => [3,4], baz => [5] ]);
```

If there aren't any arguments for an element, you can specify an empty, zero, or null argument list:

```
[% foo.bar %]

$stash->get([ 'foo', 0, 'bar', 0 ]);
```

The set method works in a similar way. It takes a variable name and a variable value that should be assigned to it:

```
[% x = 10 %]

$stash->set('x', 10);



[% x.y = 10 %]

$stash->set([ 'x', 0, 'y', 0 ], 10);
```

So the stash gives us access to template variables and the context provides the higher-level functionality. Alongside the process method lies the include method. Just as with the *PROCESS* and *INCLUDE* directives, the key difference is in variable localization. Before processing a template, the process method simply updates the stash to set any new variable definitions, overwriting any existing values. In contrast, the include method creates a copy of the existing stash, in a process known as *cloning* the stash, and then uses that as a temporary variable store. Any previously existing variables are still defined, but any changes made to variables, including setting the new variable values passed as arguments, will affect only the local copy of the stash (although note that it's only a shallow copy, so it's not foolproof). When the template has been processed, the include method restores the previous variable state by *decloning* the stash.

The context also provides an insert method to implement the *INSERT* directive, but doesn't provide a wrapper method. This functionality can be implemented by rewriting the Perl code and calling include:

```
[% WRAPPER foo %]

   blah blah [% x %]

[% END %]



$context->include('foo', {

    content => "\n   blah blah " . $stash->get('x') . "\n",

});
```

In addition to the template processing methods process, include, and insert, the context defines methods for fetching plugin objects (plugin) and filters (filter):

```
[% USE foo = Bar(10) %]


$stash->set('foo', $context->plugin('Bar', [10]));


[% FILTER bar(20) %]
   blah blah blah
[% END %]


my $filter = $context->filter('bar', [20]);
&$filter("\n   blah blah blah\n");
```

Pretty much everything else you might want to do in a template you can do in Perl code. Things such as IF, UNLESS, FOREACH, and so on all have direct counterparts in Perl.

```
[% IF msg %]
   Message: [% msg %]
[% END %];


if ($stash->get('msg')) {
    $output .= "\n   Message: \n";
    $output .= $stash->get('msg');
    $output .= "\n";
}
```

The best way to get a better understanding of what's going on underneath the hood is to set the $Template::Parser::DEBUG flag to a true value and start processing templates. This will cause the parser to print the generated Perl code for each template it compiles to STDERR. You'll probably also want to set the $Template::Directive::PRETTY option to have the Perl pretty-printed for human consumption (see Example 7-16).

Example 7-16. debug.pl

```
use Template;
use Template::Parser;
use Template::Directive;


$Template::Parser::DEBUG = 1;
$Template::Directive::PRETTY = 1;


my $tt = Template->new( );
$tt->process(*DATA, { cat => 'dog', mat => 'log' })
    || die $tt->error;


_ _DATA_ _
The [% cat %] sat on the [% mat %]
```

The output sent to STDOUT remains as you would expect:

```
The dog sat on the log
```

The output sent to STDERR would look something like the code shown in Example 7-17.

Example 7-17. Compiled main template document block

```
sub {
    my $context = shift || die "template sub called without context\n";
    my $stash   = $context->stash;
    my $output  = '';
    my $error;

    eval { BLOCK: {
        $output .=  "The ";
        $output .=  $stash->get('cat');
        $output .=  " sat on the ";
        $output .=  $stash->get('mat');
        $output .=  "\n";
    } };
    if ($@) {
        $error = $context->catch($@, \$output);
        die $error unless $error->type eq 'return';
    }

    return $output;
}
```

Different versions of the Template Toolkit produce slightly different code. When the compiled document is written out to disk, the Template Toolkit version is part of the compiled code, as shown in Example 7-18.

Example 7-18. A compiled document

```
#------------------------------------------------------------------------
# Compiled template generated by the Template Toolkit version 2.09c
#------------------------------------------------------------------------

Template::Document->new({
    METADATA => {
        'modtime' => '1054300677',
        'name' => 'cat.tt2',
    },
    BLOCK => sub {
        my $context = shift || die "template sub called without context\n";
        my $stash   = $context->stash;
```

```
        my $output  = '';
        my $error;

        eval { BLOCK: {
            $output .=  "The ";
            $output .=  $stash->get('cat');
            $output .=  " sat on the ";
            $output .=  $stash->get('mat');
            $output .=  "\n";
        } };
        if ($@) {
            $error = $context->catch($@, \$output);
            die $error unless $error->type eq 'return';
        }

        return $output;
    },
    DEFBLOCKS => {

    },
});
```

Constants defined in the CONSTANTS configuration option are implemented by the Template::Namespace::Constants module. If we modify *debug.pl* slightly, as shown in Example 7-19, the code produced is slightly different, as shown in Example 7-20.

Example 7-19. debug-constants.pl

```
use Template;
use Template::Parser;
use Template::Directive;

$Template::Parser::DEBUG = 1;
$Template::Directive::PRETTY = 1;

my $tt = Template->new(
    CONSTANTS => {
        cat => 'dog',
    },
);
$tt->process(*DATA, { mat => 'log' })
    || die $tt->error;
```

```
_ _DATA_ _

The [% constants.cat %] sat on the [% mat %]
```

Example 7-20. Compiled main template document block (with constant folding)

```
sub {
    my $context = shift || die "template sub called without context\n";
    my $stash   = $context->stash;
    my $output  = '';
    my $error;

    eval { BLOCK: {
        $output .=  "The ";
        $output .=  'dog';
        $output .=  " sat on the ";
        $output .=  $stash->get('mat');
        $output .=  "\n\n";
    } };
    if ($@) {
        $error = $context->catch($@, \$output);
        die $error unless $error->type eq 'return';
    }

    return $output;
}
```

Notice that [% constants.dog %] was turned into 'dog' at *compile time*, rather than at runtime. This can be a potentially huge gain, especially for templates that contain data that changes infrequently.

7.3 Module Interfaces

Now that our idea of how the Template Toolkit is put together is coming into focus, we can begin discussing the individual modules. In this section, we will describe each core component of the Template Toolkit, as well as the public interface the components present. Developers who wish to extend the Template Toolkit programmatically, or who wish to replace components with their own versions, will do well to pay close attention to the APIs exposed by the components. Most methods are illustrated with small replacement versions that extend the functionality of the component, adding debugging or other simple enhancements—but keep in mind that these are intentionally small examples. You are limited only by your imagination.

Each Template Toolkit module knows about the other modules it needs to do its job, and will create instances of these objects unless one is passed explicitly. This means that modules are loaded and instances are created on demand.

The hash containing the configuration parameters is passed to each module's new method. For example, Template::Service creates a Template::Context instance like so:

```
# In Service.pm
sub _init {
    my ($self, $config) = @_;

    # Some other configuration

    $context = $self->{ CONTEXT } = $config->{ CONTEXT }
        || Template::Config->context($config)
        || return $self->error(Template::Config->error);

    return $self;
}
```

In this case, if a Template::Context instance was part of $config, a new one would not be created. This feature is most useful for overriding settings, such as *TOLERANT*, for specific instances:

```
my $context = Template::Context->new(TOLERANT => 1);
my $tt = Template->new({
    CONTEXT => $context,
    TOLERANT => 0
});
```

7.3.1 Template's process Method

The main interface to the Template Toolkit from within Perl is through the Template module. Recall our basic script from Chapter 6, shown again in Example 7-21.

Example 7-21. ttperl.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;
```

```
my $tt    = Template->new( );
my $input = 'answer.tt';
my $vars  = {
    answer => 42,
    author => 'Douglas Adams',
};

$tt->process($input, $vars)
    || die $tt->error( );
```

Chapter 6 covered the basics of this script; let's discuss the details in more depth. The process method is where the action begins:

```
$tt->process($input, $vars)
    || die $tt->error( );
```

We pass the name of the template file that we want processed, here stored in the $input variable followed by template variables defined in $vars. We could of course pass the template filename as the literal string 'answer.tt2' and save ourselves the effort of creating a temporary variable, but we'll continue to use the $input variable in the examples that follow. As we'll see when we look more closely at the process method, the first argument doesn't always have to be a filename, so it helps to keep things deliberately vague.

The process method returns a true value if the template was sucessfully processed. The output generated will be printed to STDOUT by default, so you'll see it scrolling up your screen when you run the program.

Suppose the source template *answer.tt2* contains the text shown in Example 7-22.

Example 7-22. answer.tt2

```
The answer to the Ultimate Question of Life, the
Universe and Everything is [% answer %].

                -- [% author %]
```

Then we can expect to see the following output generated:

```
The answer to the Ultimate Question of Life, the
Universe and Everything is 42.

                -- Douglas Adams
```

If an error occurs, the process method returns false. In this case, we call the error method to find out what went wrong and report it as a fatal error using die. An error can be returned for a number of reasons, such as the file specified could not be found, had embedded directives containing illegal syntax that could not be parsed, or generated a runtime error while the template was being processed.

7.3.1.1 The process method

The Template process method is the gateway into the Template Toolkit for processing templates:

```
$tt->process($input, $vars, $output, $options)
    || die $tt->error( );
```

process takes up to four arguments: the first specifies the input; the second is a reference to a hash of variables to be made available to the template; the third specifies the destination of the output; and the fourth defines modifiers for that output destination, such as setting binmode on Windows platforms.

7.3.1.1.1 Input template

The first parameter to process specifies where the input should come from. Most often this will be the name of a file:

```
$tt->process('H2G2/entry/earth');
```

The Template Toolkit looks for the template in the directory or directories specified in the INCLUDE_PATH option. If you haven't specified INCLUDE_PATH, the Template Toolkit will look in the current working directory.

In addition to a filename, you can pass a reference to text:

```
my $text = "Hello, [% name %]!";

$tt->process(\$text);
```

or you can pass a reference to a filehandle or a typeglob; as in:

```
my $fh = IO::File->new("file.tmpl") or die $!;

$tt->process($fh);
```

or, as in:

```
$tt->process(\*STDIN);
```

Because the Template Toolkit can read from a filehandle, a quick and easy way to pass a template to process is via a reference to the DATA filehandle. (The DATA filehandle contains everything in the current file after the special marker _ _DATA_ _.) This can simplify writing single-usage scripts and tests greatly, as shown in Example 7-23.

Example 7-23. hello.pl

```
#!/usr/bin/perl



use strict;

use warnings;

use Template;



my $tt = Template->new;

$tt->process(\*DATA) or die $tt->error( );



_ _DATA_ _

Hello, world!
```

7.3.1.1.2 Template variables

The second, optional argument to the process method is a reference to a hash defining template variables and corresponding values. The Template Toolkit allows you to bind almost any kind of Perl data to template variables, including scalars, arrays, hashes, subroutines, and objects. The code in Example 7-24 contains examples of all of these.

Example 7-24. Template variables

```
my $vars = {

    name    => 'Arthur Dent',

    planet  => 'Earth',

    friends => [ 'Ford Prefect', 'Slartibartfast' ],

    people  => {

        'Erotica Gallumbits' => {
```

```
            description => 'Triple breasted whore',
            location    => 'Erotican 6',
        },
        'Bugblatter Beast' => {
            description => 'Ravenous (but stupid)',
            location    => 'Traal',
        },
        'Hotblack Desiato' => {
            description => 'Dead (for tax purposes)',
            location    => 'Milliways',
        },
    },
    consult_guide => sub {
        my $arg = shift;
        return "Don't panic, $arg!";
    },
    magrethea => Acme::Planet->new(name  => 'Magrethea',
                                   edges => 'Crinkly'),
};

$tt->process($input, $vars)
    || die $tt->error( );
```

Internally, these variables are incorporated into the Template::Stash instance that is made available via the Template::Context object.

7.3.1.1.3 Redirecting template output

The default behavior for the process method is to print the output generated by processing a template to STDOUT. The third argument to the process method can be used to specify an alternate destination for the output.

When a plain string is passed as the third argument, it indicates a filename to which output should be written. The OUTPUT_PATH option must be defined to specify a root directory for generating output files. The file specified will be located relative to this directory (see Example 7-25).

Example 7-25. Redirecting Template output to a file

```
my $tt = Template->new(OUTPUT_PATH => '/tmp');


$tt->process($input, $vars, 'output.html')
    || die $tt->error( );
```

In this example, the output will be written to the */tmp/output.html* file.

A reference to a string can instead be passed as the third argument. In this case, the output will be appended to the string. The process method doesn't clear any existing value that the string has (see Example 7-26).

Example 7-26. Redirecting Template output to a scalar

```
my $output;


$tt->process($input, $vars, \$output)

    || die $tt->error( );


print $output;
```

A reference to an array can also be passed as the third argument. The output will be added as an item to the end of the list, as shown in Example 7-27.

Example 7-27. Redirecting Template output to an array

```
my @output;


for my $file (qw( header body footer )) {

    $tt->process($file, $vars, \@output)

        || die $tt->error( );

}


print @output;
```

Another option is to pass a reference to a filehandle that is open and ready for output, as shown in Example 7-28.

Example 7-28. Redirecting Template output to a filehandle

```
use File::Temp qw(tempfile);

my ($fh) = tempfile( );


$tt->process($input, $vars, $fh)

    || die $tt->error( );
```

Yet another option for the third argument is to pass a reference to a subroutine. The subroutine will be called with the output passed to it as the first argument (see Example 7-29).

Example 7-29. Redirecting Template output to a subroutine

```
sub process_to_db {

    my $content = shift;

    $dbh->do("INSERT INTO content (id, content) VALUES (NULL, ?)",

        undef, $content);

}


$tt->process($input, $vars, \&process_to_db)

    || die $tt->error( );
```

The final option for the third argument is to pass a reference to an object that implements a print method. This includes the Apache::Request object and those derived from IO::Handle, for example. The print method will be called with the output passed as the first argument, as per subroutines (see Example 7-30).

Example 7-30. Redirecting Template output to an object with a print method

```
my $fh = IO::File->new(">$tmpfile");

$tt->process($input, $vars, $fh)

    || die $tt->error( );
```

The OUTPUT configuration option can also be used to set the output destination for the Template module as a whole. It can be set to any of the same values as the third argument to process. When a third argument is passed to process, it will override any value defined in OUTPUT (see Example 7-31).

Example 7-31. Using the OUTPUT configuration option

```
my $tt = Template->new(OUTPUT => \$output);



$tt->process($input, $vars)

    || die $tt->error( );
```

This is functionally equivalent to the code in Example 7-32.

Example 7-32. process equivalent of OUTPUT

```
my $tt = Template->new( );



$tt->process($input, $vars, \$output)

    || die $tt->error( );
```

7.3.1.1.4 Processing options

The fourth argument to process is an optional reference to a hash array of processing options. There's only one option at present, binmode, but there's a chance that others will be added at some later date, and this is where they'll go. Example 7-33 shows the code for setting processing options.

Example 7-33. Setting processing options

```
$tt->process($in, $vars, $out, { binmode => 1 })

    || die $tt->error( );
```

The binmode option is typically used on the Windows platform to ensure that line endings are correctly preserved as \r\n instead of being transformed into \n, which is the standard for Unix and other platforms (except Mac OS, which uses \r just to confuse matters). Example 7-34 shows the code for setting binmode on a filehandle.

Example 7-34. Setting binmode on a filehandle

```
local *FH;

open FH, $filename;

binmode FH;
```

For convenience, you can also specify processing options as a list of arguments, as shown in Example 7-35.

Example 7-35. Setting processing options using a list

```
$tt->process($in, $vars, $out, binmode => 1)
    || die $tt->error( );
```

7.3.1.2 The error method

If the process method returns a false value, the error method can be called to return a reference to a *Template::Exception* object that encapsulates information about the error. The exception object has type and info methods that return a short string identifying the kind of error that occurred (e.g., parse, file, etc.), and a message containing further information, respectively. Example 7-36 shows the code for reporting process errors.

Example 7-36. Reporting process errors

```
unless ($tt->process($input, $vars)) {
    my $error = $tt->error( );
    print "error type: ", $error->type( ), "\n";
    print "error info: ", $error->info( ), "\n";
}
```

The nice thing about this object is that you don't need to do anything special with it. You can just print the object and leave the magical stringification method as_string to generate a printable representation of the error. Hence the idiom should be familiar by now (see Example 7-37).

Example 7-37. Error-reporting idiom

```
$tt->process('no/such/page', $vars)
    || die $tt->error( );
```

The message generated is of the form $type error - $info (see Example 7-38).

Example 7-38. Error example

```
file error - no/such/page not found
```

7.3.2 Template::Config

Template::Config provides a factory method for each major component of the Template Toolkit—context, filters, iterator, parser, plugins, provider, service, stash, and constants (see Example 7-39). The type of object that each method creates is, in turn, controlled by a series of variables in the $Template::Config namespace.

Example 7-39. Template::Config package variables

```
$CONTEXT   = 'Template::Context';
$FILTERS   = 'Template::Filters';
$ITERATOR  = 'Template::Iterator';
$PARSER    = 'Template::Parser';
$PLUGINS   = 'Template::Plugins';
$PROVIDER  = 'Template::Provider';
$SERVICE   = 'Template::Service';
$STASH     = 'Template::Stash';
$CONSTANTS = 'Template::Namespace::Constants';
```

These are set when the Template Toolkit is installed; some of them might differ based on how the installation was performed. For example, the fast XS-based stash (Template::Stash::XS) might have been installed instead of the default stash.

Each method works in basically the same way; Example 7-40 shows provider, by way of example.

Example 7-40. Template::Config::provider

```
sub provider {
    my $class  = shift;
    my $params = defined($_[0]) && UNIVERSAL::isa($_[0], 'HASH')
            ? shift : { @_ };

    return undef unless $class->load($PROVIDER);
    return $PROVIDER->new($params)
        || $class->error("failed to create template provider: ",
                     $PROVIDER->error);
}
```

$PROVIDER, as we just saw, defaults to Template::Provider, but it should be apparent that this can be changed to another class:

```
use Template::Config;
$Template::Config::PROVIDER = 'TTBook::Template::Provider';

my $tt = Template->new( ) || die Template->error( );
```

The provider that gets instantiated is going to be a TTBook::Template::Provider, not a Template::Provider.

7.3.2.1 load

Template::Config provides a general module-loading method, load, which takes a name (such as *TTBook::Template::Config*) and loads the module, using require. It returns undef if there were problems loading the module; the error is available via Template::Config->error.

7.3.2.2 preload

preload will load all of the defined components (based on the contents of the variables $SERVICE, $PROVIDER, etc.), mostly for the benefit of long-running processes, such as mod_perl. For example, it is automatically called by the Template frontend when $ENV{'MOD_PERL'} is set:

```
# Template.pm
# preload all modules if we're running under mod_perl
Template::Config->preload( ) if $ENV{ MOD_PERL };
```

preload can be called with extra module names as well, so it can be used to load custom modules:

```
Template::Config->preload('TTBook::Template::Provider',
                  'TTBook::Template::Plugin::NNTP');
```

7.3.2.3 instdir

This helper method returns the directory in which the optional components were installed, such as /usr/local/tt2 or *C:/Template Toolkit 2*. If the optional components were not installed, instdir returns undef and sets $ERROR.

For example, to add the Spash! templates that come with the Template Toolkit to your INCLUDE_PATH, which are installed in *$instdir/templates/spash*, use this code:

```
my $splash = Template::Config->instdir('templates/splash')

    || die Template::Config->error;


my $tt = Template->new(INCLUDE_PATH => [ $splash ]);
```

7.3.3 Template::Constants

Template::Constants defines the constants used and returned by the other elements of the Template Toolkit. Symbols can be imported into your module in the usual way:

```
use Template::Constants qw( :status );
```

7.3.3.1 :status

The status constants are used to check the results of certain operations. The following symbols are imported as part of :status:

```
STATUS_OK               # ok

STATUS_RETURN           # ok, block ended by RETURN

STATUS_STOP             # ok, stopped by STOP

STATUS_DONE             # ok, iterator done

STATUS_DECLINED         # ok, declined to service request

STATUS_ERROR            # error condition
```

Example 7-41, from the insert method of Template::Context, illustrates how the status codes are used; we are iterating through all available providers until one of them successfully loads the template whose name is stored in $name.

Example 7-41. Using ERROR constants

```
foreach my $provider (@$providers) {

    ($text, $error) = $provider->load($name, $prefix);

    next FILE unless $error;

    if ($error =  = Template::Constants::STATUS_ERROR) {

        $self->throw($text) if ref $text;

        $self->throw(Template::Constants::ERROR_FILE, $text);

    }

}

$self->throw(Template::Constants::ERROR_FILE, "$file: not found");
```

7.3.3.2 :error

The *ERROR_** status codes are primarily used when things go wrong. All Template::Exception objects are instantiated with one of these error codes as the type field.

The error constants are:

```
ERROR_RETURN            # return a status code

ERROR_FILE              # file error: I/O, parse, recursion

ERROR_VIEW              # view error

ERROR_UNDEF             # undefined variable value used
```

```
ERROR_PERL           # error in [% PERL %] block

ERROR_FILTER         # filter error

ERROR_PLUGIN         # plugin error
```

7.3.3.3 :chomp

The :chomp symbol imports the whitespace-related constants *CHOMP_NONE*, *CHOMP_ALL*, and *CHOMP_COLLAPSE*. These can be used when specifying a value for the *PRE_CHOMP* and *POST_CHOMP* configuration options:

```
use Template::Constants qw( :chomp );


my $tt = Template->new(TRIM => CHOMP_COLLAPSE);
```

The chomp constants are:

```
CHOMP_NONE           # do not remove whitespace

CHOMP_ALL            # remove whitespace

CHOMP_COLLAPSE       # collapse whitespace to a single space
```

7.3.3.4 :debug

The *DEBUG_** constants let you debug specific core components and not others. These constants are imported with the :debug tag, and include the following:

```
DEBUG_OFF            # do nothing

DEBUG_ON             # basic debugging flag

DEBUG_UNDEF          # throw undef on undefined variables

DEBUG_VARS           # general variable debugging

DEBUG_DIRS           # directive debugging

DEBUG_STASH          # general stash debugging

DEBUG_CONTEXT        # context debugging

DEBUG_PARSER         # parser debugging

DEBUG_PROVIDER       # provider debugging

DEBUG_PLUGINS        # plugins debugging

DEBUG_FILTERS        # filters debugging

DEBUG_SERVICE        # context debugging

DEBUG_ALL            # everything

DEBUG_CALLER         # add caller file/line info
```

These constants are binary OR-ed together to produce a bitmask that specifies the components to debug. For example, to debug the service, context, and provider, use the code in Example 7-42.

Example 7-42. Using constants from Perl

```
use Template;

use Template::Constants qw( :debug );


my $debug = DEBUG_SERVICE | DEBUG_CONTEXT | DEBUG_PROVIDER;

my $tt = Template->new(DEBUG => $debug);

$tt->process("test.tt2") || die $tt->error( );
```

Processing a simple test template, *test.tt2*, yields debugging information for the service, context, and provider objects, as expected:

```
[Template::Provider] creating cache of unlimited slots for [ . ]

[Template::Service] process(test.tt2, <no params>)

[Template::Context] template(test.tt2)

[Template::Context] looking for block [test.tt2]

[Template::Context] asking providers for [test.tt2] [  ]

[Template::Provider] _fetch_path(test.tt2)

[Template::Provider] searching path: ./test.tt2

[Template::Provider] _load(./test.tt2, test.tt2)

[Template::Provider] _compile(HASH(0x823cf1c), <no compfile>)

[Template::Provider] _store(./test.tt2, Template::Document=HASH(0x829f4a8))

[Template::Provider] adding new cache entry

[Template::Service] PROCESS: Template::Document=HASH(0x829f4a8)

[Template::Context] process([ Template::Document=HASH(0x829f4a8) ], <no params>, <unlocalized>)

[Template::Context] template(Template::Document=HASH(0x829f4a8))
```

Using these DEBUG flags, it is possible to debug individual components. Adding the *DEBUG_CALLER* mask causes the debugging messages to include the filename and line number:

```
my $debug = DEBUG_SERVICE |  DEBUG_CALLER;

my $tt = Template->new(DEBUG => $debug);

$tt->process("test.tt2") || die $tt->error( );


[Template::Provider] creating cache of unlimited slots for [ . ] at /usr/local/lib/perl5/

site_perl/5.6.1/Template/Provider.pm line 350

[Template::Service] process(test.tt2, <no params>)

[Template::Context] template(test.tt2) at /usr/local/lib/perl5/site_perl/5.6.1/Template/

Context.pm line 81

...
```

7.3.4 Template::Base

Template::Base implements a common base class used by almost all of the other Template Toolkit modules. Template::Base implements a few important methods that the other modules inherit, namely new, error, and debug. Template::Base has also made its way to CPAN, with slight variations and enhancements, as Class::Base (http://search.cpan.org/dist/Class-Base/).

7.3.4.1 new

When new is called on an object, it invokes the class's _init method, which is where instance-specific initialization takes place. The new method handles the folding of name => value pairs into a single hash; a reference to this hash is passed to the other modules. This is why objects can be created with either a series of name-value pairs or a hashref:

```
my %opts = (
    INCLUDE_PATH => \@paths,
    ANYCASE => 1,
);
my $tt1 = Template->new(\%opts);
my $tt2 = Template->new(%opts);
```

Both invocations are valid and produce similar instances.

7.3.4.2 error

If something goes wrong, most public methods return undef. When this happens, the error message can be retrieved by calling the error method on the instance:

```
$tt->process($template, \%vars)
    || die $tt->error;
```

The error method behaves analogously for classes as well:

```
my $tt = Template->new(\%opts)
    || die Template->error;
```

If error is called with arguments, these arguments become the current error value, and the call to error returns undef, as shown in Example 7-43.

Example 7-43. TTBook::Template::Plugin::LDAP

```
package TTBook::Template::Plugin::LDAP;


use strict;
use Net::LDAP;


sub new {
    my ($self, $context, $host) = @_;

    return $self->error("Missing required host")
        unless ($host);

    my $ldap = Net::LDAP->new($host)
        || return $self->error("Error connecting to $host: $@");

    $ldap->bind;
    return $ldap;
}
```

This short example implements a basic Net::LDAP plugin, which dies if it is not passed a host to which to connect. It also dies if there is a problem connecting to the host.

7.3.4.3 debug

debug generates a debugging message by concatenating all arguments passed into a string and printing it to STDERR. A prefix is added to indicate the module of the caller. This Template::Context subclass emits debugging information whenever a filter is defined using the context's define_filter method. To use these subclasses of standard modules, remember to set the appropriate $Template::Config variable to the name of the class to be used. In Example 7-44, we're setting $Template::Config::CONTEXT to be TTBook::Template::Context::Debugging.

Example 7-44. TTBook::Template::Context::Debugging

```
package TTBook::Template::Context::Debugging;


use base qw(Template::Context);


sub define_filter {
    my ($self, $name, $filter, $is_dynamic) = @_;


    $self->debug(sprintf "defining %s filter '%s'",
                 $is_dynamic ? "dynamic" : "static",
                 $name);


    return $self->SUPER::define_filter($name, $filter, $is_dynamic);
}
```

Given a simple test template of:[1]

[1] We know that the wrap plugin defines a static filter; see Chapter 8.

```
[% USE wrap %]
```

we get this on STDERR:

```
[Template::Context::Debugging] defining static filter 'wrap'
```

debug itself does not check to see whether the module is currently in debugging mode (as specified by the caller via the *DEBUG* configuration option), but $self->{DEBUG} will be set to a true value if debugging was requested. Our debug call should look like this:

```
$self->debug(sprintf "defining %s filter '%s'",
          $is_dynamic ? "dynamic" : "static",
          $name)
    if $self->{ DEBUG };
```

7.3.5 Template::Context

The Template::Context module defines an object class for representing a runtime context in which templates are processed. It provides an interface to the fundamental operations of the Template Toolkit processing engine through which compiled templates can process templates, load plugins and filters, raise exceptions, and so on.

Plugins and dynamic filters are passed a reference to the current context when they are invoked. This reference can then be used to invoke any of the context's methods, such as define_filter or include.

7.3.5.1 stash

This method returns a reference to the stash (see the sectionSection 7.1.3 earlier in this chapter):

```
my $stash = $context->stash;
```

This reference can then be used to get or set values, which are accessible from templates in the usual way:

```
$stash->set('arp', "with or without is the different");


# In the template:

[% arp %]
```

If you get access to the stash while you are within an *INCLUDE*d template, the stash you get will be the localized one; changes made to this stash will not persist to outer scopes (unless the changes are made to nested structures).

7.3.5.2 insert, include, and process

The context provides methods such as include, process, and insert, which implement the *INCLUDE*, *PROCESS*, and *INSERT* directives. For example, a *PROCESS* directive such as:

```
[% PROCESS box quote = 'A city is like a large, complex, rabbit' %]
```

is translated by the Template::Directive class into something like this:

```
$context->process('box', { 'quote' => 'A city is like a large complex rabbit' });
```

7.3.5.3 template

When a template is specified by name, the context instance queries its internal list of Template::Provider instances, using the template method:

```
my $doc = $context->template($name)

    || die $context->error;
```

$doc will be a Template::Document instance, which, as mentioned earlier, is basically an object wrapper around a compiled subroutine (see the Section 7.3.13, earlier in this chapter). If a template can't be loaded for whatever reason, template returns undef, and the error is available via the error method.

7.3.5.4 plugin and filter

The plugin method uses one or more Template::Plugins objects to load plugins specified by USE, and the filter method uses the Template::Filters objects to fulfill FILTER requests. A simple USE statement, such as:

```
[% USE CGI %]
```

is transformed into something like:

```
$stash->set('CGI', $context->plugin('CGI'));
```

A more complex example, such as:

```
[% USE q = CGI('name=darren&title=JAPH') %]
```

becomes more or less what you would expect:

```
$stash->set('q', $context->plugin('CGI', [ 'name=darren&title=JAPH' ]));
```

Arguments supplied to a plugin are passed as a reference to an array. Named arguments are passed in a hashref, as the last element in the array:

```
[% USE MP3('Got the Time.mp3'

      dir = 'Joe Jackson/Look Sharp!'

      utf8 = 1) %]
```

Reformatted slightly, the resulting Perl code is:

```
$stash->set('MP3',
          $context->plugin('MP3',
                        [ 'Got the Time.mp3',
                          { 'dir' => 'Joe Jackson/Look Sharp!',
                            'utf8' => 1
                          }
                        ]));
```

Note that if a name is not specified to USE, the name of the plugin itelf is used.

Filters are handled similarly. The filter method of the context fetches a filter (using the Template::Filters instance), using the filter method. A simple text string, filtered through upper:

```
[% 'do not leave it is not real' | upper %]
```

turns into this Perl:

```
my $filter = $context->filter('upper')
         || $context->throw($context->error);


$output .= 'do not leave it is not real';


&$filter($output);
```

The upper filter is a static filter, so there isn't much interesting going on there: the filter method calls on the Template::Filters instances to load the filter subroutine. If this fails, the throw method creates a new Template::Exceptions instance and passes it up. Otherwise, the subroutine reference gets assigned to $filter, and we invoke filter on the text waiting to be filtered.

Dynamic filters get passed arguments, which are collected and passed in the same way for filters as they are for plugins:

```
[% FILTER format("%.12f");
    PI = 22 / 7;
    radius = 14.5;
    PI * radius * radius;
   END
%]
```

Arguments are passed as a reference to an array:

```
my $filter = $context->filter('format', [ '%.12f' ])
         || $context->throw($context->error);


$stash->set('PI', 22 / 7);
$stash->set('radius', 14.5);
$output .=  $stash->get('PI') * $stash->get('radius') * $stash->get('radius');


&$filter($output);
```

7.3.5.5 define_filter

Use this method to define a filter:

```
use Term::ANSIColor qw(colored);

$context->define_filter('red', sub { colored($_[0], "red") }, 0);
```

Pass the name of the filter, a reference to the filter sub, and a boolean indicating whether the filter is a dynamic or static filter. This filter becomes available immediately.

7.3.6 Template::Provider

The Template::Provider is used to load, parse, compile, and cache templates. This object may be subclassed to provide more specific facilities for loading or otherwise providing access to templates.

The Template::Context objects maintain a list of Template::Provider objects that are polled in turn (via fetch) to return a requested template. Each may return a compiled template, raise an error, or decline to serve the request, giving subsequent providers a chance to do so.

This is the "Chain of Responsibility" pattern. See *Design Patterns*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addision-Wesley), for further information.

Template::Provider has a few interesting methods, described in the next sections.

7.3.6.1 fetch($name)

fetch returns a compiled template for $name. If the template cannot be found, (undef, STATUS_DECLINED) is returned. If an error occurs (e.g., read error, parse error), ($error, STATUS_ERROR) is returned, where $error is the error message generated. If the TOLERANT flag is set, the method returns (undef, STATUS_DECLINED) instead of returning an error.

Template::Provider can also be used as a general-purpose file loader. Because a normal text file (without delimiters) is a valid template, any file can be loaded via the fetch method (see Example 7-45).

Example 7-45. Using Template::Provider for non-Template Toolkit files

```
my $prov = Template::Provider->new(ABSOLUTE => 1);

my $file = "/etc/passwd";



my ($doc, $error) = $prov->fetch($file);

die "Couldn't load $file" if defined $error;
```

As noted earlier, fetch returns a pair of values: the Template::Document instance and an error string. Only one of the two will be defined: if there was an error loading the file, $error will contain STATUS_DECLINED (from Template::Constants), and $doc will be undefined; if the file was loaded without incident, $error will be undefined and $doc will contain the Template::Document instance, which will have modtime and name methods, at the very least:

```
printf "%s was last modified on %s.\n",

       $doc->name( ), $doc->modtime( );
```

The modtime method returns the number of seconds since the epoch, which can be passed to localtime to get a more meaningful value:

```
printf "%s was last modified on %s.\n",

       $doc->name( ), localtime($doc->modtime( ));
```

More interesting formatting is possible using POSIX::strftime:

```
use POSIX qw(strftime);

my @date = localtime($doc->modtime( ));

printf "%s was last modified on %s.\n",

       $doc->name( ), strftime("%Y/%m/%d", @date);
```

This might return, for example:

```
/etc/passwd was last modified on 2002/10/18.
```

7.3.6.2 store($name, $template)

This method stores the compiled template, $template, in the cache as $name. Subsequent calls to fetch($name) will return this template in preference to any disk-based file.

7.3.6.3 paths

paths expands the object's INCLUDE_PATHS and returns a reference to an array of pathnames. Since Version 2.08 of the Template Toolkit, elements of INCLUDE_PATH can be subroutine references or objects, and paths will correctly call and expand these references.

```
package TTBook::Template::Provider::ExpandPaths;


use strict;

use base qw(Template::Provider);


sub paths {

    my $self = shift;

    my $orig_paths = $self->SUPER::paths( );

    my ($path, @paths, %unique);


    for $path (@$orig_paths) {

        my @chunks = split '/', $path;


        while (@chunks) {

            push @paths, join '/', @chunks;

            pop @chunks;

        }

    }


    # Remove duplicates from the list

    @paths = grep { ++$names{$_} =  = 1 } grep { length } @paths;


    return \@paths;

}
```

TTBook::Template::Provider::ExpandPaths will expand each element of @orig_paths into a list consisting of expanded versions of @orig_paths. For example, given an INCLUDE_PATH of /web/www/html:/web/search/html, this provider will return a reference to this array:

```
('/web/www/html',

 '/web/www',

 '/web',

 '/web/search/html',

 '/web/search')
```

Using this provider allows a user to situate templates anywhere along the INCLUDE_PATH, which means that they can be shared. For example, general headers and footers can be located in */web*, while specific subdirectories could implement their own header and/or footer simply by placing a file somewhere along the search path.

7.3.7 Template::Stash

The most common thing that a template needs to do is to access variables. This is where the stash comes in. As we saw earlier, the stash manages the variables that are available to templates and implements the dot (.) operator.

7.3.7.1 get, set

Template variables are stored in and managed by a Template::Stash object. This is a blessed hash array in which template variables are defined:

```
my $stash = Template::Stash->new({
    planet => 'Earth',
    about  => 'Mostly harmless'
 });
```

The object wrapper provides get and set methods that implement all the magical variable features of the Template Toolkit.

Each context object has its own stash, a reference to which can be returned by the appropriately named stash method. So to print the value of some template variable, or, for example, to represent the following source template:

```
<entry>[% planet %]</entry>
<about>
[% about %]
</about>
```

we might have a subroutine definition something like this:

```
sub {
    my $context = shift;
    my $stash   = $context->stash( );
    return '<entry>' . $stash->get('planet') . "</entry>\n"
         . "<about>\n"  . $stash->get('about')  . "\n</about>\n";
}
```

The get method retrieves the variable named by the first parameter:

```
$value = $stash->get('planet');
```

Dotted compound variables can be requested by passing a single list reference to the get method in place of the variable name. Each pair of elements in the list should correspond to the variable name and reference a list of arguments for each dot-delimited element of the variable.

```
[% guide.entry(314159).about %]


$stash->get([ 'guide', 0, 'entry', [ 314159 ], 'about', 0 ]);
```

If there are no arguments for an element, you can specify an empty, zero, or null argument list:

```
[% hitchhiker.name %]


$stash->get([ 'hitchhiker', 0, 'name', 0 ]);
```

The set method works in a similar way. It sets the variable named in the first parameter to the value specified in the second:

```
[% x = 10 %]


$stash->set('x', 10);
```

Dotted compound variables may be specified as per get:

```
[% x.y = 10 %]


$stash->set([ 'x', 0, 'y', 0 ], 10);
```

If the third parameter evaluates to a true value, the variable is set only if it did not have a true value before. This implements the behavior of the DEFAULT directive:

```
$stash->set('about', 'This page intentionally left blank.', 1);
```

7.3.7.2 clone, declone

The stash has clone and declone methods that are used by the template processor to make temporary copies of the stash for localizing changes made to variables. This localization takes place for INCLUDE directives (but not PROCESS). Conceptually, INCLUDE looks like this:

```
$stash = $stash->clone( );

$content->process($template);

$stash = $stash->declone( );
```

The clone method creates and returns a new Template::Stash object that represents a localized copy of the parent stash. Variables can be freely updated in the cloned stash; when declone is called, the original stash is returned with all its members intact and in the same state as they were before clone was called.

For convenience, a hash of parameters may be passed into clone that are used to update any simple variable (i.e., those that don't contain any namespace elements, such as foo and bar but not foo.bar) while cloning the stash. For adding and updating complex variables, the set method should be used after calling clone. This will correctly resolve and/or create any necessary namespace hashes.

The declone method returns the original stash and is used to restore the state of a stash as described earlier.

7.3.8 Template::Filters

The Template::Filters module implements a provider for creating and/or returning subroutines that implement the standard filters. As is done with its brother Template::Provider, the context keeps an array of Template::Filters instances handy for fetching filters. The filter method of the context iterates through these instances and calls the fetch method on them, passing the name of the desired filter, until one of them returns a nonerror value:

```
# Context.pm (simplified)

sub filter {

    my ($self, $name, $args) = @_;

    my ($filter, $error);


    foreach my $provider (@{ $self->{ LOAD_FILTERS } }) {

        ($filter, $error) = $provider->fetch($name, $args, $self);

        last unless $error;

    }


    return $filter;

}
```

7.3.8.1 new

The contructor for Template::Filters receives the *FILTERS* option, which should be a hashref of name => filter sub pairs. These filters become part of the instance, and calls to fetch look in this list of filters in addition to the standard filters.

```
use Text::Soundex qw(soundex);

use Text::Metaphone qw(Metaphone);


my $tf = Template::Filters->new({

    FILTERS => {

        soundex => sub { soundex($_[0]) },

        metaphone => sub { Metaphone($_[0]) },

    }

});
```

The soundex and metaphone filters can now be used like any other filter:

```
[% PROCESS page | metaphone %]
```

7.3.8.2 fetch

The main method that Template::Filters implements is fetch, as illustrated earlier. fetch will be called with three arguments: the name of the filter being requested (which should be either one of the standard filters or a filter defined in the FILTERS option passed to new); a reference to an array of configuration parameters; and the current Template::Context instance.

7.3.8.3 store

Use store to store a new filter:

```
$filters->store('soundex', sub { soundex($_[0]) });
```

This is what is called by the context's define_filter method. You should probably use define_filter if you are installing a new filter because the context will always install the new filter in the right place. If you are creating a replacement for Template::Filters, you might want to implement store differently. For example, the Template::Filters subclass TTBook::Template::Filters::Logging logs when a filter is fetched or stored, as shown in Example 7-46.

Example 7-46. TTBook::Template::Filters::Logging

```
package TTBook::Template::Filters::Logging;


use strict;

use base qw( Template::Filters );


use Template::Filters;


# Store the filter, and store the time

sub store {

    my ($self, $name, $filter) = @_;

    my $now = time;
```

```
    $self->SUPER::store($name, $filter);
    $self->{ FILTER_TIMESTAMPS }->{ $name } = $now;

    $self->debug("store($name => $filter) at $now");

    return 1;
}

# Keeps track of the difference in time between when the filter
# was stored and when it was first used.
sub fetch {
    my ($self, $name, $args, $context) = @_;
    my ($filter_sub, $filter_ts, $now);

    $filter_sub = $self->SUPER::fetch($name, $args, $context);
    $filter_ts = $self->{ FILTER_TIMESTAMPS }->{ $name };
    $now = time;

    $self->debug("fetch($name) at $now");

    return $filter_sub;
}
```

The simple Template::Filters subclass shown in Example 7-47 counts the number of times each filter is fetched.

Example 7-47. TTBook::Template::Filters::Counting

```
package TTBook::Template::Filters::Counting;

use strict;
use base qw( Template::Filters );

sub fetch {
    my ($self, $name, $args, $context) = @_;
    my $count = $self->{ FILTERS_COUNT } ||= { };

    $count->{ $name }++;
    $self->debug("filter $name has been loaded $count->{$name} times.");
```

```
    return $self->SUPER::fetch($name, $args, $context);
}
```

7.3.9 Template::Plugin

The Template::Plugin module provides both an API and a base class for plugins that implement the three basic methods that are required for a plugin to be loaded by the Template::Plugins module: load, new, and error. All the standard plugins inherit from Template::Plugin. By default, a Template::Plugin-based module has no functionality other than to load correctly; subclasses may override these and of course, can implement any other methods they need to perform their duties.

7.3.9.1 load

This method is called when the plugin module is first loaded. It is called as a package method and thus implicitly receives the package name as the first parameter. A reference to the Template::Context object loading the plugin is also passed. The default behavior for the load method is to simply return the class name; the calling context then uses this class name to call the new package method:

```
package MyPlugin;

sub load {              # called as MyPlugin->load($context)
    my ($class, $context) = @_;
    return $class;      # returns 'MyPlugin'
}
```

7.3.9.2 new

This method is called to instantiate a new plugin object for the *USE* directive. It is called as a package method against the class name returned by load. A reference to the Template::Context object creating the plugin is passed, along with any additional parameters specified in the *USE* directive.

```
sub new {               # called as MyPlugin->new($context)
    my ($class, $context, @params) = @_;
    bless {
        _CONTEXT => $context,
        _PARAMS => \@params,
    }, $class;          # returns blessed MyPlugin object
}
```

7.3.9.3 error

This method, inherited from the Template::Base module, is used for reporting and returning errors. It can be called as a package method to set/return the $ERROR package variable, or as an object method to set/return the object's _ERROR member. When called with an argument, it sets the relevant variable and returns undef. When called without an argument, it returns the value of the variable.

```
sub new {
    my ($class, $context, $dsn) = @_;

    return $class->error('No data source specified')
        unless $dsn;
```

```
    bless {
        _DSN => $dsn,
    }, $class;
}

...

my $something = MyModule->new( )
    || die MyModule->error( );

$something->do_something( )
    || die $something->error( );
```

The Template::Plugins object that handles the loading and use of plugins calls the new and error methods against the package name returned by the load method. In pseudocode terms, it looks something like this:

```
$class  = MyPlugin->load($context);       # returns 'MyPlugin'

$object = $class->new($context, @params)  # MyPlugin->new(...)
    || die $class->error( );              # MyPlugin->error( )
```

The load method may alternately return a blessed reference to an object instance. In this case, new and error are then called as *object* methods against that prototype instance. Example 7-48 provides a concrete illustration: this plugin implements a print service.

Example 7-48. TTBook::Template::Plugin::Printer

```
package TTBook::Template::Plugin::Printer;

use strict;
use vars qw($PRINTER $SERVER);
use base qw(Template::Plugin);

use Template::Plugin;
use Template::Exception;
use Net::Printer;

$PRINTER = "jeckyl";
$SERVER = "mr-hyde";

sub load {
    my ($class, $context) = @_;
    my $printer = Net::Printer->new(printer => $PRINTER,
```

```
                              server  => $SERVER);
    my $self = bless {
        _CONTEXT => $context,
        _PRINTER => $printer,
    }, $class;

    return $self;
}

sub new {
    my ($self, $context) = @_;
    return $self;
}

sub print {
    my ($self, $data) = @_;
    my ($printer, $context) = @$self{ qw( _PRINTER _CONTEXT) };

    my $result = $printer->printstring($data);
    $context->throw('printer', $result)
        unless (int($result) =  = 1);

    return "";
}

1;
```

In this example, we have implemented a Singleton plugin. One instance of TTBook::Template::Plugin::Printer gets created when load is called, and it simply returns itself for each call to new.

Because calls to print throw printer exceptions if there is a problem, they should be wrapped in TRY / CATCH blocks, as shown in Example 7-49.

Example 7-49. The Printer plugin

```
[% USE Printer %]
[% TRY %]
    [% Printer.print(data) %]
[% CATCH printer %]
    There was an error printing: [% error %]
[% END %]
```

7.3.10 Template::Plugins

Template::Plugins defines a plugins provider. It is used in almost the same way as Template::Filters and has a similar interface. The Template Toolkit allows multiple plugin providers, again using the "Chain of Responsibility" pattern.

7.3.10.1 new

The new construtor method handles the *PLUGIN* configuration option, which should be a hashref of name => plugin module pairs:

```
my $tp = Template::Plugins->new({
    PLUGINS => {
        'css' => 'TTBook::Template::Plugin::CSS',
        'javascript' => 'TTBook::Template::Plugin::JS',
    },
});
```

These newly defined plugins are stored in the instance, which is where fetch looks first when trying to load plugins. new also stores the *PLUGIN_BASE* and *LOAD_PERL* options, if present. These options affect how fetch finds plugins.

7.3.10.2 fetch

fetch is called by the context's plugin method, in the same way as the filter provider's fetch method gets called from the filter method. fetch is called with the name of the plugin, a reference to an array of parameters, and the current context, and is expected to return a blessed object, which is used in the templates.

The *PLUGIN_BASE* configuration option defines a relative base for loading plugins. If a plugin cannot be loaded by name from *PLUGINS*, each element in *PLUGIN_BASE* (which should be a reference to an array) is prepended to the name, in turn, until the plugin is found or the list exhausted. Template::Plugin is always appended to this list.

The LOAD_PERL configuration option tells the plugin's provider that standard Perl modules can be treated as plugins, after the list of known plugins has been checked and the PLUGIN_BASE search path exhausted. For example, to load the WWW::Wikipedia module, set LOAD_PERL to 1 and use:

```
[% USE wiki = WWW.Wikipedia %]
```

There is no standard WWW.Wikipedia plugin, so the plugins provider will try to load WWW::Wikipedia. Modules loaded this way must have a new method; the result of calling this method is what is returned by the call to fetch.

Given a two-element PLUGIN_BASE and LOAD_PERL:

```
my $tt = Template::Plugins->new({
    PLUGIN_BASE => [ 'TTBook::Template::Plugin',
                     'MyOrg::Template::Plugin' ],
    LOAD_PERL => 1,
});
```

and a simple *USE* statement:

```
[% USE Monitor %]
```

the plugin's provider will look for TTBook::Template::Plugin::Monitor, MyOrg::Template::Plugin::Monitor, Template::Plugin::Monitor, and Monitor; it will throw a plugin exception if none of those is found.

7.3.11 Template::Parser and Template::Grammar

Template::Parser and Template::Grammar are closely related. The parser starts things off by tokenizing the input template, and then refers to the grammar to determine whether the sequence of tokens gleaned from the template makes any sense. Template::Directive is used to generate the Perl code that represents the template.

Template::Parser is the ultimate recipient of all configuration parameters that affect the style of the template, such as TAG_STYLE, START_TAG, END_TAG, ANYCASE, INTERPOLATE, PRE_ and POST_CHOMP, V1DOLLAR, and GRAMMAR (see the Appendix for all the configuration options). The main methods of the parser are new and parse, as shown in Example 7-50.

Example 7-50. Creating and using parser and grammar objects

```
my $parser = Template::Parser->new({
  ANYCASE => 1,
  GRAMMAR => [% namespace %]::Template::Grammar->new( ),
});

my $data = $parser->parse($template_string);
my $doc = Template::Document->new($data);
```

$data is a reference to a hash, which is in the format expected by Template::Document.

In general, there isn't much reason to use Template::Parser or Template::Grammar directly. To get compiled versions of templates, use Template::Provider rather than Template::Parser—the version returned by the parser is in a raw, uncompiled form, used primarily for communication between the parser and the provider. Template::Grammar is generated using the *parser/Parser.yp* source file, which is processed by Parse::Yapp. It consists primarily of the rules and states used by the parser when determining whether the set of tokens created from the input template is valid. If you're interested in how this works, see Chapter 8.

7.3.12 Template::Directive

The Template::Directive module is a Perl factory—it exists only to return strings of valid Perl code, based on input from the parser. Template::Directive interacts closely with Template::Parser and Template::Grammar: the parser tokenizes the input, and the grammar determines which method to call on the factory class to produce the code that implements a directive.

The grammar also determines the arguments that get passed to the factory method, based on the type of directive. For example, an anonymous BLOCK definition, such as [% BLOCK %] Hello! [% END %], receives one argument, which is the contents of the block. (It is possible that this block contains other compiled directives, rather than just plain text, of course; this doesn't affect the generation of the code.) The factory code for anonymous blocks looks like this:

```
#------------------------------------------------------------------------
# anon_block($block)                          [% BLOCK %] ... [% END %]
#------------------------------------------------------------------------

sub anon_block {
    my ($class, $block) = @_;
    $block = pad($block, 2) if $PRETTY;

    return <<EOF;

# BLOCK
$OUTPUT do {
    my \$output  = '';
    my \$error;

    eval { BLOCK: {
$block
    } };
    if (\$@) {
```

```
        \$error = \$context->catch(\$@, \\\$output);
        die \$error unless \$error->type eq 'return';
    }

    \$output;
};
EOF
}
```

It's kind of ugly, primarily because the return value from the method is a string containing Perl, which will be compiled later.

The $block variable contains the results of calling other factory methods (e.g., ident, which handles [% GET foo %] directives). The pad function adds leading spaces to each line in $block if the $PRETTY variable (actually $Template::Directive::PRETTY) is set to a true value to indicate a human will read the generated code.

To control the code that gets written out for a given directive, subclass Template::Directive, and implement the appropriate method or methods. Many of these methods have names that are similar to the directives they implement, such as get, call, insert, and include, but many of the methods have unintuitive names. The easiest way to figure out which methods are called for each directive is to examine the grammar defined in *Parser.yp* (see Chapter 8).

You shouldn't need to touch most of the definitions in this module, but you will need to subclass it to implement any changes to the language you might want to make.

The best way to get a feel for how this module works is to set both $Template::Parser::DEBUG and $Template::Directive::PRETTY to 1, as noted earlier.

Template::Directive sports the following methods:

template($block)

An overall template wrapper.

anon_block($block)

An anonymous block.

block($block)

Any block of template directives.

textblock($text)

A block of text.

text($text)

A single piece of text.

quoted($items)

A quoted string.

ident($ident)

An identifier.

identref($ident)

A reference to an identifier.

assign($var, $val, $default)

An assignment.

args($args)

A list of arguments.

filenames($names)

A filename.

get($expr)

The GET directive.

call($expr)

The CALL directive.

set($setlist)

The SET directive.

default($setlist)

The DEFAULT directive.

insert($nameargs)

The INSERT directive.

include($nameargs)

The INCLUDE directive.

process($nameargs)

The PROCESS directive.

if($expr, $block, $else)

The IF directive.

foreach($target, $list, $args, $block)

The FOREACH directive.

next($nameargs, $block)

The NEXT directive.

wrapper($nameargs, $block)

The WRAPPER directive when specific with a single file.

multi_wrapper($file, $hash, $block)

The WRAPPER directive when specific with multiple files.

while($expr, $block)

The WHILE directive.

switch($expr, $case)

The SWITCH directive.

try($block, $catch)

The TRY directive.

throw($nameargs)

The THROW directive.

return( )

The RETURN directive.

stop( )

The STOP directive.

use($lnameargs)

The USE directive.

view($nameargs, $block, $defblocks)

The VIEW directive.

perl($block)

The PERL directive.

no_perl( )

The PERL directive when EVAL_PERL is disabled.

rawperl($block, $line)

The RAWPERL directive.

filter($lnameargs, $block)

The FILTER directive.

capture($name, $block)

Generates code to capture the output of a directive into a variable.

macro($ident, $block, $args)

The MACRO directive.

debug($nameargs)

The DEBUG directive.

7.3.13 Template::Document

This module defines an object class whose instances represent compiled template documents. The parser module creates a Template::Document instance to encapsulate a template as it is compiled into Perl code.

7.3.13.1 new

new expects a hashref containing BLOCK, DEFBLOCKS, and METADATA items. The BLOCK item should contain a reference to a Perl subroutine or a textual representation of Perl code, as generated by the Template::Parser module, which is then evaluated into a subroutine reference using eval. The DEFBLOCKS item should be a hashref containing further named BLOCKs, which may be defined in the template. The keys represent BLOCK names, and the values should be subroutine references or text strings of Perl code, such as the main BLOCK item. The METADATA item should be a hashref of metadata items relevant to the document.

Though Template::Document instances are usually created by the provider as it receives parsed data from the parser, it is possible to create standalone instances as well:

```
my $doc = Template::Document->new({
    BLOCK => sub { return "Hello!" },
    METADATA => { name => "greeting" },
    DEFBLOCKS => { }
});


print $doc->name( );
```

The only required parameter in the hashref is BLOCK:

```
my $timer = Template::Document->new({
    BLOCK => sub { time },
});
```

7.3.13.2 process

The process method can then be called on the instantiated Template::Document object, passing a reference to a Template::Content object as the first parameter. This will install any locally defined blocks (DEFBLOCKS) in the contexts BLOCKS cache (via a call to visit), so that they may be subsequently resolved by the context. The main BLOCK subroutine is then executed, passing the context reference on as a parameter. The text returned from the template subroutine is then returned by the process method, after calling the context leave method to permit cleanup and de-registration of named BLOCKs previously installed.

7.3.13.3 write_perl_file

The Template::Document module implements the methods necessary to write a compiled template to disk. These methods are as_perl and write_perl_file. If COMPILE_EXT and/or COMPILE_DIR are set, the provider calls write_perl_file, supplying it with a filename.

7.3.13.4 AUTOLOAD

Template::Document has an *AUTOLOAD* method that provides read-only access to the metadata defined for that template.

This includes all items defined in the template with *META*:

```
# thneed.tt2

[% META title = 'You need a thneed!'

       author = 'The Once-ler' %]



# Perl

my $doc = $context->template('thneed.tt2');

print $doc->author;
```

7.3.14 Template::Exception

The Template::Exception module defines an object class for representing exceptions within the template processing life cycle.

Exceptions can be thrown from Perl code in several different ways. The most straightforward way is to call die with a Template::Exception object as the argument. This will then be caught by any enclosing *TRY* blocks from where the code was called:

```
use Template::Exception;

...

die(Template::Exception->new('bad.things',

                             'Bad things happened.'));
```

This can be caught normally in the template:

```
[% USE Something %]

[% TRY %]

   ...

[% CATCH bad.things %]

   "Error: $error" ;

[% END %]
```

which will output:

```
Error: bad.things error - Bad things happened.
```

The info field can also be a reference to another object or data structure, if required:

```
die(Template::Exception->new('bad.things', {

       module => 'foo.pl',

       errors => [ 'bad permissions', 'naughty boy' ],

   }));
```

Later, in a template:

```
[% TRY %]

   ...

[% CATCH bad.things %]

  [% error.info.errors.size or 'no';

     error.info.errors.size =  = 1 ? ' error' : ' errors' %]

  in [% error.info.module %]:

     [% error.info.errors.join(', ') %].

[% END %]
```

it generates this output:

```
2 errors in foo.pl:

  bad permissions, naughty boy.
```

You can also call die with a single string, as is common in much existing Perl code. This will automatically be converted to an exception of the undef type (that's the literal string undef, not the undefined value). If the string isn't terminated with a newline, Perl will append the familiar *at $file line $line* message.

```
sub foo {

    # ... do something ...

    die("I'm sorry, Dave, I can't do that\n");

}
```

Within plugins, which are passed a reference to the context as the second argument, or some extension code that has the current Template::Context in scope, you can also raise an exception by calling the context throw method. You can pass it Template::Exception object reference, a pair of ($type, $info) parameters, or just an $info string to create an exception of undef type:

```
$context->throw($e);            # exception object

$context->throw('Denied');      # 'undef' type

$context->throw('bad.things', 'Bad things happened.');
```

7.3.15 Template::Iterator

The Template::Iterator module provides an easy way to create iterators. Iterator objects can be used within FOREACH loops, and they maintain the magic loop variable available in FOREACH loops.

To create a Template::Iterator instance, pass to the constructor a reference to an array:

```
use Template::Iterator;

my $iter = Template::Iterator->new(\@data);
```

Data is retrieved by calling get_first and then get_next until each item in the original list has been returned.

Iterator instances can be returned by methods designed to be called within *FOREACH* loops:

```
sub results {

    my $self = shift;

    my $iter = Template::Iterator->new($self->{ _RESULTS });



    return $iter;

}
```

From within a template, usage is as you would expect:

```
[% FOREACH result = search.results %]

    ...
```

Template::Iterator automatically provides the size, max, index, count, first, last, prev, and next methods, based on the result set used to initialize the instance. These methods correspond to the methods of the same names that can be called on loop within a FOREACH loop:

```
[% FOREACH result = search.results %]

    Size:   [% loop.size  # $iter->size( ) %]

    Max:    [% loop.max   # $iter->max( ) %]

    Index:  [% loop.index # $iter->index( ) %]

    Count:  [% loop.count # $iter->count( ) %]

    First:  [% loop.first # $iter->first( ) %]
```

```
    Last:   [% loop.last  # $iter->last( ) %]

    Prev:   [% loop.prev  # $iter->prev( ) %]

    Next:   [% loop.next  # $iter->next( ) %]

[% END %]
```

The astute reader will notice the similarity between loop and $iter; they are in fact the same Perl object.

A Template::Iterator instance can be created with a reference to an array of items, as noted earlier, or with an object that implements an as_list method. We can rewrite the preceding example to have as_list:

```
sub as_list {

    my $self = shift;

    return $self->{ _RESULTS };

}



sub results {

    my $self = shift;

    return Template::Iterator->new($self);

}
```

The constructor will also accept a reference to a hash array and will expand it into a list in which each entry is a hash array containing a key and value item, sorted according to the hash keys:

```
my $iter = Template::Iterator->new({

    foo => 'Foo Item',

    bar => 'Bar Item',

});
```

This is equivalent to:

```
my $iter = Template::Iterator->new([

    { key => 'bar', value => 'Bar Item' },

    { key => 'foo', value => 'Foo Item' },

]);
```

Chapter 8. Extending the Template Toolkit

Most of the customization you are likely to perform will fall under one of two categories: creating new frontends and writing filters and plugins. However, some things cannot be handled with a new frontend or by writing a plugin, such as modifying how the provider finds templates to process or limiting access to certain plugins. Luckily, the Template Toolkit makes it easy to replace or extend any of the core components; its modular design makes replacing individual components simple. Chapter 7 gives public API details for each component.

8.1 Using and Implementing Noncore Components

Each Template Toolkit module knows about the other modules it needs to do its job, and will create instances of these objects unless one is passed to it explicitly. This means that modules are loaded and instances created on demand. The Template::Config module provides a convenient and centralized place to override core elements of the Template Toolkit, in the form of factory methods for each major component—context, filters, iterator, parser, plugins, provider, service, stash, and constants. The type of object that each method creates is, in turn, controlled by a series of variables in the $Template::Config namespace:

```
$CONTEXT   = 'Template::Context';
$FILTERS   = 'Template::Filters';
$ITERATOR  = 'Template::Iterator';
$PARSER    = 'Template::Parser';
$PLUGINS   = 'Template::Plugins';
$PROVIDER  = 'Template::Provider';
$SERVICE   = 'Template::Service';
$STASH     = 'Template::Stash';
$CONSTANTS = 'Template::Namespace::Constants';
```

These are given default values when the Template Toolkit is installed, and some of them might differ based on how the installation was performed. For example, the fast XS-based Stash (Template::Stash::XS) might have been installed instead of the default Stash.

The hash containing configuration parameters is passed around to each module's constructor. For example, Template::Service creates a Template::Context instance like so:

```
# In Service.pm
sub _init {
  my ($self, $config) = @_;

  # Some other configuration

  $context = $self->{ CONTEXT } = $config->{ CONTEXT }
    || Template::Config->context($config)
    || return $self->error(Template::Config->error);

  return $self;
}
```

In this case, if a Template::Context instance was part of $config, a new one would not be created. This feature is most useful for overriding settings, such as TOLERANT, for specific instances:

```
my $context = Template::Context->new(TOLERANT => 1);
my $tt = Template->new({
  CONTEXT => $context,
  TOLERANT => 0
});
```

To give a feel for implementing core module replacements, we'll illustrate a few simple ones. In most cases, the core modules can serve as a base class, and our subclasses need to override only a few methods.

All of the provider classes—Template::Provider, Template::Plugins, and Template::Filters—are stored as arrays, rather than as single items, specifically so that they can be supplemented by new modules. Simply create your new module and pass it around in the appropriate array when you create your Template object. The PREFIX_MAP gives the context hints as to which provider it should consult, based on the prefix, which looks very similar to the scheme of a URI:

```
[% PROCESS foo:bar/baz %]
```

The preceding code would invoke the provider mapped to foo to resolve the template *foo/bar*:

```
my $tt = Template->new({
    LOAD_TEMPLATES => [
        Template::Provider::Foo->new( ),
        Template::Provider->new( ),
    ],
    PREFIX_MAP => {
        foo     => 1,
        default => 0,
    },
});
```

8.1.1 A Provider That Can Fetch Files over HTTP

A relatively common question on the mailing list is, "Can I fetch templates via HTTP?" The official Template Toolkit FAQ[1] explains that, yes, you can, simply by using Template::Provider::HTTP. The problem, though, is that Template::Provider::HTTP does not exist.

[1] Find it at http://www.template-toolkit.org/faq.html.

Template::Provider already does most of what we want, including caching. Template::Provider::HTTP simply needs to add an LWP::UserAgent instance and customize the fetching process to use URIs rather than filesystem paths:

```
package Template::Provider::HTTP;


use strict;
use vars qw($VERSION);
use base qw(Template::Provider);


$VERSION = 1.00;


use File::Spec;
use HTTP::Request::Common qw(HEAD GET);
use LWP::UserAgent;
use Template::Constants qw(:status);
use Template::Provider;
use URI;
use URI::Escape qw(uri_escape);
```

In addition to Template::Provider and Template::Constants (for the STATUS constants), we need LWP::UserAgent, with which we will do the actual fetching, HTTP::Request::Common to create HTTP::Request objects (the GET and HEAD functions are very convenient shortcuts), and the URI, URI::Escape, and File::Spec modules to manipulate URIs and files.

When a Template::Provider::HTTP object is created, we need to also create an LWP::UserAgent instance.

Template::Provider::_init already handles the caching parameters, so we call it from our own _init:

```
sub _init {

    my ($self, $params) = @_;

    my ($ua, %lwp_args, $lwp_arg);



    $self->SUPER::_init($params);
```

Now we can do the LWP initialization. This list contains all the constructor options that LWP knows about, but for the sake of consistency with the Template Toolkit's native configuration methods, we require all uppercase option names:

```
for $lwp_arg (qw(agent from timeout use_eval parse_head

                max_size cookie_jar conn_cache protocols_allowed

                protocols_forbidden protocols_redirectable)) {

    my $uc_lwp_arg = uc $lwp_arg;

    $lwp_args{ $lwp_arg } = $params->{ $uc_lwp_arg }

        if defined $params->{ $uc_lwp_arg };

}



$self->{ USERAGENT } = $ua = LWP::UserAgent->new(%lwp_args);
```

A busy web site using this provider might want to put a caching proxy between the application server and the server providing the templates (even with the caching, we still need to HEAD the URI to see if it has changed). Setting up LWP's proxy support is simple:

```
if (my $proxy = $params->{ PROXY }) {

    $ua->proxy('http', $proxy);

}



if (my $no_proxy = $params->{ NO_PROXY }) {

    $no_proxy = [ $no_proxy ] unless ref($no_proxy) eq 'ARRAY';

    $ua->no_proxy(@$no_proxy);

}
```

The NO_PROXY option defines domains for which LWP should not use the proxy.

If we're debugging the provider, we can turn on debugging in LWP as well, using LWP::Debug:

```
if ($self->{ DEBUG }) {

    require LWP::Debug;

    LWP::Debug::level('+');

}
```

And, for good measure, we uniquely identify this agent, so it can be specifically picked out by the logs:

```
$ua->agent(sprintf "%s [%s/%.02f]",

    $ua->_agent, ref($self), $VERSION);
```

Because we do not have a base filename to use when contructing paths for compiled versions of the templates, we need to have COMPILE_DIR set if COMPILE_EXT is set (otherwise, the provider will try to create directories in /; we'll see this in more detail when we discuss _fetch).

```
# IF COMPILE_EXT is set, COMPILE_DIR must also be set

my ($cdir, $cext) = @$params{ qw( COMPILE_DIR COMPILE_EXT ) };

if (length($cext) && ! length($cdir)) {
```

```
    return $self->error("COMPILE_DIR must be set if COMPILE_EXT is set");
}

    return $self;
}
```

The main method of our provider, fetch, can be much simpler than the default fetch:

```
sub fetch {
    my ($self, $name) = @_;

    my $uri = URI->new($name, "http");
    $uri->scheme("http");
```

When the context determines which provider to use, based on the PREFIX_MAP, the prefix is stripped off. The URI module will help us put that back in. (The other methods in Template::Provider::HTTP that are expecting URIs will actually be expecting URI objects.)

```
$self->debug("Got request for '$uri'") if $self->{ DEBUG };

    return $self->_fetch($uri);
}
```

Just like Template::Provider, we defer the hard work to the _fetch method. In our case, this is mainly for consistency with the default provider, because fetch is so simple.

_fetch is a little more complicated—it has to be aware of the cache and needs to know how to request a new copy of the template if the one we have is out of date. The LWP::UserAgent module knows how to handle conditional requests, so we can take advantage of that here:

```
sub _fetch {
    my ($self, $uri) = @_;
    my ($data, $error, $compiled, $request, $response);
    my $ua = $self->{ USERAGENT };
    my $now = time;

    $self->debug("_fetch($uri)") if $self->{ DEBUG };
```

_compiled_filename determines what the filename would be if we were writing the Perl versions of the templates to the disk-based cache. There are two reasons we do this: we need to know where to look to see whether we already have a compiled version of the templates, and we need to know where to write compiled versions of the templates.

```
$compiled = $self->_compiled_filename($uri);
```

The HTTP equivalent of stat is to HEAD the URI and check for freshness headers, such as Expires or Last-Modified:

```
# HEAD the URI, to see if we need to refetch it all
$request = HEAD($uri);
$response = $ua->request($request);
```

Once we have the headers for the request, we can check whether it is newer than the compiled version (if we have one):

```
if ($compiled && -f $compiled && $response->is_fresh &&
    (stat($compiled))[9] <= $response->fresh_until) {
    # The compiled version is alright; return it;
```

```
    $data = $self->_load_compiled($compiled);
    $error = defined $data
        ? STATUS_OK
        : $self->{ TOLERANT }
            ? STATUS_DECLINED
            : STATUS_ERROR;
}
```

_load_compiled is a standard Template::Provider method that reads a compiled version of a template from disk, requires it, and returns a compiled subroutine.

If the template fails to load, we need to set $error appropriately. (The context will treat $data as the error message if $error is not undefined.) The TOLERANT flag is a signal from the user that these errors should not be immediately fatal, so we return STATUS_DECLINED if TOLERANT is set, and return STATUS_ERROR otherwise.

```
else {
    # The compiled version either doesn't exist or is out of date
    $request = GET($uri);
    $response = $ua->request($request);

    if ($response->is_success) {
        $data = {
            name => "$uri",
            text => $response->content,
            time => int($response->fresh_until),
            load => time,
        };
        $error = STATUS_OK;

        ($data, $error) = $self->_compile($data, $compiled);
        ($data, $error) = $self->store($compiled, $data);
        $data = $data->{ data }
            unless $error;
    }
    else {
        $data = $response->error_as_HTML( );
        $error = $self->{ TOLERANT } ? STATUS_DECLINED : STATUS_ERROR;

    }
}

    return ($data, $error);
  }
```

_compiled_filename is pretty straightforward, and again, we can take advantage of the superclass's version:

```
sub _compiled_filename {
    my ($self, $uri) = @_;

    # This adds '/' to the list of characters not encoded; we want those
    # so that we can make nested directories in which to store cache files.
    $uri = uri_escape($uri->opaque, "^A-Za-z0-9\-_.!~*'( )/");

    return File::Spec->canonpath($self->SUPER::_compiled_filename($uri));
}
```

This method turns an opaque (schemeless) URI such as //templates.tt2.org/config into a filename such as *//templates.tt2.org/config*. Template::Provider::_compiled_filename appends this to the value of COMPILE_DIR, so it ends up somewhere we can write (because you're not running this as the superuser, of course). Finally, File::Spec->canonpath *canonicalizes* the filename, which in this case means removing duplicate forward slash (/) characters. The / character had to be added to the list of characters not escaped by uri_escape, or we would have ended up with a filename such as *%2F%2Ftemplates.tt2.org%2Fconfig*, which is pretty ugly. With the slashes in place, we end up with a nested filesystem structure for our cache directory, which is easily navigable both by the curious developer and the provider as it walks the filesystem looking for compiled files. As a side effect, because we are not doing anything to prevent the escaping of the query string parameters, they become part of the compiled filename—invocations of the same URI but with different query strings will result in different cache files.

Using this new provider is easy:

```
my $http = Template::Provider::HTTP->new( );
my $prov = Template::Provider->new( );

my $tt = Template->new({
    LOAD_TEMPLATES => [
        $prov,
        $http,
    ],
    PREFIX_MAP => {
        http => 1,
        default => 0,
    }
});
```

As mentioned earlier, PREFIX_MAP is necessary to give the context a hint about which provider to use. We use the normal Template::Provider object by default, but for HTTP templates, use the HTTP provider:

```
[%
    PROCESS 'http://use.perl.org/journal.pl?uid=18&content_type=rss' |
        redirect('davorg.xml');
    USE davorg = XML.RSS('davorg.xml');
    FOREACH item IN davorg.items %]
    * [% item.title %]
        * [% item.link;
```

```
    END;
-%]
```

Example 8-1 is the complete Template::Provider::HTTP.

Example 8-1. Template::Provider::HTTP

```
package Template::Provider::HTTP;


use strict;
use vars qw($VERSION);
use base qw(Template::Provider);


$VERSION = 1.00;


use File::Spec;
use HTTP::Request::Common qw(HEAD GET);
use LWP::UserAgent;
use Template::Constants qw(:status);
use Template::Provider;
use URI;
use URI::Escape qw(uri_escape);


# ----------------------------------------------------------------------
# fetch($name)
#
# Retrieve the template identified by $name.  The PREFIX_MAP ensures
# that this gets called only when appropriate.
# ----------------------------------------------------------------------
sub fetch {
    my ($self, $name) = @_;


    # The Context's prefix handling strips out the 'http:', so we
    # need to add it back in.
    my $uri = URI->new($name, "http");
    $uri->scheme("http");


    $self->debug("Got request for '$uri'") if $self->{ DEBUG };


    return $self->_fetch($uri);
```

```
}

# ---------------------------------------------------------------------
# fetch($name)
#
# Uses LWP::UserAgent to fetch a template referenced via http://...,
# and then uses standard Template::Provider methods to compile,
# cache, and so on.
# ---------------------------------------------------------------------
sub _fetch {
    my ($self, $uri) = @_;
    my ($data, $error, $compiled, $request, $response);
    my $ua = $self->{ USERAGENT };

    $self->debug("_fetch($uri)") if $self->{ DEBUG };

    $compiled = $self->_compiled_filename($uri);

    # HEAD the URI, to see if we need to refetch it all
    $request = HEAD($uri);
    $response = $ua->request($request);

    if ($compiled && -f $compiled && $response->is_fresh &&
        (stat($compiled))[9] <= $response->fresh_until) {
        # The compiled version is alright; return it;

        $data = $self->_load_compiled($compiled);
        $error = defined $data
            ? STATUS_OK
            : $self->{ TOLERANT }
                ? STATUS_DECLINED
                : STATUS_ERROR;
    }
    else {
        # The compiled version either doesn't exist or is out of date
        $request = GET($uri);
        $response = $ua->request($request);

        if ($response->is_success) {
```

```
            $data = {
                name => "$uri",
                text => $response->content,
                time => int($response->fresh_until),
                load => time,
            };
            $error = STATUS_OK;

            ($data, $error) = $self->_compile($data, $compiled);
            ($data, $error) = $self->store($compiled, $data);
            $data = $data->{ data }
                unless $error;
        }
        else {
            $data = $response->error_as_HTML( );
            $error = $self->{ TOLERANT } ? STATUS_DECLINED : STATUS_ERROR;
        }
    }

    return ($data, $error);
}

# ---------------------------------------------------------------------
# _compiled_filename($uri)
#
# Transforms the URI into a filename.
# ---------------------------------------------------------------------
sub _compiled_filename {
    my ($self, $uri) = @_;

    # This adds '/' to the list of characters not encoded; we want those
    # so that we can make nested directories in which to store cache files.
    $uri = uri_escape($uri->opaque, "^A-Za-z0-9\-_.!~*'( )/");

    return File::Spec->canonpath($self->SUPER::_compiled_filename($uri));
}

# ---------------------------------------------------------------------
# _init(\%params)
```

```
#
# This is here primarily to initialize the LWP::UserAgent instance.
# -------------------------------------------------------------------
sub _init {
    my ($self, $params) = @_;
    my ($ua, %lwp_args, $lwp_arg);

    $self->SUPER::_init($params);

    for $lwp_arg (qw(agent from timeout use_eval parse_head
                     max_size cookie_jar conn_cache protocols_allowed
                     protocols_forbidden protocols_redirectable)) {
        my $uc_lwp_arg = uc $lwp_arg;
        $lwp_args{ $lwp_arg } = $params->{ $uc_lwp_arg }
            if defined $params->{ $uc_lwp_arg };
    }

    $self->{ USERAGENT } = $ua = LWP::UserAgent->new(%lwp_args);

    if (my $proxy = $params->{ PROXY }) {
        $ua->proxy('http', $proxy);
    }

    if (my $no_proxy = $params->{ NO_PROXY }) {
        $no_proxy = [ $no_proxy ] unless ref($no_proxy) eq 'ARRAY';
        $ua->no_proxy(@$no_proxy);
    }

    if ($self->{ DEBUG }) {
        require LWP::Debug;
        LWP::Debug::level('+');
    }

    $ua->agent(sprintf "%s [%s/%.02f]",
        $ua->agent, ref($self), $VERSION);

    # IF COMPILE_EXT is set, COMPILE_DIR must also be set
    my ($cdir, $cext) = @$params{ qw( COMPILE_DIR COMPILE_EXT ) };
    if (length($cext) && ! length($cdir)) {
```

```
        return $self->error("COMPILE_DIR must be set if COMPILE_EXT is set");
    }

    return $self;
}

1;
```

8.1.2 Restricting Access to Plugins

By default, all of the Template Toolkit's plugins are available to every template. Sometimes it makes sense to limit the available plugins, such as in a web-hosting or education situation. For these cases, restricting which plugins are available is useful.

Again, we can use the chain of responsibility to our advantage. By creating a Template::Plugins provider that governs access to plugins, we can ensure that only allowed plugins are loaded.

As you recall, the context interacts with the plugin providers by calling its fetch method, which is expected to return a plugin, or (undef, $error) if the plugin could not be loaded. Because the purpose of this plugin is to allow access only to specific plugins, it needs only to implement fetch, and doesn't have to do much more than simply decline to handle requests for allowed plugins by returning STATUS_DECLINED. If a plugin provider declines to handle a request, the context will move on the next provider in line or throw an exception if no more providers are available.

Here is the complete Template::Plugins::Allow:

```
package Template::Plugins::Allow;

use strict;
use Template::Constants qw(:status);

sub new {
    my $class = shift;
    bless { map { ($_, 1) } @_ }, $class;
}

sub fetch {
    my $self = shift;
    my $name = shift;
    return $self->{ $name }
        ? (undef,  STATUS_DECLINED)
        : ("access to $name not allowed", STATUS_ERROR);
}

1;
```

This provider is initialized with the names of the plugins that are allowed. We also need the regular plugins provider, to actually load the allowed plugins:

```
my $allow    = Template::Plugins::Allow->new(qw( Date Table ));
my $plugins  = Template::Plugins->new( );
```

Then we define the LOAD_PLUGINS chain of command with the Allow provider first:

```
my $tt = Template->new({
    LOAD_PLUGINS => [ $allow, $plugins ]
});
```

If the plugin is allowed, the Allow provider returns STATUS_DECLINED and control passes to the regular plugins provider. Otherwise, the Allow provider returns an error.

Here it is in use:

```
[% TRY;
     USE Date;
     "got date\n";
   CATCH;
     "not date: $error\n";
   END;

   TRY;
     USE Table([1, 2, 3]);
     "got table\n";
   CATCH;
     "not table: $error\n";
   END;

   TRY;
     USE Format;
     "got format";
   CATCH;
     "not format: $error\n";
   END;
%]
```

Here's the output:

```
got date
got table
not format: plugin error - access to Format not allowed
```

8.1.3 A chrooted Provider

By default, the Template Toolkit doesn't allow inclusion of files using absolute paths. This is to help disallow malicious or inexperienced users from including potentially sensitive files in output:

```
[% INSERT /etc/aliases %]
```

Sometimes, however, allowing absolute files does make sense. For example, you might want to specify the absolute path to a template to ensure that the INCLUDE_PATH doesn't supply you with a different template that happens to have the same name as the one you want. In these cases, it would be nice to be able to provide a limited directory structure for the templates to access. Normally, an entire process would be run in a *chrooted jail*, which means that the entire process (in this case, the Perl interpreter that is processing the templates via the Template Toolkit) would have a limited view of the underlying filesystem. (chroot is the name of the Unix system call that implements this functionality, and so has become synonymous with the activity.) This can be problematic, however; because everything that the Perl interpreter needs would need to be present in this limited filesystem, including system libraries, this means copying a lot of files around.

However, we can implement a Template::Provider subclass that has a limited view of the filesystem, by superficially emulating what chroot does: we can simply prepend a specific root (we'll call it CHROOT_BASE) to every absolute filename passed to INCLUDE, PROCESS, and INSERT. Then, a request such as:

```
[% INSERT /etc/aliases %]
```

would be translated into a request for */var/www/etc/aliases* (assuming a CHROOT_BASE of */var/www*).

We can build upon Template::Provider—we are modifying the default behavior only slightly. File::Spec::Functions provides a clean, function-oriented interface to File::Spec, while still preserving File::Spec's "cross-platform-y" goodness:

```
package Template::Provider::Chroot;



use strict;

use vars qw($VERSION);

use base qw(Template::Provider);



$VERSION = 1.00;



use File::Spec::Functions qw(canonpath catfile file_name_is_absolute);
```

We'll pull the CHROOT_BASE parameter out of the configuration, and then let Template::Provider::_init take over handling the rest of the parameters:

```
sub _init {

    my ($self, $params) = @_;

    $self->{ CHROOT_BASE } = $params->{ CHROOT_BASE } || "";



    return $self->SUPER::_init($params);

}
```

We need to override only the fetch method, and even then we need to do something only when the requested template is an absolute filename:

```
sub fetch {

    my ($self, $name) = @_;

    my $chroot = $self->{ CHROOT_BASE };

    my $newname = $name;



    if ($chroot && file_name_is_absolute($name)) {

        $newname = canonpath(catfile($chroot, $name));

        $self->debug("Using path of '$newname' instead of '$name'")

            if $self->{ DEBUG };

    }



    return $self->SUPER::fetch($newname);

}
```

One happy side effect of the method this provider uses is that if a template cannot be found, the error that the context emits references the original template name, not the adjusted filename.

Because this provider falls through to the behavior of the default provider, we don't need to use an array of providers or set up a PREFIX_MAP. We can simply tell Template::Config to use our new class instead of the default provider:

```
use Template;
use Template::Config;

$Template::Config::PROVIDER = 'Template::Provider::Chroot';
```

and continue as normal.

Example 8-2 shows the complete Template::Provider::Chroot.

Example 8-2. Template::Provider::Chroot

```
package Template::Provider::Chroot;

use strict;
use base qw(Template::Provider);

use File::Spec::Functions qw(canonpath catfile file_name_is_absolute);
use Template::Provider;

sub fetch {
    my ($self, $name) = @_;
    my $chroot = $self->{ CHROOT_BASE };
    my $newname = $name;

    if ($chroot && file_name_is_absolute($name)) {
        $newname = canonpath(catfile($chroot, $name));
        $self->debug("Using path of '$newname' instead of '$name'")
            if $self->{ DEBUG };
    }

    return $self->SUPER::fetch($newname);
}

sub _init {
    my ($self, $params) = @_;

    $self->{ CHROOT_BASE } = $params->{ CHROOT_BASE } || "";
```

```
    return $self->SUPER::_init($params);
}


1;
```

These few simple examples should be enough to get you started extending the Template Toolkit to do your bidding.

8.2 Creating Filters

Chapter 5 introduced Template Toolkit filters. This section explains how to write your own filters.

There are two types of filters: *static* and *dynamic*. A static filter is one that always operates the same way, and a dynamic filter is one that can be configured differently for each invocation. From within templates, they are invoked almost identically, with the exception that dynamic filters can take arguments, while static filters cannot.

8.2.1 Static Filters

Internally, filters are implemented as references to subroutines; when invoked, these subroutines are passed the text to be filtered as a string, and are expected to return a string. Defining a static filter is as simple as creating a subroutine and declaring it in the FILTERS configuration option (it can also be installed into the context with the define_filter method). All invocations of a static filter will use the same subroutine reference, which won't be passed any parameters other than the text to be filtered. Standard filters such as html and lower are examples of static filters.

Here is a simple Perl subroutine, designed to be used as a static filter, which rot13s text:[2]

> [2] rot13 is a simple, well-known substitution cipher, in which each character in a string of text is replaced by the character 13 positions away. For example, a becomes n, b becomes o, and so on. Passing a string through rot13 two times restores the original string.

```
sub rot13 {
    my $text = shift;
    $text =~ tr/a-zA-Z/n-za-mN-ZA-M/;
    return $text;
}
```

Once our rot13 subroutine has been defined, it can be installed in the processing context by passing a subroutine reference to the Template constructor:

```
my $tt = Template->new({
    FILTERS => {
        'rot13' => \&rot13,
    },
});
```

Using our rot13 filter is easy:

```
[% FILTER rot13 %]
Gur juvgr mbar vf sbe ybnqvat naq haybnqvat bayl.
[% END %]
```

The preceding code produces, naturally:

```
The white zone is for loading and unloading only.
```

And that's most of what there is to static filters: define a subroutine that expects one text argument, munges that argument in some way, and returns the output. The processing can be arbitrarily complex, and of course the text returned can be anything at all, or even nothing.

8.2.2 Dynamic Filters

The FILTER directive is expecting a reference to a subroutine that will be invoked with its text. For static filters, this subroutine reference was installed by the FILTERS or LOAD_FILTERS options when the Template instance was created. However, because the parameters of a dynamic filter might not be known until runtime, they must be treated differently. Dynamic filters are installed differently than static filters (via the FILTERS call), and the context knows to invoke them differently. Installing a dynamic filter at constructor time looks like this:

```
my $tt = Template->new({
    FILTERS => {
        'rot13'  => \&rot13, # our trusty static filter
        'censor' => [ \&censor_factory, 1 ], # our dynamic filter
    },
});
```

As you can see, dynamic filters are installed as array references, where the first element is a code reference and the second is a flag: 1 for dynamic, 0 for static. Analogously, static filters can be installed as:

```
FILTERS => {
    'rot13' => [ \&rot13, 0 ],
},
```

which explicitly marks it as a static filter.

When a dynamic filter is fetched, it is expected to return a reference to a subroutine, which is what the FILTER directive is expecting. The subroutine that is called and expected to return another subroutine to FILTER is called a *factory*.

Let's look at censor_factory, referred to earlier.

```
sub censor_factory {
    my ($context, $letter) = @_;

    return sub {
        my $text = shift;
        $text =~ s/($letter)/"*" x length($1)/eg;
        return $text;
    }
}
```

When called as:

```
[% text FILTER censor("a") %]
```

each a in $text will be replaced with *. When called as:

```
[% text FILTER censor("lemon") %]
```

each lemon in $text will be replaced with *****, and so on. Note that the arguments to censor—a and lemon—need to be given to censor_factory, which uses them to create a closure. This closure is then passed to FILTER, which invokes the subroutine and then discards it. If the dynamic filter is going to be reused, with the same arguments, it can be assigned to a variable:

```
[% text | no_lemons = censor("lemon") %]

[% more_text | no_lemons %]
```

The second invocation of no_lemons behaves identically to the first.

censor_factory is invoked with the Template::Context object as its first argument, and any other arguments as the rest of @_. Named parameters are folded into a hash reference and passed as the last argument, as is usual for invoked subroutines within templates. The factory subroutine should take into account the number and type of arguments it is expecting. Filters are free to ignore any or all of these arguments, of course.

We can redefine censor_factory to accept configuration parameters this way:

```
sub censor_factory {
    my ($context, @args) = @_;
    my $args = ref($args[-1]) eq 'HASH' ? pop @args : { };
```

```
    my $repl = $args->{'replacement'} || "*";


    return sub {
        my ($text, $letter) = @_;
        $text =~ s/($letter)/$repl x length($1)/eg;
        return $text;
    }
}
```

The key is @args: if there are any named parameters, they will be collected and passed, as a reference to a hash, as the last element of @_. These are popped off @args and assigned to hash references $args, from which we extract the replacement key (or a default of *, to make it backward compatible with our earlier version of censor_factory).

Now, we can call censor with a configurable replacement character:

```
[% text | censor("lemon", replacement = "#") %]
```

And each occurrence of the string lemon will be replaced with #####. Because the Template Toolkit rearranges named parameters to be passed last, our filter can be called with replacement replacement anywhere in the argument list, with identical results:

```
[% text | censor(replacement = "#", "lemon") %]
```

It it possible to pass arguments to static filters, but they are ignored:

```
[% FILTER rot13(all_caps = 1) %]
Gur juvgr mbar vf sbe ybnqvat naq haybnqvat bayl.
[% END %]


The white zone is for loading and unloading only.
```

The Template Toolkit ignores parameters passed to items that are not expecting them: because the presentation language is implementation neutral, a template has no way of knowing whether this filter can take arguments.

8.2.3 Template::Plugin::Filter

The Template::Plugin::Filter module, which allows for filters to be written and treated as plugins, is a bit of an odd beast—it is actually a plugin, but is designed to be used as a filter:

```
[% USE myfilt = MyFilter %]
[% FILTER $myfilt %]
    ...
[% END %]
```

Using Template::Plugin::Filter to write filters is more akin to writing plugins than to writing filters, with one major difference: when the variable is used as a filter, a method named filter is invoked. All of our filter examples can be turned into Template::Plugin::Filter objects by renaming the subroutine to filter and putting it into its own class, which inherits from Template::Plugin::Filter:

```
package TTBook::Template::Plugin::Rot13;


use strict;
use base qw(Template::Plugin::Filter);


sub filter {
    my $text = shift;
```

```
    $text =~ tr/a-zA-Z/n-za-mN-ZA-M/;
    return $text;
}
```

Now our rot13 filter can be used like so:

```
[% USE encryptor = Rot13 %]
[% text | $encryptor %]
```

Note that you must explicitly dereference the plugin filter using the $encryptor format; this is key!

8.2.4 Writing New Filters

As we have seen, a filter is a subroutine reference that can be invoked from within the processing context. There are many mature and full-featured modules on CPAN that filter text. Often, you will need the functionality of one of these modules within your templates, and filters are the easiest way to glue the two together. We cover some of these modules next.

8.2.4.1 Digest::MD5

The Digest::MD5 module creates a *message digest* of text or files. According to the manpage:

> *The "Digest::MD5" module allows you to use the RSA Data Security Inc. MD5 Message Digest algorithm from within Perl programs. The algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input.*

This makes a good candidate for a filter. We could use the MD5 filter from within *ttree* to generate our checksum files:

```
[% USE dir = Directory(".");
   FOREACH file = dir.files;
    checksum = INSERT $file.name | md5 %]
    * [% file.name %] = [% checksum %]
[% END %]
```

Digest::MD5 exports a function called md5_hex that does exactly what we are looking for. Our md5 static filter is simple:

```
use Digest::MD5 qw(md5_hex);
sub md5 {
    my $text = shift;
    return md5_hex($text);
}
```

This static filter is so simple that it is possible to inline it with almost no loss of clarity:

```
use Digest::MD5 qw(md5_hex);


my $tt = Template->new(
    FILTERS => {
        "md5" => sub { my $text = shift; return md5_hex($text); },
    },
);
```

8.2.4.2 Text::Bastardize

Text::Bastardize is a great little module for manipulating text. It has methods for transformations to pig Latin, numerical abbreviation, and k3wlt0k, among others.

Using Text::Bastardize is simple:

```
use Text::Bastardize;


my $bastard = Text::Bastardize->new;

$tb->charge($data);


print $tb->rev;
```

The various methods return arrays, which in general is appropriate when dealing with text, but we'll need strings; join is our friend:

```
print join "", $tb->rev;
```

The methods Text::Bastardize makes available include the following:

rdct

"Reduce" text:

```
$tb->charge("The white zone is for loading and unloading only.");

$tb->rdct( );

# the whte z1 is fr ladng nd unladng only.
```

pig

Transform text into pig Latin:

```
$tb->charge("You need a thneed!");

$tb->pig( );

# youay eednay away eedthnay!
```

rot13

Hey, this looks familiar:

```
$tb->charge("with or without is the different.");

$tb->rot13( );

# jvgu be jvgubhg vf gur qvssrerag
```

k3wlt0k

Transforms your text into its "elite" form:

```
$tb->charge("You'll love it, it's a way of life");

$tb->k3wlt0k( );

# JUR11 10V4 17, 17Z 3 W3Y 0F 11F4
```

rev

Reverses your text:

```
$tb->charge("A thing of beauty is a joy forever.")
$tb->rev( );
# .reverof yoj a si ytuaeb fo gniht A
```

n20e

Replaces long words (more than six characters) with numeric equivalents:

```
$tb->charge("Every nonzero finite dimensional inner " .
            "product space has an orthonormal basis."
$tb->n20e( );
# Every n5o finite d9l inner p5t space has an o9l basis.
```

Turning these Text::Bastardize methods into filters is relatively straightforward:

```
use Template;

my $tt = Template->new(
   FILTERS => {
      "rdct" => \&rdct,
      "n20e" => \&n20e,
   },
);

sub rdct {
   my $text = shift;

   my $tb = Text::Bastardize->new;
   $tb->charge($text);

   return join "", $tb->rdct;
}

sub n20e {
   my $text = shift;

   my $tb = Text::Bastardize->new;
   $tb->charge($text);

   return join "", $tb->rdct;
}
```

And so on. Each Text::Bastardize method follows the same general pattern:

```
my $tb = Text::Bastardize->new;
$tb->charge($data);

return join "", $tb->METHOD;
```

This means that we can produce these subroutines automatically, with a factory function:

```
sub bastardize_factory {
    my $type = shift || "rot13";

    return sub {
        my $text = shift;

        my $tb = Text::Bastardize->new;
        $tb->charge($text);

        return join "", $tb->$type( );
    };
}

my $tt = Template->new(
    FILTERS => {
        "rdct" => bastardize_factory("rdct"),
        "n20e" => bastardize_factory("n20e"),
    },
);
```

This is exactly what is needed to create dynamic filters; we can make bastardize available to our templates as a dynamic filter:

```
my $tt = Template->new(
    FILTERS => {
        "bastardize" => [ \&bastardize_factory, 1 ]
    },
);
```

The bastardize dynamic filter would be used with an argument:

```
[% FILTER bastardize("n20e") %]
Numeric abbreviation.
[% END %]
```

The filter subroutine created by calling bastardize(TYPE) can be captured for later use, by assigning it to a variable:

```
[% FILTER rot13 = bastardize("rot13") %]

Grzcyngr Gbbyxvg Ehyrf

[% END %]


[% text | rot13 %]
```

As you will recall, dynamic filters get called with a Template::Context instance as their first argument. bastardize_factory needs to deal with this:

```
sub bastardize_factory {

    shift( ) if ref $_[0];
```

If the first argument is a reference, it is not the type that we are expecting; therefore, we can shift it away. bastardize_factory, in its entirety, is pretty simple:

```
sub bastardize_factory {

    shift if ref $_[0];

    my $type = shift;

    my $tb = Text::Bastardize->new;


    return sub {

        my $text = shift;


        $tb->charge($text);

        return join "", $tb->$type;

    };

}
```

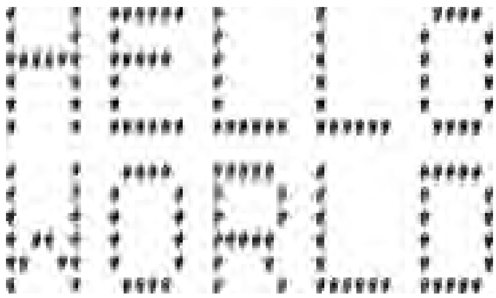
And, of course, we can have both the static and dynamic versions of our bastardize filters in our Template::Filters instance:

```
my $tt = Template->new(

    FILTERS => {

        rdct       => [   bastardize_factory("rdct"),    0 ],

        pig        => [   bastardize_factory("pig"),     0 ],

        k3wlt0k    => [   bastardize_factory("k3wlt0k"), 0 ],

        rot13      => [   bastardize_factory("rot13"),   0 ],

        rev        => [   bastardize_factory("rev"),     0 ],

        n20e       => [   bastardize_factory("n20e"),    0 ],

        bastardize => [ \&bastardize_factory,            1 ],

    },

);
```

8.2.4.3 Text::FIGlet

FIGlet is a program for making large letters out of ordinary, unexpecting text, and Text::FIGlet (http://www.figlet.org/) is a Perl implementation. *FIGlet* is akin to the Unix program banner, which formats a message for printing on a line printer (see Figure 8-1).

Figure 8-1. "Hello world" created by the Unix program banner



FIGlet does something similar, but adds font capability kerning, and the ability to make your text face in the correct direction. The default font looks like Figure 8-2.

Figure 8-2. "Hello world" created by FIGlet (using the default font)

But there are hundreds of other fonts, such as *rozzo* (see Figure 8-3).

Figure 8-3. The rozzo font in FIGlet

The possibilities here are staggering, of course.

Using Text::FIGlet is easy:

```
use Text::FIGlet;

my $figgy = Text::FIGlet->new(-f => $fontname);



print $figgy->figify(-A => $text);
```

Turning this into a dynamic filter is straighforward: we need to handle the various -X constructor parameters, one of which is a scalar containing the text to be figified. Hey, we have one of those:

```
sub figify_filter_factory {

    my ($context, @args) = @_;

    my $args = ref($args[-1]) eq 'HASH' ? pop @args : { };

    my $figgy = Text::FIGlet->new(%$args);



    return sub {

        my $text = shift;

        $figgy->figify(-A => $text);
```

```
    }
}
```

Using this figify filter feels a little unnatural, however, mainly due to the strange-looking format of the constructor parameters:

```
[% FILTER figify("-f" => "acrobatic") %]
Hello, world!
[% END %]
```

We can provide intuitive mappings for these in our implementation:

```
# some nice aliases...
my %fig_params = (
    "german"        => "-D",
    "fontdir"       => "-d",
    "fontfile"      => "-f",
    "smushmode"     => "-m",
    "direction"     => "-X",
    "justification" => "-x",
    "width"         => "-w",
);
# ...and some even nicer aliases
$fig_params{'font'} = $fig_params{'fontfile'};
$fig_params{'dir'} = $fig_params{'fontdir'};

sub figify_filter_factory {
    my ($context, @args) = @_;
    my $args = ref($args[-1]) eq 'HASH' ? pop @args : { };
    my %cons_args;

    for my $a (%$args) {
        my $p = $fig_params{ $a };
        $cons_args{ $p } = $args->{ $a } if defined $p;
    }

    my $figgy = Text::FIGlet->new(%cons_args);

    return sub {
        my $text = shift;
        $figgy->figify(-A => $text);
    }
}
```

Now our figified templates look a little more like other templates:

```
[% FILTER figify(font => "cosmic") %]

Hello, world!

[% END %]
```

The output is shown in Figure 8-4.

Figure 8-4. "Hello world" using a dynamic filter in FIGlet

8.2.4.4 Normalizing HTML: HTML::Clean

The HTML::Clean module encapsulates a number of common techniques for minimizing the size of HTML output: removing unnecessary whitespace, comments, and META tags; replacing longer tags with shorter ones; and removing empty unnecessary tags. HTML::Clean normally operates in filter mode, which makes it an ideal filter. HTML::Clean is available from http://search.cpan.org/dist/HTML-Clean/.

The "clean level" and types of cleaning that HTML::Clean does are controlled by options passed to strip, so HTML::Clean is a good candidate for a dynamic filter:

```
use HTML::Clean;


sub clean {

    my ($context, @args) = @_;

    my $config = ref($args[-1]) eq 'HASH' ? pop @args : { };


    return sub {

        my $text = shift;


        my $h = HTML::Clean->new(\$text);


        $h->level($config->{'level'})

            if (defined $config->{'level'});


        $h->strip($config);


        return ${ $h->data };

    };

}


my $tt = Template->new(FILTERS => { clean => [ \&clean, 1 ] });
```

This makes a good overall filter:

```
[% BLOCK page %]
[% FILTER clean(level = 9) %]
<html>
  <head>
    <title>[% template.title %]
  </head>
  <body>
[% content %]
  </body>
</html>
[% END %]
[% END %]

[% WRAPPER page %]
    ...
```

Using Subroutine References as Filters

Because filters are "just" subroutine references, and the Template Toolkit allows for subroutine references to be passed as values in the second parameter to process, you might be thinking that we should be able to rephrase our filter examples as:

```
my %filters = (
    'rot13' => \&rot13,
    'censor' => \&censor_factory,
);
my $t = Template->new( );
$t->process($file, \%filters);
```

The answer, of course, is, yes, there's more than one way to do it. However, this method requires that your filters be called as:

```
[% rot13(text) %]
[% censor(text) %]
```

Because "real" filters can be called using the FILTER or | notation, you lose the ability to pipe PROCESS and INCLUDE calls through your subroutine:

```
[% rot13(INCLUDE encrypted.txt) %]
```

Therefore, the previous code produces a parser error. Using an intermediate variable is an option, of course:

```
[% enc = INCLUDE encrypted.txt; rot13(enc); %]
```

But that's no fun.

These examples, by the way, produce something like:

```
Gur juvgr mbar vf sbe ybnqvat naq haybnqvat bayl.
CODE(0x83a85c4)
```

which, in the second case, is not what we wanted. Dynamic filter factories, which return subroutine references, need to be handled differently:

```
$filters{'censor_a'} = censor_factory("a");

$filters{'censor_b'} = censor_factory("b");
```

And so on, which has obvious ramifications in the template. In these cases, dynamic filters have to be rewritten to return text, and not a code reference:

```
sub censor {

    my ($text, $letter) = @_;

    $text =~ s/($letter)/"*" x length($1)/eg;

    return $text;

}
```

8.3 Creating Plugins

As we saw in Chapter 6, a plugin is implemented as an object-oriented Perl module. This module must implement a few basic methods in order for the context to load it correctly, and all of these methods can be inherited from the Template::Plugin module; otherwise, a plugin can be very free form.

8.3.1 The Template::Plugin Module

The Template::Plugin module both defines the plugin API and serves as a base class for plugin implementations. By default, a Template::Plugin instance has almost no functionality, other than to load correctly.

Template::Plugin defines three methods: load, new, and error. Subclasses are free to override any of these methods, or implement any others they might need to perform their duties.

load($context)

This method is called by Template Toolkit when the plugin module is first loaded. It is called as a package method and thus implicitly receives the package name as the first parameter. A reference to the Template::Context object loading the plugin is also passed. The default behavior for the load method is to simply return the class name; the calling context then uses this class name to call the new package method:

```
package MyPlugin;

sub load {              # called as MyPlugin->load($context)

    my ($class, $context) = @_;

    return $class;      # returns 'MyPlugin'

}
```

new($context, @params)

This method is called to instantiate a new plugin object for the USE directive. It is called as a package method against the class name returned by load. A reference to the Template::Context object creating the plugin is passed, along with any additional parameters specified in the USE directive:

```
sub new {               # called as MyPlugin->new($context)

    my ($class, $context, @params) = @_;

    bless {

        _CONTEXT => $context,

        _PARAMS => \@params,

    }, $class;          # returns blessed MyPlugin object

}
```

error($error)

This method, inherited from the Template::Base module, is used for reporting and returning errors. It can be called as a package method to set/return the $ERROR package variable, or as an object method to set/return the object's _ERROR member. When called with an argument, it sets the relevant variable and returns undef. When called without an argument, it returns the value of the variable.

```
sub new {

    my ($class, $context, $dsn) = @_;

    return $class->error('No data source specified')

        unless $dsn;

    bless {
```

```
        _DSN => $dsn,
    }, $class;
}
...
my $something = MyModule->new( )
    || die MyModule->error( ), "\n";
$something->do_something( )
    || die $something->error( ), "\n";
```

The Template::Context object that handles the loading and use of plugins calls the new and error methods against the package name returned by the load method. In pseudocode terms, it might look something like this:

```
$class  = MyPlugin->load($context);      # returns 'MyPlugin'

$object = $class->new($context, @params)  # MyPlugin->new(...)
    || die $class->error( );              # MyPlugin->error( )
```

The load method may alternately return a blessed reference to an object instance. In this case, new and error are then called as *object* methods against that prototype instance.

Example 8-3 is the complete TTBook::Template::Plugin::Printer plugin, which implements a print service.

Example 8-3. TTTBook::Template::Plugin::Printer

```
package TTBook::Template::Plugin::Printer;

use strict;
use vars qw($PRINTER $SERVER);
use base qw(Template::Plugin);

use Template::Plugin;
use Template::Exception;
use Net::Printer;

$PRINTER = "jeckyl";
$SERVER = "mr-hyde";

sub load {
    my ($class, $context) = @_;
    my $printer = Net::Printer->new(printer => $PRINTER,
                                    server  => $SERVER);
    my $self = bless {
        _CONTEXT => $context,
        _PRINTER => $printer,
```

```
    }, $class;

    return $self;
}

sub new {
    my ($self, $context) = @_;
    return $self;
}

sub print {
    my ($self, $data) = @_;
    my ($printer, $context) = @$self{ qw( _PRINTER _CONTEXT) };

    my $result = $printer->printstring($data);
    $context->throw('printer', $result)
        unless (int($result) =  = 1);

    return "";
}

1;
```

In this example, we implemented a *Singleton* plugin. One object gets created when load is called; the object simply returns itself for each call to new.

When the plugin is loaded, a TTBook::Template::Plugin::Printer instance is created; each call to new is called against this object, which instantiates and returns that same instance.

Because calls to print throw printer exceptions if there is a problem, they should be wrapped in TRY / CATCH blocks:

```
[% USE Printer %]
[% TRY %]
    [% Printer.print(data) %]
[% CATCH printer %]
    There was an error printing: [% error %]
[% END %]
```

print explicitly returns an empty string so that there is no unwanted output in the template.

8.3.2 Installing Functions into the Stash from Within a Plugin

While plugins are implemented as object-oriented modules, there is no reason that every plugin has to be used in an object-oriented way. Because a plugin is invoked with $context as an argument, a plugin writer can elect to install functions in the stash in addition to returning an object designed to be used:

```
package TTBook::Template::Plugin::Red;

use strict;
use base qw(Template::Plugin);

sub new {
    my ($class, $context) = @_;
    my $stash = $context->stash;

    $stash->set('red', \&make_red);

    return sub { make_red(@_) };
}

sub make_red {
    my $text = shift;
    return qq|<font color="red">$text</font>|;
}

1;
```

The plugin still needs to return a blessed object, but it will probably be ignored. This plugin would be used like this:

```
[% USE Red %]

Hello, [% red('World') %]
```

However, because we've chosen to return a subroutine reference, the plugin name can also be used, to the same effect:

```
[% USE colorizer = Red %]

Hello, [% red('red world!') %]
I am [% colorizer('also red') %].
```

This example, while silly, illustrates two important points. First, once a plugin has a reference to the stash, arbitrary functionality can be added to your templates. Second, a plugin need merely return something that Perl considers true—it doesn't have to be a blessed object.

Instead of make_red, we could have created an incrementing counter:

```
my $count = 0;
$stash->set('counter' => sub { ++$count });
```

Each time counter is invoked, it returns the next number:

```
[% FOREACH [ 1 .. 10 ] %]
    * [% counter %]
[% END %]
```

As such, the previous code returns:

```
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
```

By making new( ) accept an argument, we can seed the counter:

```
sub new {
    my ($class, $context, $start) = @_;
    my $stash = $context->stash;

    my $count = int($start || 0);
    $stash->set('counter' => sub { ++$count });

    bless { } => $class;
}
```

This counter will start where we tell it to:

```
[% USE Counter(100) %]
[% counter %]
```

As such the previous code yields:

```
101
```

Example 8-4 is the complete TTBook::Template::Plugin::Counter.

Example 8-4. TTBook::Template::Plugin::Counter

```
package TTBook::Template::Plugin::Counter;

use strict;
use vars qw($VERSION);
use base qw(Template::Plugin);

sub new {
    my ($class, $context, $start) = @_;
    my $stash = $context->stash;

    my $count = int $start;
```

```
    $stash->set("counter" => sub { ++$count });

    bless { } => $class;
}

1;
```

8.3.3 Defining Filters from Within a Plugin

Earlier, we saw how the define_filter( ) method can be called against the $context object to define new filters. Let's look at a plugin that does this.

Let's revisit our Digest::MD5 filter and install it from within a plugin. Recall that the body of the filter was a very simple subroutine:

```
use Digest::MD5 qw(md5_hex);
sub md5 {
    my $text = shift;
    return md5_hex($text);
}
```

Installing a plugin into the current stash is something that should be done when the module is loaded, so load is an ideal place for it:

```
sub load {
    my ($class, $context) = @_;
    $context->define_filter('md5', \&md5);
    return $class;
}
```

Example 8-5 is the complete $namespace::Template::Plugin::MD5.

Example 8-5. $namespace::Template::Plugin::MD5

```
package TTBook::Template::Plugin::MD5;

use strict;
use vars qw($VERSION);
use base qw(Template::Plugin);

use Template::Plugin;
use Digest::MD5 qw(md5_hex);

$VERSION = 1.01;

sub md5 {
    my $text = shift;
```

```
    return md5_hex($text);
}

sub load {
    my ($class, $context) = @_;
    $context->define_filter("md5", \&md5);
    return $class;
}

1;
```

The Printer plugin shown earlier is another good example of a plugin that could also work as a filter:

```
[% USE Printer %]
[% text | print %]
```

Modifying load to do what we intend is simple:

```
sub load {
    my ($class, $context) = @_;
    my $printer = Net::Printer->new(printer => $PRINTER,
                                    server  => $SERVER);
    my $self = bless {
        _CONTEXT => $context,
        _PRINTER => $printer,
    }, $class;

    $context->define_filter('print', sub { $self->print(@_) });

    return $self;
}
```

We need to pass a closure to define_filter because print needs access to $self (the plugin object) when it is invoked, unlike the MD5 filter, where md5 was simple enough to stand on its own.

8.3.4 Defining New Virtual Methods from Within a Plugin

Virtual methods are defined within Template::Stash, and are implemented as subroutine references attached to package-scoped hashes within the Template::Stash namespace: $Template::Stash::SCALAR_OPS for scalar vmethods, $Template::Stash::LIST_OPS for list vmethods, and $Template::Stash::HASH_OPS for hash vmethods. Creating a new vmethod is as simple as assigning a subroutine reference to the appropriate package variable.

To get a feel for creating vmethods, let's add a few. Graham Barr's *List::Util* package (shipped with Perl as of 5.8.0, available from http://search.cpan.org/dist/List-Util/ for versions before 5.8.0) provides several very useful functions that operate on arrays, such as shuffle, which will randomize an array, and max, which will return the largest numeric value in an array:

```
use Template::Stash;
use List::Util;

my $l_ops = $Template::Stash::LIST_OPS;

$l_ops->{'shuffle'} = \&List::Util::shuffle;
$l_ops->{'max'} = \&List::Util::max;
```

These new virtual methods can now be used like any other virtual methods:

```
[% list = [ 1 2 3 4 5 ];
   shuflist = list.shuffle;
%]
```

Note that because of how virtual methods are implemented, once a subroutine is installed as a vmethod, it is global, and available to all templates.

8.3.5 Writing New Plugins

To help you get a feel for the real-world issues that crop up when you build plugins, let's look closely at three sample plugins, building from a simple wrapper to one that searches Google.

8.3.5.1 A simple wrapper plugin

One of the simplest types of plugins is one that acts as a factory for another object-oriented module, such as CGI or Apache. In a case such as this, the entire plugin can be implemented by having the plugin's new( ) method defer to the modules constructor. A good example is the standard CGI plugin, the entirety of which is Example 8-6.

Example 8-6. Standard CGI plugin

```
package Template::Plugin::CGI;

use strict;
use base qw( Template::Plugin );
use Template::Plugin;
use CGI;

sub new {
    my $class   = shift;
    my $context = shift;
    CGI->new(@_);
}

1;

_ _END_ _
```

Most of the time, however, plugins require a little more work. Under mod_perl, the Apache module provides a way to directly access the current requested object and manipulate the request. An Apache plugin, to be used in a template running under mod_perl, might look like Example 8-7.

Example 8-7. Apache plugin

```
package TTBook::Plugin::Apache;

use strict;
use vars qw($VERSION);

$VERSION = 1.00;

use Apache;
use base qw(Template::Plugin);

sub new {
    return Apache->request;
}
```

In the case of the Apache class, the constructor is named request( ), which returns a reference to the current Apache request object. This plugin would be used like this:

```
[% USE r = Apache %]

<p>Query parameters are: '[% r.args %]'.</p>
<p>You are using [% r.header_in('User-Agent') %].</p>
```

Of course, most plugins are not this simple, including this one. Because this module delegates to a regular Apache instance, we can still call standard Apache methods against it, including the print method, which can have unpredictable results when invoked within a template. Because we're dealing with a plugin, and plugins are basically regular Perl modules, we can inherit from the Apache module, implement a Template Toolkit-friendly version of the print method, and return a reference to our subclass. The Apache module makes special allowances for subclasses: an object that is not an Apache instance is checked to see whether it is a hash, and whether it contains an Apache instance or subclass as a data member named _r. Using this information, we can rewrite our plugin to be a little more interesting. The rewritten plugin is shown in Example 8-8.

Example 8-8. Rewritten Apache plugin

```
package TTBook::Template::Plugin::Apache;

use Apache;
use base qw( Template::Plugin Apache );
use vars qw($VERSION);

$VERSION = 1.01;

sub new {
    my ($class, $context) = @_;
```

```
    bless {
        '_r' => Apache->request,
    } => $class;
}

sub print {
    my ($self, @data) = @_;
    my ($str, $output);

    for $str (@data) {
        if (ref $str eq 'SCALAR') {
            $output .= $$str;
        } else {
            $output .= $str;
        }
    }

    return $output;
}
```

We've added a print method that accumulates output and returns it to the context. (Apache's print method allows scalar references to be passed, for efficiency; our method defeats this efficiency at the cost of working correctly.) Now, calls to the instance's print( ) method Do the Right Thing:

```
[% r.print('foo') %]
```

The preceding code is the same as:

```
[% foo %]
```

which isn't all that useful, in and of itself, except that it prevents unforeseen errors.

Something similar has to be done with the send_http_header( ) method, but in this case, we can discard the call, assuming that something else will be sending the headers. Apache's send_http_header( ) takes an optional $content_type, which is used to set the Content-Type header (this is generally optional, as the TypeHandler usually has already set the content type). Our send_http_header( ) can call the content_type( ) method to set the content type if one is provided:

```
sub send_http_header {
    my $r = shift;

    if (my $content_type = shift) {
        $r->content_type($content_type);
    }

    return '';
}
```

send_http_header( ) explicitly returns an empty string, so we don't get any unexpected output.

We can make this plugin available to our templates using the PLUGIN configuration parameter:

```
my $t = Template->new({
    PLUGINS => {
        'apache' => 'TTBook::Template::Plugin::Apache',
    }
});
```

Example 8-9 is the complete TTBook::Template::Plugin::Apache.

Example 8-9. TTBook::Template::Plugin::Apache

```
package TTBook::Template::Plugin::Apache;

use strict;
use vars qw($VERSION);

use Apache;
use base qw(Template::Plugin Apache);

$VERSION = 1.02;

sub new {
    my ($class, $context) = @_;

    bless {
        '_r' => Apache->request,
    } => $class;
}

sub print {
    my ($self, @data) = @_;
    my ($str, $output);

    for $str (@data) {
        if (ref $str eq 'SCALAR') {
            $output .= $$str;
        } else {
            $output .= $str;
        }
    }

    return $output;
```

```
}

sub send_http_header {
    my $r = shift;

    if (my $content_type = shift) {
        $r->content_type($content_type);
    }

    return "";
}

1;
```

8.3.5.2 A more complex wrapper plugin

The next type of plugin is one that is based on an object-oriented module, but that needs configuration or runtime translation; a good example is LWP. LWP provides a web useragent in the LWP::UserAgent class, and a host of supporting modules, representing HTTP requests and responses, server messages, and even robots; using these powerful modules can be complex. We will develop a simple, easy-to-use plugin frontend for LWP::UserAgent; most of the work that we need to do will involve translating data that the Template Toolkit wraps up into hashrefs back into the hashes that the LWP::UserAgent methods are expecting:

```
package TTBook::Template::Plugin::LWP;

use strict;
use base qw(Template::Plugin);

use HTTP::Request;
use LWP::UserAgent;
use Template::Plugin;
```

We would like it to be useable in standard plugin style:

```
[% USE lwp %]
```

perhaps with some specified parameters to indicate the name of the useragent:

```
[% USE lwp(agent => 'TTBook bot/1.0') %]
```

or proxy information:

```
[% USE ua = lwp(env_proxy => 1) %]
```

or all:

```
[% USE lwp(agent     => 'TTBook bot/1.0',
           env_proxy => 1,
           timeout   => 60) %]
```

The constructor for LWP::UserAgent expects a hash of (name, value) pairs, rather than the hashref that the Template Toolkit passes to plugin constructors, which means that we will need to do a little translation. The new( ) method for our plugin, therefore, looks like this:

```
sub new {
    my ($class, $context, $plugin_params) = @_;
    my ($self, $ua, %lwp_params);

    %lwp_params = %$plugin_params;
    $ua = LWP::UserAgent->new(%lwp_params);

    return bless {
        _CONTEXT => $context,
        _UA      =>   $ua,
    } => $class;
}
```

Using the plugin should be simple, too; LWP::UserAgent supports GET, POST, and HEAD requests in the form of the get( ), post( ), and head( ) methods, so our plugin will inherit these, but they will require some parameter mapping to make their calling sequence seem more natural to plugin users. These methods take, as parameters, the request URI and then (name, value) pairs that specify headers; the special header named Content will be used to set the content of the request (for POST and PUT requests), rather than to create a header. Our plugin interface will maintain this split, but, just like the constructor, will need to map from hashref to hash.

These methods can be accessed simply as:

```
[% use.perl.org = lwp.get('http://use.perl.org/') %]
```

The URL plugin can be of great assistance here:

```
[% USE url('http://use.perl.org/journal.pl', light = 1) %]
[% use.perl.org = lwp.get(url(uid = 18)) %]
```

Our plugin doesn't have to do anything to get the benefits of this; url has been dereferenced by the Template Toolkit before, and our method is passed a string.

Our get, post, and head wrappers would look like this:

```
sub get {
    my ($self, $url, $query_params) = @_;
    my %get_params = %$query_params;
    my $ua = $self->{ _UA };

    return $ua->get($url, %get_params);
}


sub head {
    my ($self, $url, $query_params) = @_;
    my %head_params = %$query_params;
    my $ua = $self->{ _UA };
```

```
    return $ua->head($url, %head_params);
}


sub post {
    my ($self, $url, $query_params) = @_;
    my %post_params = %$query_params;
    my $ua = $self->{ _UA };


    return $ua->post($url, %post_params);
}
```

We can use these pretty simply:

```
[% lwp.post(url, 'Content' = my_text) %]
```

We have often wished that there was a general-purpose download method in the LWP::UserAgent class, so let's create one. The request method of the LWP::UserAgent class will write the requested content to a disk file when passed a string as a second argument, so we can begin there:

```
sub download {
    my ($self, $uri, $filename) = @_;
    my ($ua, $context, $request);


    $ua      = $self->{ _UA };
    $context = $self->{ _CONTEXT };
```

We can't just defer to the get method of LWP::UserAgent here; we'll need to use HTTP::Request directly:

```
$request = HTTP::Request->new(GET => $uri);
```

(We assume a GET request; implementing download for other request types is left as an exercise for the reader.)

```
$ua->request($request, $filename)
    || $context->throw('file', "Can't write $filename: $!");
```

Because this method is writing to the filesystem, there is the possibility that it can fail; this needs to be checked for success. If the write fails, we throw a file exception using $context.

Finally, we return the content of the response:

```
return $response->content;
}
```

Making our LWP plugin available to templates can be achieved by passing it as an element of the PLUGINS hash:

```
my $t = Template->new({
    PLUGINS => {
        'lwp'   => 'TTBook::Template::Plugin::LWP',
    }
});
```

Example 8-10 is the complete TTBook::Template::Plugin::LWP.

Example 8-10. TTBook::Template::Plugin::LWP

```
package TTBook::Template::Plugin::LWP;

use strict;
use vars qw($VERSION);
use base qw(Template::Plugin);

use HTTP::Request;
use LWP::UserAgent;
use Template::Plugin;

$VERSION = 1.00;

sub new {
    my ($class, $context, $plugin_params) = @_;
    my ($self, $ua, %lwp_params);

    %lwp_params = %$plugin_params;
    $ua = LWP::UserAgent->new(%lwp_params);

    return bless {
        _CONTEXT => $context,
        _UA      =>   $ua,
    } => $class;
}

sub get {
    my ($self, $url, $query_params) = @_;
    my %get_params = %$query_params;
    my $ua = $self->{ '_UA' };

    return $ua->get($url, %get_params);
}

sub head {
    my ($self, $url, $query_params) = @_;
    my %head_params = %$query_params;
    my $ua = $self->{ _UA };
```

```
    return $ua->head($url, %head_params);
}


sub post {
    my ($self, $url, $query_params) = @_;
    my %post_params = %$query_params;
    my $ua = $self->{ _UA };


    return $ua->post($url, %post_params);
}


sub download {
    my ($self, $uri, $filename) = @_;
    my ($ua, $context, $request);


    $ua      = $self->{ _UA };
    $context = $self->{ _CONTEXT };
    $request = HTTP::Request->new(GET => $uri);


    $ua->request($request, $filename)
        || $context->throw('file', "Can't write $filename: $!");


    return $response->content;
}


1;
```

8.3.5.3 A plugin that sends mail

Sending mail is such a common thing to do with the Template Toolkit, it is surprising that there is no standard plugin to handle it. Many mail-related Perl modules are on CPAN, but the simplest is Mail::Sendmail, which exports a single subroutine (sendmail) that takes a hash of arguments. We can use this as the basis for our Mail plugin.

A mail plugin would need to have methods to get and set the To, From, Cc, Bcc, Subject, and Body fields:

```
[% Mail.To('you@yourhost.com') %]
[% Mail.From('me@myhost.com') %]
[% Mail.Subject('Re: your mail') %]


[% body = BLOCK %]
Hello, friend!
[% END %]


[% Mail.Body(body) %]
```

```
[% Mail.body(body) %]
```

Additionally, it would be nice to be able to reuse the plugin instance—in a loop, for example:

```
[% addresses = [ 'one@addr.ess'
                 'two@addr.ess'
                 'three@addr.ess'
                 'four@addr.ess'
               ];

  message_content = 'The system will be down blah blah blah.';

  USE Mail from    => 'Administrator <admin@addr.ess>',
           subject => 'Scheduled downtime',
           body    => message_content;

  FOREACH address = addresses;
      Mail.send(to => address);
      Mail.reset;
  END;
%]
```

Our plugin begins fairly predictably:

```
package TTBook::Template::Plugin::Mail;

use strict;
use base qw(Template::Plugin);
use vars qw($VERSION $AUTOLOAD);

use Mail::Sendmail;
use Net::Domain qw(hostfqdn);
use Template::Exception;
use Template::Plugin;

$VERSION = 1.00;
$AUTOLOAD = undef;
```

We'll be using Template::Exception to propagate errors, so they can be caught and handled appropriately. Net::Domain gives us hostfqdn, which will help us generate a Message-ID header. We'll need $VERSION and $AUTOLOAD later, so we declare them now.

Because we want the user to be able to invoke our plugin not only as:

```
[% USE Mail %]
```

but also with default arguments:

```
[% USE Mail subject = 'Testing, testing, testing'
            from = 'admin@template-toolkit.org' %]
```

we can write new to accept parameters:

```
sub new {
    my ($class, $context, $params) = @_;
    my $self;
```

As you recall, named parameters are passed to subroutines as the last element in @_, as a reference to a hash; any parameters that the user specifies in the USE line will be there.

```
$params->{ server } = 'mailhost'
    unless defined $params->{ server };
```

Mail::Sendmail requires the name of the SMTP relay to be specified as one of its arguments, but we'll take that responsibility out of the user's hands and use a reasonable default. Savvy users can still specify a server to use, for example:

```
[% USE Mail server => 'localhost' %]
```

In order to reuse our plugin, we'll need to keep the default configuration values separate from values set later. To do this, we will use two data members for parameters:

```
$self = bless {
    _CONTEXT       => $context,
    _ORIG_PARAMS   => $params,
    _PARAMS        => { },
    _LOGMESSAGE    => '',
} => $class;
```

_ORIG_PARAMS is the configuration parameters that were specified at instance creation time and that will be used as our defaults. We finish our new( ) method with:

```
$self->reset( );
return $self;
    }
```

The reset( ) method is responsible for copying the elements of _ORIG_PARAMS into _PARAMS:

```
sub reset {
    my $self = shift;
    delete $self->{ _ORIG_PARAMS }->{ 'message-id' };
    %{ $self->{ _PARAMS } } = %{ $self->{ _ORIG_PARAMS } };
    $self->{ _LOGMESSAGE } = '';
    return $self;
}
```

reset( ) takes the precaution of deleting the Message-ID key: because this must be unique for each outgoing email, we don't take the chance that the user hasn't specified it manually. We also reset the _LOGMESSAGE string, which will contain a transcript of the conversation with the server.

The most important method, send, is very straightforward. It is used like this:

```
[% Mail.send(params) %]
```

This is our last chance to specify parameters—they will be mixed in with _PARAMS. Mail::Sendmail provides a transcript of its communications with the server in the $Mail::Sendmail::log variable; we'll store this in the _LOGMESSAGE instance variable.

```
sub send {
    my $self = shift;
    my ($params, $context) = @$self{ qw( _PARAMS _CONTEXT) };
    my $mail = ref($_[-1]) eq 'HASH' ? pop @_ : { };
```

```
    %$mail = ('X-Mailer' => join('/', ref $self, $VERSION),
              %$params,
              %$mail);

    $mail->{'message-id'} = $self->generate_mid( )
        unless defined $mail->{'message-id'};

    sendmail(%$mail)
        or $context->throw('mail', $Mail::Sendmail::error);

    $self->{ _LOGMESSAGE } = $Mail::Sendmail::log;

    return '';
}
```

Both $params and $mail are hash references, so they can be dereferenced sequentially to produce one hash. Because $mail is dereferenced after $params, any keys defined in $mail supercede those in $params—which is to say that parameters specified in send override those set earlier. Finally, we add a vanity header (X-Mailer), which also can be overridden by either $params or $mail:

```
[% Mail.send('X-Mailer' => 'Micros~1 Outlook 6.6.6') %]
```

The send method returns an empty string so that there is no unintentional output when it is invoked.

We need to explicitly create a Message-ID header if one hasn't been provided by the user. Most MTAs will add a Message-ID header if it isn't present, but many will not, so we cannot rely on it. The Message-ID header will be used to uniquely identify a message in space and time; ideally, it should consist of enough information to identify the message without giving away too much information about the user. The generate_mid method creates a Message-ID based on the time, domain name, and eight characters of randomness ($junk):

```
sub generate_mid {
    my $self = shift;
    my @time = localtime;
    my $junk = join '', map { ('a'..'z', 'A'..'Z')[rand 52] } (0..8);

    my $mid = sprintf '<%d%02d%02d.%s@%s>',
        $time[5] + 1900, $time[4], $time[3], $junk, hostfqdn( );

    return $mid;
}
```

We can access the transcript using the logmessage( ) method:

```
sub logmessage {
    my $self = shift;
    return $self->{ _LOGMESSAGE };
}
```

Finally, the other methods can be handled by an AUTOLOAD method:

```
my %multi = map { $_ => 1 } qw(to cc bcc);
```

```
sub AUTOLOAD {
    my $self = shift;
    my ($method, $item);

    $method = $AUTOLOAD;
    $method =~ s/.*:://;
    $method = ucfirst lc $AUTOLOAD;
    $method =~ s/_(\w)/-\u$1/g;

    # Make an alias
    $item = \$self->{ _PARAMS }->{ $method };

    if (@_) {
        if (defined $multi{ $method }) {
            my @addrs;
            if (ref $_[0] eq 'ARRAY') {
                @addrs = @{$_[0]};
            } else {
                @addrs = @_;
            }
            $$item = join ', ', @addrs;
        } else {
            $$item = shift @_;
        }
        return '';
    }

    return $$item;
}
```

Perl's AUTOLOAD facility catches calls for methods that do not exist (which makes it perfect as a catchall method for this plugin). Mail::Sendmail will pass on any parameters passed to the sendmail( ) function as headers; we can combine these two facts to let Perl write the rest of our methods for us. When AUTOLOAD is invoked, the name of the invoked method is in the variable $AUTOLOAD, with the fully qualified package name. Mail::Sendmail takes header names in any case, but we normalize it (by lowercasing) to keep from storing duplicates in _PARAMS. Using this AUTOLOAD, we can set any arbitrary header, not just the ones mentioned earlier:

```
[% Mail.message_id('20030811-093159@localhost') %]

[% Mail.x_pgp_fingerprint(pgp_f) %]
```

To, Cc, and Bcc can be multivalued elements (as defined in %multi), so we accept a list of elements. This allows us to do this:

```
[% Mail.To(address1, address2, address3) %]
```

We also explicitly check to see whether $_[0] is an array reference, and dereference it if it is. This is because if we pass a list created in our template, it will be an array reference:

```
[% addresses = [ 'one@addr.ess',
                 'two@addr.ess',
                 'three@addr.ess'
               ];
 Mail.To(addresses) %]
```

If we are setting a value, we explicitly return the empty string, so there are no side effects.

Because send throws an exception if it cannot contact the mail server, or if something else goes wrong, we need to wrap calls to send in a TRY...CATCH block:

```
[% TRY %]
   [% Mail.send %]
[% CATCH mail %]
   Error: [% error %]
[% END %]
```

The last thing to do is to make the plugin available to our templates:

```
my $t = Template->new({
    PLUGINS => {
        'mail' => 'TTBook::Template::Plugin::Mail',
    }
});
```

Example 8-11 is the complete TTBook::Template::Plugin::Mail.

Example 8-11. TTBook::Template::Plugin::Mail

```
package TTBook::Template::Plugin::Mail;

use strict;
use base qw(Template::Plugin);
use vars qw($VERSION $AUTOLOAD);

use Mail::Sendmail;
use Net::Domain qw(hostfqdn);
use Template::Exception;
use Template::Plugin;

$VERSION = 1.00;
$AUTOLOAD = undef;

sub new {
    my ($class, $context, $params) = @_;
    my $self;
```

```
    $params->{ server } = 'mailhost'
        unless defined $params->{ server };

    $self = bless {
        _CONTEXT        => $context,
        _ORIG_PARAMS    => $params,
        _PARAMS         => { },
        _LOGMESSAGE     => '',
    } => $class;

    $self->reset( );
    return $self;
}

sub reset {
    my $self = shift;
    delete $self->{ _ORIG_PARAMS }->{ 'message-id' };
    %{ $self->{ _PARAMS } } = %{ $self->{ _ORIG_PARAMS } };
    $self->{ _LOGMESSAGE } = '';
    return $self;
}

sub send {
    my $self = shift;
    my ($params, $context) = @$self{ qw( _PARAMS _CONTEXT) };
    my $mail = ref($_[-1]) eq 'HASH' ? pop @_ : { };

    %$mail = ('X-Mailer' => join('/', ref $self, $VERSION),
              %$params,
              %$mail);

    $mail->{'message-id'} = $self->generate_mid( )
        unless defined $mail->{'message-id'};

    sendmail(%$mail)
        or $context->throw('mail', $Mail::Sendmail::error);

    $self->{ _LOGMESSAGE } = $Mail::Sendmail::log;
```

```
    return '';
}

sub generate_mid {
    my $self = shift;
    my @time = localtime;
    my $junk = join '', map { ('a'..'z', 'A'..'Z')[rand 52] } (0..8);

    my $mid = sprintf '<%d%02d%02d.%s@%s>',
        $time[5] + 1900, $time[4], $time[3], $junk, hostfqdn( );

    return $mid;
}

sub logmessage {
    my $self = shift;
    return $self->{ _LOGMESSAGE };
}

my %multi = map { $_ => 1 } qw(to cc bcc);
sub AUTOLOAD {
    my $self = shift;
    my ($method, $item);

    $method = $AUTOLOAD;
    $method =~ s/.*:://;
    $method = ucfirst lc $method;
    $method =~ s/_(\w)/-\u$1/g;

    # Make an alias
    $item = \$self->{ _PARAMS }->{ $method };

    if (@_) {
        if (defined $multi{ $method }) {
            my @addrs;
            if (ref $_[0] eq 'ARRAY') {
                @addrs = @{$_[0]};
            } else {
```

```
            @addrs = @_;
        }
        $$item = join ', ', @addrs;
    } else {
        $$item = shift @_;
    }
    return '';
  }

  return $$item;
}

1;
```

8.3.5.4 GoogleSearch

Everybody loves Google, right? Since the advent of the Google API, everybody can write their own custom search interface. Aaron Straup Cope's Net::Google provides a nice, simple Perl interface to the Google SOAP API.

In order to use this plugin, you'll need to register with Google; you can do so at http://api.google.com/.

Using the GoogleSearch plugin should be straightforward:

```
[% USE g = GoogleSearch('Template Toolkit') %]
[% num = g.num_results %]

[% FOREACH result = g.results %]
  [% result.title %]
  [% result.URL %]
[% END %]
```

The plugin starts with the usual prologue:

```
package TTBook::Template::Plugin::GoogleSearch;

use strict;
use vars qw($VERSION $KEY);
use base qw(Template::Plugin);

use Net::Google;
use Template::Exception;
use Template::Iterator;
use Template::Plugin;

$VERSION = 1.00;
$KEY = 'cc42973b5c5f292a7be146e1b444379e';
```

$KEY is your Google key. Don't use the one in the preceding code because it isn't real (it's the MD5 hash of the string Template Toolkit).

Net::Google works by creating and reusing a Net::Google instance, which acts as a factory for Net::Google::Search instances. The best way to represent this is by using the singleton plugin pattern described earlier:

```
sub load {
    my ($class, $context) = @_;
    my $google = Net::Google->new(key => $KEY);

    bless {
        _CONTEXT => $context,
        _GOOGLE => $google,
    } => $class;
}
```

We will need $context for throwing exceptions.

new( ) is where we create the Net::Google::Search instance:

```
sub new {
    my ($self, $context, @args) = @_;
    my ($params, $google, $search, $p);

    $params = ref $args[-1] eq 'HASH' ? pop @args : { };

    $google = $self->{ _GOOGLE };
    $search = $self->{ _SEARCH } = $google->search( );

    for $p (qw/ lr ie oe starts_at
               max_results safe filter /) {
        $search->$p($params->{$p})
            if defined $params->{$p};
    }

    $search->query(join ' ', @args);

    return $self;
}
```

Search terms are provided as positional arguments, while other elements of the search are provided as named arguments:

```
[% USE g = GoogleSearch max_results = 50
                        lr = [ 'de' 'es' ]
                        'perl'
                        '"templating languages"' %]
```

This search, for *perl* and *templating languages*, will return up to 50 results (instead of the default 10) and will search

German and Spanish pages only. (See the Net::Google::Search manpage for what the available parameters actually are.)

Our result set will be wrapped in a Template::Iterator instance:

```
sub results {
    my $self = shift;
    my ($search, @results, $iter);

    $search = $self->{ _SEARCH } || return Template::Iterator->new([  ]);
    @results = @{$search->results( )};
    $iter = Template::Iterator->new(\@results);

    return $iter;
}
```

Each element in the iterator is a Result object (created by the Net::Google::Response object), and has methods useable to access the elements of the result:

```
[% FOREACH result = g.results %]
    blah blah blah
```

Example 8-12 is the complete TTBook::Template::Plugin::GoogleSearch.

Example 8-12. TTBook::Template::Plugin::GoogleSearch

```
package TTBook::Template::Plugin::GoogleSearch;

use strict;
use vars qw($VERSION $KEY);
use base qw(Template::Plugin);

use Net::Google;
use Template::Exception;
use Template::Iterator;
use Template::Plugin;

$VERSION = 1.00;
$KEY = "cc42973b5c5f292a7be146e1b444379e";

sub load {
    my ($class, $context) = @_;
    my $google = Net::Google->new(key => $KEY);

    bless {
        _CONTEXT => $context,
        _GOOGLE => $google,
```

```
    } => $class;
}

sub new {
    my ($self, $context, @args) = @_;
    my ($params, $google, $search, $p);

    $params = ref $args[-1] eq 'HASH' ? pop @args : { };

    $google = $self->{ _GOOGLE };
    $search = $self->{ _SEARCH } = $google->search( );

    for $p (qw/ lr ie oe starts_at
               max_results safe filter /) {
        $search->$p($params->{$p})
            if defined $params->{$p};
    }

    $search->query(join " ", @args);

    return $self;
}

sub results {
    my $self = shift;
    my ($search, @results, $iter);

    $search = $self->{ _SEARCH } ||
        return Template::Iterator->new([  ]);
    @results = @{$search->results( )};
    $iter = Template::Iterator->new(\@results);

    return $iter;
}

1;
```

8.3.5.5 Normalizing URLs

For some reason, many organizations find it difficult to keep their URLs consistent. This plugin might be helpful: given a relative URL, it will return the canonical version of it, relative to either the main host, or to the graphics host if the link looks like it might be an image. For example:

```
[% USE Link www_host = 'www.example.com' %]

<a href="[% link('/foo/bar.html') %]">...</a>
```

will produce:

```
<a href="http://www.example.com/foo/bar.html">...</a>
```

This Link plugin accepts a few arguments: www_host, graphics_host, and opaque. graphics_host will be used for things that appear to be images, and www_host will be used for everything else. If opaque is specified, the resulting URL will not have a scheme; this is most useful for templates that might be served under multiple protocols—for example, *http* and *https*. The client will assume the current scheme if one is not provided, so the server does not have to check whether the current page is secure.

```
[% USE Link www_host      = 'www.tt2.org',

            graphics_host = 'graphics.tt2.org',

            opaque        = 1

%]

<a href="[% link('/index.html') %]">

   <img src="[% link('ttpower.gif') %]">

</a>
```

Calls to link( ) would expand to full URIs:

```
<a href="//www.tt2.org/index.html">

   <img src="//graphics.tt2.org/ttpower.gif">

</a>
```

The URI referring to an image was detected, and the host was set to the graphics server.

It would be straightforward to modify this plugin to treat arguments to link as keywords rather than filenames.

Example 8-13 is the complete TTBook::Template::Plugin::Link.

Example 8-13. TTBook::Template::Plugin::Link

```
package TTBook::Template::Plugin::Link;



use strict;

use vars qw($VERSION $DEFAULT_WWW_HOST $DEFAULT_GRAPHICS_HOST $DEFAULT_OPAQUE);

use base qw(Template::Plugin);



use LWP::MediaTypes qw(guess_media_type);

use URI;



$VERSION      = 1.00;

$DEFAULT_WWW_HOST      = "www.example.com";

$DEFAULT_GRAPHICS_HOST = "graphics.example.com";

$DEFAULT_OPAQUE        = 0;



sub load {
```

```
    my ($class, $context, @args) = @_;
    my $params = ref $args[-1] eq 'HASH' ? pop @args : { };

    $context->stash->set("link", link_factory($params));

    bless { } => $class;
}

# Nominal new; can't inherit from Template::Plugin
sub new { return shift }

sub link_factory {
    my $params        = shift;
    my $www_host      = sprintf "http://%s/", $params->{ www_host }
                                     || $DEFAULT_WWW_HOST;
    my $graphics_host = sprintf "http://%s/", $params->{ graphics_host }
                                     || $DEFAULT_GRAPHICS_HOST;
    my $opaque        = $params->{'opaque'} || $DEFAULT_OPAQUE;

    return sub {
        my $url = shift || return;

        my $link = URI->new($url);

        # This will be the case for URIs such as "/foo", which
        # URI will decide are of type "URI::_generic"
        $link = URI->new($link, "http")->abs($www_host)
            unless ($link->can("host"));

        $link->host($graphics_host)
            if (guess_media_type($url) =~ /^image/);

        return $opaque ? $link->opaque( ) : $link->canonical( );
    };
}

1;
```

8.4 Building a New Frontend

The Template module is the default frontend to the Template Toolkit, but there are others. The Apache::Template module, available from CPAN, is one, as are the familiar *tpage* and *ttree*. Here is a description of these default frontends:

Template

The Template module is the frontend that most users are familiar with. Template provides the familiar process method:

```
$tt->process($input, $vars, $output)
    || die $tt->error( );
```

Template uses the underlying Template::Service instance internally to process $input, and then redirect that output appropriately, based on the third argument to process( ) (see Chapter 7 for details).

Apache::Template

The Apache::Template module provides a simple interface to the Template Toolkit from Apache/mod_perl. Apache::Template allows configuration to be handled in an Apache-specific manner, using directives in Apache's *httpd.conf* configuration file.

Apache::Template is covered in Chapter 12. The Appendix lists valid Apache::Template-related *httpd.conf* configuration directives.

tpage and ttree

We've already met *tpage* and *ttree* in Chapter 1 and Chapter 2; these two scripts are also Template Toolkit frontends.

A Template Toolkit frontend manages the Template::Service instance, and, generally, manages input and output. In this section, we look at these standard frontends and how to build a custom frontend for email.

8.4.1 Mail::Template

Because email is basically text, and generating text is so simple using the Template Toolkit, why isn't there a dedicated mail frontend? Well, there could be; let's develop one.

Our Template Toolkit frontend module needs two user-facing methods, new and process. The Template::Base module implements most of the common functionality of the modules that ship with the Template Toolkit, so we can start there:

```
package Mail::Template;


use strict;
use vars qw($VERSION $MAILHOST $MAILPORT);
use base qw(Template::Base);


use Mail::Sendmail qw(sendmail);
use Template::Base;


$VERSION = 1.00;
$MAILHOST = "mailhost" unless defined $MAILHOST;
$MAILPORT = 25 unless defined $MAILPORT;
```

The Mail::Sendmail module provides the sendmail function, which, well, sends mail. $MAILHOST and $MAILPORT are defined as package variables so that the defaults can be overridden in client code:

```
use Mail::Template;

$Mail::Template::MAILHOST = "smtp.example.com";
```

The new method inherited from Template::Base calls the _init method, which Mail::Template can use to handle specific constructor details. _init is called with a reference to a hash containing the parameters passed to new.

```
sub _init {
    my ($self, $config) = @_;

    $self->{ _MAILHOST } = $config->{ MAILHOST } || $MAILHOST;

    if (not defined $config->{ MAILPORT }) {
        if ($self->{ _MAILHOST } =~ s/:(\d+)$//) {
            $self->{ _MAILPORT } = $1;
        }
        else {
            $self->{ _MAILPORT } = $MAILPORT;
        }
    }

    # Setup a Template::Service instance
    $self->{ SERVICE } = $config->{ SERVICE }
        || Template::Config->service($config)
        || return $self->error(Template::Config->error);

    return $self;
}
```

Mail::Template looks for two unique parameters: MAILHOST and MAILPORT, both of which are assigned reasonable defaults (mailhost and 25, respectively). We can use an alternate port or host by passing them specifically, or the two can be joined with a colon as MAILHOST:

```
my $config = { MAILHOST => "smtp-server:2525" };

my $mt = Mail::Template->new($config);
```

The Template::Service instance is created as an idiom that occurs in many places throughout the Template Toolkit. The error method, which is inherited from Template::Base, does double-duty: if called without an argument, it returns the most recent error message, but if called with an argument, it sets the error data field and returns undef. The Template::Config class defines methods for instantiating all of the major components of the Template Toolkit in one easy-to-use, easy-to-override place. Any other parameters specified to the Mail::Template constructor will be passed on to the objects that the Template::Service instance creates.

The format of the process method is modeled after Template::process:

```
sub process {
    my ($self, $input, $vars, $addrs, @opts) = @_;
    my ($output, $error);
    my $service = $self->{ SERVICE };
    my $options = (@opts =  = 1) && ref($opts[0]) eq 'HASH'
```

```
        ? shift(@opts) : { @opts };
    $addrs = ref($addrs) eq 'ARRAY' ? $addrs : [ $addrs ];

    return $self->error("No recipients specified")
        unless @$addrs;

    $output = $service->process($input, $vars);

    if (defined $output) {
        $options->{ To      }   = $addrs;
        $options->{ Message }   = $output;
        $options->{ Server  } ||= $self->{ MAILHOST };
        $options->{ Port    } ||= $self->{ MAILPORT };

        if (sendmail(%$options)) {
            return 1;
        }
        else {
            return $self->error($Mail::Sendmail::error);
        }
    }
    else {
        return $self->error($service->error);
    }
}
```

Just like Template::process, Mail::Template::process can take up to four arguments: the template to be processed; a reference to a hash of parameters; a reference to a list of addresses; and a reference to a hash of mail options, which will be used to set mail-specific headers, such as Subject and From:

```
my $friends = [ qw(abw@cpan.org dave@dave.org.uk) ];
my $options = {
    Subject => "Testing Mail::Template",
    From => "Darren Chamberlain <darren@cpan.org>",
};

$mt->process($input, $vars, $friends, $options)
    || die $tt->error;
```

The processing of the template is handled by the Template::Service instance, which was created in _init. This leaves only the sending of the mail for process to handle (we farm that out to Mail::Sendmail).

Example 8-14 is the complete Mail::Template.

Example 8-14. Mail::Template

```
package Mail::Template;

use strict;
use vars qw($VERSION $MAILHOST $MAILPORT);
use base qw(Template::Base);

use Mail::Sendmail qw(sendmail);
use Template::Base;

$VERSION = 1.00;
$MAILHOST = "mailhost" unless defined $MAILHOST;
$MAILPORT = 25 unless defined $MAILPORT;

sub _init {
    my ($self, $config) = @_;

    $self->{ _MAILHOST } = $config->{ MAILHOST } || $MAILHOST;

    if (not defined $config->{ MAILPORT }) {
        if ($self->{ _MAILHOST } =~ s/:(\d+)$//) {
            $self->{ _MAILPORT } = $1;
        }
        else {
            $self->{ _MAILPORT } = $MAILPORT;
        }
    }

    # Set up a Template::Service instance
    $self->{ SERVICE } = $config->{ SERVICE }
        || Template::Config->service($config)
        || return $self->error(Template::Config->error);

    return $self;
}

sub process {
    my ($self, $input, $vars, $addrs, @opts) = @_;
    my ($output, $error);
```

```
    my $service = $self->{ SERVICE };
    my $options = (@opts =  = 1) && ref($opts[0]) eq 'HASH'
        ? shift(@opts) : { @opts };
    $addrs = ref($addrs) eq 'ARRAY' ? $addrs : [ $addrs ];

    return $self->error("No recipients specified")
        unless @$addrs;

    $output = $service->process($input, $vars);

    if (defined $output) {
        $options->{ To      }   = $addrs;
        $options->{ Message }   = $output;
        $options->{ Server  } ||= $self->{ MAILHOST };
        $options->{ Port    } ||= $self->{ MAILPORT };

        if (sendmail(%$options)) {
            return 1;
        }
        else {
            return $self->error($Mail::Sendmail::error);
        }
    }
    else {
        return $self->error($service->error);
    }
}

1;
```

8.4.2 Custom Apache Handlers

In many ways, writing a mod_perl-based frontend is easier than writing other types of frontends because it doesn't need to be as flexible. There is only one way that your handler will be called, and you know exactly what arguments will be provided. There are a few things to keep in mind when writing this frontend, though; a primary goal should be to avoid recreating Template Toolkit components whenever possible, especially expensive objects such as the parser. Providing full access to the request object and the metadata associated with it, such as cookies and form parameters, is also very important.

The differences between Apache 1.3 and Apache 2.0 make themselves known only in the machinery needed to make the handler work; the Template Toolkit aspects are identical. Let's take a look at a simple Apache 1.3/mod_perl 1.x handler:

```
package TTBook::ApacheHandler;


use strict;

use vars qw($VERSION);


$VERSION = 1.00;    # Apache 1.3.x handler


use Apache;

use Apache::Constants qw(OK SERVER_ERROR);

use Template::Config;

use URI::Escape qw(uri_unescape);


# Preload all Template Toolkit modules

Template::Config->preload( );


my $tt;
```

We'll need the OK, DECLINED, and SERVER_ERROR constants—OK for when there are no problems, SERVER_ERROR for when there are, and DECLINED so that we can specifically decline to handle requests for files that don't exist (or requests for things that aren't files, such as directories). Using DECLINED like this means that Apache's normal error handlers can be used for 404's and the like.

Using Template::Config and getting a service instance through Template::Config->service means that we can use a custom subclass without having to change our handler code. The Template Toolkit will defer loading modules until they are needed, but calling Template::Config->preload will force all of them to be loaded immediately. Under mod_perl, this is important because modules compiled in the parent process will reside in the segment of memory shared among all the child processes, which can result in memory savings.

We use a package-scoped lexical variable, $tt, to store our service instance so that it can be shared between multiple requests by the same child:

```
sub handler {

    my $r = shift;

    my ($filename, $docroot, %vars, $template, $content);


    $filename = $r->filename;

    $docroot = $r->docroot;


    return DECLINED unless -f $filename;
```

If this is the first time the current child process has been called up to handle a template, $tt will not be defined. We define it here, and check for errors:

```
$tt ||= do {

    Template::Config->service({

        INCLUDE_PATH => [ $docroot ],

    });
```

```
};

unless (defined $tt) {
    # Catch errors here, and return SERVER_ERROR
    my $mod = $Template::Config::SERVICE;
    $r->log_error("Can't create $mod instance: ",
                  Template::Config->error);
    return SERVER_ERROR;
}
```

If creating a Template::Service instance fails, we need to report it. A well-behaved mod_perl script will write to Apache's error_log and the best way to do that is to use the Apache object's log_error method. We feed it the error according to Template::Config.

We can make query parameters available as top-level variables so that a request for /news/2003/08/11?article=34293 makes a variable called article available within the templates:

```
[% article %]
```

In list context, both $r->args and $r->content return a hash of variables, which is, conveniently enough, what we will need to pass to process:

```
%vars = $r->method eq 'POST' ? $r->content : $r->args;
```

Apache doesn't make the parsed cookies available, but they can be pulled out pretty easily:

```
my @cookies = split /;\s*/, $r->header_in('cookie');
for my $cookie (@cookies) {
    my ($name, $value) = map { uri_unescape($_) } split /=/, $cookie;
    $vars{$name} = $value;
}
```

This makes cookies available as top-level variables, just like query parameters.

The service instance uses the DocumentRoot for its INCLUDE_PATH, so we need to strip it from the filename. A request for something like /news/2003/08/11 will be resolved to a filename such as */var/www/news/2003/08/11*, which we then turn into *news/2003/08/11*:

```
($template = $filename) =~ s,^\Q$docroot\E/?,,;
```

We pass $template to the service instance to process and check for errors. Again, we return SERVER_ERROR if something goes wrong. A more robust implementation might check whether TOLERANT was set, and return DECLINED so that the next content handler in line gets a shot (which might be Apache's default-handler):

```
$content = $tt->process($template, \%vars) || do {
    $r->log_error("$template returned no content: ",
                  $tt->error);
    return SERVER_ERROR;
};
```

At this point, $content contains the results of processing our template, and control is returned to our handler. We can add some extra header fields to the response (such as Content-Length) call $r->print($content) to tell Apache to send the data to the client, and return OK to tell Apache that we handled the request successfully:

```
$r->content_type('text/html');
$r->headers_out->add('Content-Length', length($content));
$r->send_http_header;

$r->print($content);

    return OK;
}

1;
```

You might have noticed that this handler makes no attempt to account for virtual hosts. A reasonable way to use this module—or one like it—with virtual hosts is to store the service instances in a hash keyed by $r->server_name; then each virtual host will have its own set of template objects.

Setting up TTBook::ApacheHandler within *httpd.conf* is very similar to setting up Apache::Template:

```
<Files *.html>
    SetHandler perl-script
    PerlHandler TTBook::ApacheHandler
</Files>
```

Example 8-15 is the complete TTBook::ApacheHandler.

Example 8-15. TTBook::ApacheHandler

```
package TTBook::ApacheHandler;

use strict;
use vars qw($VERSION);

$VERSION = 1.00;    # Apache 1.3.x handler

use Apache;
use Apache::Constants qw(OK SERVER_ERROR);
use Template::Config;
use URI::Escape qw(uri_unescape);

# Preload all Template Toolkit modules
Template::Config->preload( );

my $tt;

sub handler {
    my $r = shift;
```

```
my ($filename, $docroot, %vars, $template, $content);

$filename = $r->filename;
$docroot = $r->docroot;

return DECLINED unless -f $filename;

$tt ||= do {
    Template::Config->service({
        INCLUDE_PATH => [ $docroot ],
    });
};

unless (defined $tt) {
    # Catch errors here, and return SERVER_ERROR
    my $mod = $Template::Config::SERVICE;
    $r->log_error("Can't create $mod instance: ",
                  Template::Config->error);
    return SERVER_ERROR;
}

%vars = $r->method eq 'POST' ? $r->content : $r->args;

my @cookies = split /;\s*/, $r->header_in('cookie');
for my $cookie (@cookies) {
    my ($name, $value) = map { uri_unescape($_) } split /=/, $cookie;
    $vars{$name} = $value;
}

($template = $filename) =~ s,^\Q$docroot\E/?,,;

$content = $tt->process($template, \%vars) || do {
    $r->log_error("$template returned no content: ",
                  $tt->error);
    return SERVER_ERROR;
};

$r->content_type('text/html');
$r->headers_out->add('Content-Length', length($content));
```

```
    $r->send_http_header;

    $r->print($content);

    return OK;
}

1;
```

8.5 Changing the Language

The grammar for the Template Toolkit language is generated using a YACC-like parser generator written in Perl called Parse::Yapp (http://search.cpan.org/dist/Parse-Yapp/). Parse::Yapp is not distributed with or required by the Template Toolkit, but you will need it if you want to regenerate the grammar. Yapp is identical to YACC in all the important ways; for a good general introduction to YACC, see *lex & yacc*, Second Edition, by John R. Levine, Tony Mason, and Doug Brown (O'Reilly), which gives a good introduction to the principles of an LALR parser and how to define grammars in YACC. See also the Parse::Yapp documentation and the comments in Template::Parser for more information. For an in-depth study of parser and compiler theory, consult *Compilers: Principles, Techniques and Tools* (a.k.a., the "Dragon Book") by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman (Addison Wesley).

The Template Toolkit source distribution includes the subdirectory *parser*, which contains a few files, most notably one called *Parser.yp*. This is the one you will be modifying to extend the language.[3] The parser grammar is compiled by *yapp*, the frontend script to Parse::Yapp, based on the grammar skeleton *Grammar.pm.skel*, which is also in the *parser* directory.

[3] Be sure to have a backup of the file handy while you are modifying the grammar!

Changing the grammar is a simple process, in theory at least, if you're familiar with Yapp/YACC. In practice, it also requires some insight into the inner workings of the Template Toolkit.

8.5.1 Building the Grammar

The Template Toolkit distribution includes a helper script called *yc*, which builds the grammar. It is a thin wrapper around yapp that sets the appropriate options to compile, emit, and save the Perl code for the grammar. Here it is in its entirety:

```
#!/bin/sh



: ${GRAMMAR:="Parser.yp"}

: ${OUTPUT:="../lib/Template/Grammar.pm"}

: ${TEMPLATE:="Grammar.pm.skel"}



echo "Compiling parser grammar (${GRAMMAR} -> ${OUTPUT})"



yapp -v -s -o ${OUTPUT} -t ${TEMPLATE} ${GRAMMAR}
```

yc takes the grammar defined in *Parser.yp* and plugs it into the skeleton module file, *Grammar.pm.skel*. The output is written to *lib/Template/Grammar.pm*, clobbering anything that was there before. A report detailing the status of the compilation process is written to *Parser.output*:

```
$ ./yc

Compiling parser grammar (Parser.yp -> ../lib/Template/Grammar.pm)
```

yc writes the output to the *../lib/Template/Grammar.pm* file by default, so you'll need to modify the script accordingly (or set the OUTPUT environment variable) if you want to compile your own grammar module with it.

Be prepared to become intimately familiar with the (rather verbose) output in the *Parser.output* file if you're planning on writing your own grammar or making major changes to the existing grammar. Often *yapp* will refuse to compile grammar, or raise warnings about *conflicts* that indicate ambiguities in the grammar that it can't automatically resolve. In these cases, you'll need to carefully inspect the error report in *Parser.output* and trace through the rules and states listed to try and figure out where you went wrong. A good compiler reference book will be invaluable at this stage.

8.5.2 Extending the Existing Grammar

In most cases, you will be modifying the grammar because you have a specific feature or syntax element in mind that you want to be part of the core language, or your version of it. Many things can be done with plugins or filters, but you are still bound by the syntax of the language.

The Template Toolkit display language is very rich, and lacks very few control structures or directives. But occasionally, something will stand out as particularly expressive or helpful. With that in mind, let's add a feature to the language: UNTIL. UNTIL is logically equivalent to WHILE NOT, but can make for cleaner templates:

```
[% UNTIL count =  = 100 %]

   [% do.something.to(count) %]

[% END %]
```

Because UNTIL is a variation of WHILE, we can probably get away with mimicking the WHILE implementation, and simply negating the condition test. This simple implementation will give us a chance to poke around the grammar a bit.

We'll start in *parser/Parser.yp*. Download a fresh tarball (or get a new CVS checkout) of the Template Toolkit sources, and let's begin.

Parse::Yapp

As mentioned earlier, Parse::Yapp is very similar to *yacc*, and the format of the grammar file is also very similar. It consists of three main sections, divided by %%; the first section is the *preamble*, the last section is the *postamble*, and the middle section consists of sets of *rules* that define the structure of the language being represented. These rules are in the form:

```
rule: production1 | production2 | production3 ;
```

A *production* consists of two parts: a series of tokens that defines what the production looks like, and an optional action, enclosed in { and }. Productions are defined in terms of other rules and *terminals*. A terminal is a token that cannot be reduced any further— i.e., one that doesn't match any other rules.

For example, the grammar for Template::Simple defines this simple rule, chunk:

```
chunk:     TEXT           { $factory->textblock($_[1]) }

       |   statement ';'
```

The rule is chunk, and there are two productions: TEXT { ... } and statement ';' (the | indicates alternates). This means that the chunk rule is defined as either TEXT or whatever statement expands to (followed by a literal ;). The { ... } block attached to the TEXT subrule will be emitted literally into the grammar, and is assumed to be syntactically correct Perl code (it will become part of live code when the resulting grammar is actually used). The statement rule is assumed to have its own code block. The parser will pass the matching tokens to the statement as @_, with the parser as $_[0].

The parser will continue to reduce parsing until there are no expandable rules left in the input stream. At this point, the data is in its final parsed form.

The first thing to do is to modify the grammar, which means editing *parser/Parser.yp*. Because UNTIL will be based on WHILE, we can duplicate the WHILE implementation. The grammar defines WHILE as a type of loop; the definition for loop looks like this:

```
loop:      FOR loopvar ';'       { $_[0]->{ INFOR }++                }

               block END         { $_[0]->{ INFOR }--;

                                   $factory->foreach(@{$_[2]}, $_[5])  }

       |   atomexpr FOR loopvar    { $factory->foreach(@{$_[3]}, $_[1])  }

       |   WHILE expr ';'        { $_[0]->{ INWHILE }++              }

               block END         { $_[0]->{ INWHILE }--;

                                   $factory->while(@_[2, 5])          }

       |   atomexpr WHILE expr     { $factory->while(@_[3, 1])          }

;
```

We see that two types of loops are defined in the language—FOR and WHILE—and that each has a side-effect variant (e.g., atomexpr FOR loopvar).

The WHILE actions increment and decrement the INWHILE member of $_[0] (we'll see $_[0] in a moment); a quick search through the file reveals that INWHILE is used to implement the LAST and NEXT directives (these are *atomic directives*, which the grammar calls atomdir). If we are in a WHILE or FOR loop, these directives jump to the next or last occurrence of the LOOP label. Otherwise, they simply jump to the end of the current block:

```
atomdir:    GET expr      { $factory->get($_[2])              }
        ...
        |   LAST         { $_[0]->{ INFOR } || $_[0]->{ INWHILE }
                           ? 'last LOOP;'
                           : 'last;'                     }
        |   NEXT         { $_[0]->{ INFOR }
                           ? $factory->next( )
                           : ($_[0]->{ INWHILE }
                             ? 'next LOOP;'
                             : 'next;')                  }
    ...
    ;
```

So we'll need to keep INWHILE for UNTIL.

The action for WHILE calls $factory->while(@_[2, 5]). We know that $factory is a Template::Directive instance—this is what its while method looks like:

```
sub while {
    my ($class, $expr, $block) = @_;
    $block = pad($block, 2) if $PRETTY;

    return <<EOF;

# WHILE
do {
    my \$failsafe = $WHILE_MAX;
LOOP:
    while (--\$failsafe && ($expr)) {
$block
    }
    die "WHILE loop terminated (> $WHILE_MAX iterations)\\n"
        unless \$failsafe;
};
EOF
}
```

This production produces a series of five tokens: WHILE, the expansion of expr, ;, the expansion of the block, and END. These five elements, along with the parser object itself, are passed to the code block as @_. The factory's while is only interested in expr and block (which is reasonable because the other tokens are static strings):

```
|   WHILE expr ';'         { $_[0]->{ INWHILE }++              }
      block END            { $_[0]->{ INWHILE }--;
                             $factory->while(@_[2, 5])         }
```

$_[0] is the parser itself, and each token in the subrule becomes another element in the @_ array passed to the action subroutine. The parser invokes actions for subrules recursively, so $_[2], which is expr, has already been passed through the expr rule:

```
expr:   expr BINOP expr        { "$_[1] $_[2] $_[3]"              }
      | expr '/' expr          { "$_[1] $_[2] $_[3]"              }
      | expr '+' expr          { "$_[1] $_[2] $_[3]"              }
      | expr DIV expr          { "int($_[1] / $_[3])"             }
      | expr MOD expr          { "$_[1] % $_[3]"                  }
      | expr CMPOP expr        { "$_[1] $CMPOP{ $_[2] } $_[3]"    }
      | expr CAT expr          { "$_[1]  . $_[3]"                 }
      | expr AND expr          { "$_[1] && $_[3]"                 }
      | expr OR expr           { "$_[1] || $_[3]"                 }
      | NOT expr               { "! $_[2]"                        }
      | expr '?' expr ':' expr { "$_[1] ? $_[3] : $_[5]"          }
      | '(' assign ')'         { $factory->assign(@{$_[2]})       }
      | '(' expr ')'           { "($_[2])"                        }
      | term
;
```

So $_[2] contains a string of Perl code as generated by the expr rule when the action for WHILE gets to it. Most of these rules are defined in terms of themselves, except for term:

```
term:   lterm
      | sterm
;


lterm:  '[' list  ']'          { "[ $_[2] ]"                      }
      | '[' range ']'          { "[ $_[2] ]"                      }
      | '['       ']'          { "[ ]"                            }
      | '{' hash  '}'          { "{ $_[2]  }"                     }
;


sterm:  ident                  { $factory->ident($_[1])           }
      | REF ident              { $factory->identref($_[2])        }
      | '"' quoted '"'         { $factory->quoted($_[2])          }
      | LITERAL
      | NUMBER
;
```

term eventually settles itself down to be a dotted identified (ident), a quoted string (quoted), a literal (LITERAL), or a number (NUMBER), or a list, hash, or range of those things.

Similarly, $_[5] contains a string of Perl code as determined by the block rule, which is one of the core building blocks of the grammar.

We want UNTIL to call a method with the same signature that WHILE calls, so we can duplicate the appropriate lines in the loop rule:

```
loop:   FOR loopvar ';'        { $_[0]->{ INFOR }++               }
          block END            { $_[0]->{ INFOR }--;
```

```
                            $factory->foreach(@{$_[2]}, $_[5])  }
        |   atomexpr FOR loopvar    { $factory->foreach(@{$_[3]}, $_[1])  }
        |   WHILE expr ';'          { $_[0]->{ INWHILE }++                }
              block END            { $_[0]->{ INWHILE }--;
                            $factory->while(@_[2, 5])          }
        |   atomexpr WHILE expr     { $factory->while(@_[3, 1])          }
        |   UNTIL expr ';'          { $_[0]->{ INWHILE }++                }
              block END            { $_[0]->{ INWHILE }--;
                            $factory->until(@_[2, 5])          }
        |   atomexpr UNTIL expr     { $factory->until(@_[3, 1])          }
;
```

This points to the currently nonexistent until method of Template::Directive; let's add it. Open *lib/Template/Directive.pm* and find the while method. Because UNTIL is logically equivalent to WHILE NOT, while is where we need to start looking, and in fact, we can duplicate it almost in its entirety:

```
sub until {
    my ($class, $expr, $block) = @_;
    $block = pad($block, 2) if $PRETTY;

    return <<EOF;

# UNTIL
do {
    my \$failsafe = $WHILE_MAX;
LOOP:
    while (--\$failsafe && !($expr)) {
$block
    }
    die "UNTIL loop terminated (> $WHILE_MAX iterations)\\n"
        unless \$failsafe;
};
EOF
}
```

We can copy the while method, and change the name of the subroutine and the name of the directive (in case anyone looks at the generated code), as well as modify the loop expression, from:

```
while (--\$failsafe && ($expr)) {
```

to:

```
while (--\$failsafe && !($expr)) {
```

And we're finished inside Directive.pm.

The last change is one of the most important—we need to tell the grammar that UNTIL is now a reserved word. In *parser/Grammar.pm.skel*, add UNTIL to the @RESERVED array:

```
@RESERVED = qw(
        GET CALL SET DEFAULT INSERT INCLUDE PROCESS WRAPPER BLOCK END
```

```
        USE PLUGIN FILTER MACRO PERL RAWPERL TO STEP AND OR NOT DIV MOD
        IF UNLESS ELSE ELSIF FOR NEXT WHILE SWITCH CASE META IN
        TRY THROW CATCH FINAL LAST RETURN STOP CLEAR VIEW DEBUG
        UNTIL
    );
```

Now, we're ready to re-create the grammar, and start testing!

```
$ ./yc
Compiling parser grammar (Parser.yp -> ../lib/Template/Grammar.pm)
```

When making any changes to the grammar, it is important to go back to the root of the distribution and run make test, to ensure that your changes didn't accidentally break anything else. It is also a good idea to write some new tests to both illustrate and test your new functionality.

8.5.3 Replacing the Default Grammar

It is possible to completely replace the existing grammar with something radically different. Generally, this requires not only the appropriate *Grammar.pm* file, but also a Template::Directive-style factory class that knows how to emit the code to implement your new language.

8.5.3.1 Template::Simple

The Template::Simple module implements a simple template language for use with the Template Toolkit.[4] It really is simple compared to the regular Template Toolkit language. It allows you to access variables and nothing else. No directives. No INCLUDE, no IF, no FOREACH. Nothing.

> [4] Template::Simple is available via anonymous CVS at *cvs -d:pserver:cvs@tt2.org:/Template-Simple co Template-Simple*.

However, all of the functionality for accessing variables is available. You can use scalars, lists, hash arrays, subroutines, and objects, and you can call virtual methods. There is no SET directive, either implicit or explicit, so you cannot update or create new variables.

```
# simple vars
[% name %] is an inhabitant of [% planet %].

# complex vars
[% friends.0 %] and [% friends.1 %] are his friends.

# virtual methods
[% friends.join(' and ') %] are still his friends.
```

You can emulate existing directives by binding subroutines to variables that make the appropriate calls to the Template::Context object:

```
my $ts = Template::Simple->new( );
my $tc = $ts->context( );

my $vars = {
    name    => 'Arthur Dent',
    planet  => 'Earth',
    friends => [ 'Ford Prefect', 'Slartibartfast' ],
```

```
    include => sub { $tc->include(@_) },
};
```

Then you access the subroutine via the include variable, passing the template name and local variables as arguments:

```
[% include( 'person/summary',
          name   = 'Slartibartfast'
          planet = 'Magrethea' )
%]
```

The Template::Simple module is a very thin wrapper around the Template module. All it does is set the GRAMMAR configuration option to Template::Simple::Grammar. Most of the other Template Toolkit options can be passed to the Template::Simple constructor. However, any options that relate to directives that are no longer implemented will be ignored (e.g., PLUGINS, FILTERS, etc.).

8.5.3.2 The Template::Simple grammar

The heart of Template::Simple is the grammar, which is built from *Parser.yp*. Template::Simple's full grammar is relatively simple, and consists of a small set of core tokens (TEXT, IDENT, COMMA, LITERAL, NUMBER, DOT, ASSIGN) and a few more complex rules built up from these tokens.

Example 8-16 is the complete Template::Simple grammar. To read the grammar, start at the top—the first rule is the implicit "start" rule, from which the parser commences. Thus, the main rule in this grammar is template. $factory is the Perl factory, Template::Directive by default, that is used to generate Perl code that will eventually be transformed into the Template::Document instance (refer to Chapter 7 for all the details).

Example 8-16. Template::Simple grammar

```
%%

template:   block           { $factory->template($_[1]) }
;

block:      chunks          { $factory->block($_[1]) }
        |   /* NULL */      { $factory->block( )     }
;

chunks:     chunks chunk    { push(@{$_[1]}, $_[2])
                              if defined $_[2];
                              $_[1]
                            }
        |   chunk           { defined $_[1]
                              ? [ $_[1] ]
                              : [ ]
                            }
;

chunk:      TEXT            { $factory->textblock($_[1]) }
        |   statement ';'
```

```
;

statement:  term              { $factory->get($_[1]) }
        |   /* empty */
;

term:       ident             { $factory->ident($_[1])  }
        |   '"' quoted '"'    { $factory->quoted($_[2]) }
        |   LITERAL
        |   NUMBER
;

ident:      ident DOT node    { push(@{$_[1]}, @{$_[3]});
                                $_[1]
                              }
        |   ident DOT NUMBER  { push( @{ $_[1] },
                                    map { ($_, 0) }
                                    split(/\./, $_[3]) );
                                $_[1]
                              }
        |   node
;

node:       item              { [ $_[1], 0 ]                    }
        |   item '(' args ')' { [ $_[1], $factory->args($_[3]) ] }
;

item:       IDENT             { "'$_[1]'"                       }
        |   '${' term '}'     { $_[2]                           }
        |   '$' IDENT         { $factory->ident(["'$_[2]'", 0])  }
;

args:       args term         { push(@{$_[1]}, $_[2]);
                                $_[1]
                              }
        |   args param        { push(@{$_[1]->[0]}, $_[2]);
                                $_[1]
                              }
        |   args COMMA        { $_[1]                           }
```

```
        |  /* init */        { [ [ ] ]                     }
;

quoted:     quoted quotable   { push(@{$_[1]}, $_[2])
                                  if defined $_[2];
                                $_[1]
                              }
        |  /* NULL */        { [ ] }
;

quotable:   ident             { $factory->ident($_[1]) }
        |  TEXT              { $factory->text($_[1])  }
        |  ';'               { undef                  }
;

param:      LITERAL ASSIGN term  { "$_[1] => $_[3]" }
        |  item ASSIGN term     { "$_[1] => $_[3]" }
;

%%
```

Chapter 9. Accessing Databases

In many ways, the integration of a templating system and a database is natural. From e-commerce sites to Microsoft Word's MailMerge, database-backed template processing is very common. Indeed, this integration is one of the primary selling points of many systems, such as ASP and PHP.

You can integrate the Template Toolkit with a database in several ways. The most straightforward way is to simply use the DBI plugin. The DBI plugin is part of the standard Template Toolkit distribution, and provides a template-facing way to utilize Perl's DBI module (see *Programming the Perl DBI: Database Programming with Perl*, by Alligator Descartes and Tim Bunce (O'Reilly), for details about the DBI).

In addition to DBI, several database-related modules are on CPAN, such as Class::DBI and DBIx::SearchBuilder, that can be used to abstract the database layer out of code. Using these modules from within the Template Toolkit is the same as using them in Perl programs.

Writing your own abstraction layer is always an option as well. Many people like to keep SQL out of application code, for the same reasons that people prefer to keep business logic out of presentation templates; this is the primary purpose of a database abstraction layer. Many SQL-related helper modules are on CPAN, such as SQL::Abstract, SQL::OrderBy, SQL::QueryBuilder::Simple, and SQL::AnchoredWildcards, that can be used to help provide a non-SQL interface to a database.

9.1 Using the DBI Plugin

The DBI plugin provides direct access to the Perl DBI. The DBI provides a generic way of connecting to a database, and is the standard for using databases within Perl. The DBI plugin is a thin wrapper around DBI, with some Template Toolkit-specific modifications.

9.1.1 Simple Database Access with the DBI Plugin

In our first example of using the DBI plugin, we'll pull some data out of a MySQL database that contains details of a company's product range. Example 9-1 shows the template that we will use.

Example 9-1. Listing products

```
[% USE DBI('dbi:mysql:products', 'username', 'password') -%]
  Code  Name         Price   Stock
  [% FOREACH product = DBI.query('select ProductID, Name, Price, Stock from products') -%]
  [% product.ProductID | format('%05d') %] [% product.Name -%]
  $[% product.Price | format('%6.2f') %] [% product.Stock | format('%5d') %]
[% END -%]
```

The first thing to notice is the USE DBI directive, which is used to load the DBI plugin and connect to the database. The USE DBI directive takes a number of arguments. In this case, we pass it a string that identifies the data source that we want to connect to, together with the username and password that are required to make the connection.

The exact syntax of the data source identifier will vary depending on the type of the data source, but it will always start with the string dbi followed by a colon and the name of the connection type. In this case, as we are connecting to a MySQL database, we give it the string mysql followed by the name of the database that we wish to connect to (products). This usage assumes that the database is on the same server as the template processor. If it is on a different server, we can define that here by adding the hostname to the end of the data source identifier—for example, *dbi:mysql:products:db.company.com* would attempt to connect to the products database on the server *db.company.com*.

Having connected to the database, we can start to execute queries to access the required data. In this example, we will use the query function, which executes an SQL select query and returns the data a row at a time in a hash. The keys of the hash are the names of the columns selected. We assign each row in turn to the variable product, and can use that variable to access various parts of the returned row. Here are the results of processing the template in Example 9-1:

```
Code  Name         Price   Stock
00050 Basic Widget $ 49.99  2500
00051 Cheap Widget $ 29.99  5000
00101 Super Widget $ 99.99  1000
00102 Ultra Widget $149.99   500
```

Example 9-2 adds another level of complexity. Each product comes from a supplier; in this second report, we want to produce a list of each supplier followed by a sublist of the products that we get from the supplier.

Example 9-2. Listing products by supplier

```
[% USE DBI('dbi:mysql:products', 'username', 'password')
  suppliers = DBI.prepare('select SupplierID, Name from suppliers')
  products = DBI.prepare('select ProductID, Name, Price, Stock
                       from   products
                       where  SupplierID = ?')
```

```
-%]

[% FOREACH supplier = suppliers.execute -%]

[% supplier.Name %]

  [% FOREACH product = products.execute(supplier.SupplierID) -%]

  [% product.Name %]

  [% END %]

[% END -%]
```

For this, we will need two SQL queries to be active—one to list the suppliers and one to list the products. Additionally, the product query will need to take a parameter so that it returns only the products from the current supplier. To do this, we use the `prepare` method to precompile the two queries. Notice that the product query contains a clause, `where SupplierID = ?`. The `?` character marks a placeholder that will be filled in when we execute the query.

We then execute the suppliers query and process each returned row. As part of that processing, we execute the products query. The call to `products.execute` is passed the `SupplierID` for the current supplier record. Any arguments to `execute` are used as values to fill in the placeholders in the original SQL.

Here are the results of processing the template in Example 9-2:

```
Costcutter Widgets Inc.

    Basic Widget

    Cheap Widget



Quality Widgets Inc.

    Super Widget

    Ultra Widget
```

9.1.2 A More Complex Example: Web Access Logs

Having taken a look at a couple of simple templates that use the DBI plugin, it's now time to look at a more complex example. For this section, we will be using a table generated from a web server's access log (in Common Log Format). For simplicity, our examples will use `DBD::SQLite`—SQLite is a small, fast, embeddable, typeless RDBMS that implements most of SQL92, and includes advanced features such as transactions, triggers, and views. See http://www.hwaci.com/sw/sqlite/ for details about SQLite, and http://search.cpan.org/dist/DBD-SQLite/ for details about `DBD::SQLite`.

We will be using the following table definition:

access_log.sql

```
CREATE TABLE access_log (

    id INTEGER PRIMARY KEY,

    hostaddr VARCHAR,

    hostname VARCHAR,

    logname VARCHAR,

    req_time VARCHAR,

    request VARCHAR,

    uri VARCHAR,

    method VARCHAR,

    http_version VARCHAR,
```

```
    status VARCHAR,

    bytes_sent VARCHAR

);
```

The hostname field is generated by doing a DNS lookup of the hostaddr field (if it doesn't look like an IP address), and the uri, method, and http_version fields are parsed from the request field.

Example 9-3 shows the script that we used to get our file-based data into the database.

Example 9-3. Parsing log file entries

```
#!/usr/bin/perl -w


use strict;

$|++;


use DBI;

use Net::Nslookup qw(nslookup);

use Regexp::Common qw(net);


my $dsn = shift;

my $dbh = DBI->connect($dsn)

    || die "Can't connect to '$dsn': $DBI::err\n";


my $count = 0;

my $INSERT =<<'SQL';

    INSERT INTO access_log

        (hostaddr, hostname, logname, req_time, request,

         uri, method, http_version, status, bytes_sent)

    VALUES

        (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

SQL


while (<>) {

    my ($hostaddr, $logname, $remote_user, $req_time,

        $request, $status, $bytes_sent) = /^(\S+)     # host address

                                            \s+

                                           (\S+)     # remote logname

                                            \s+

                                           (\S+)     # username

                                            \s+

                                          \[(.+?)\]   # request time

                                            \s+
```

```
                      "(.+?)"    # request
                        \s+
                      ([\d-]+)   # status
                        \s+
                      ([\d-]+)   # bytes sent
                     /x;

    next unless $hostaddr;

    my ($method, $uri, $http_version) = split /\s+/, $request;

    my $hostname;
    if ($hostaddr =~ /$RE{net}{IPv4}/o) {
        $hostname = nslookup(host => $hostaddr, type => 'PTR');
    }
    else {
        $hostname = $hostaddr;
    }

    $dbh->do($INSERT, undef, $hostaddr, $hostname, $logname,
             $req_time, $request, $uri, $method, $http_version,
             $status, $bytes_sent)
        or warn "Error inserting line $.: " . $dbh->errstr;

    $count++;
    print '.' if (($count % 10) =  = 0);
    print "\n" if (($count % 700) =  = 0);
}

$dbh->commit;   # commit any outstanding lines
$dbh->disconnect;
```

Run the script with the DSN as the first argument, and an *access_log* on standard input:

```
$ logparse.pl dbi:SQLite:dbname=access_log < /home/www/logs/access_log
```

The script emits a dot character (.) for each 10 lines it inserts, breaking the output lines at 70 characters, mainly as a visual indication that it is still running (inserting thousands of entries can take a long time, after all).

With that out of the way, we can start using the DBI plugin. To connect to a database, pass the DSN to the USE DBI line in the template:

```
[% USE DBI('dbi:SQLite:dbname=access_log') %]
```

Or use the connect( ) method on a DBI object:

```
[% USE DBI %]

[% DBI.connect('dbi:SQLite:dbname=access_log') %]
```

Once we have a DBI object, we can use it to issue SQL statements:

```
[% log_entries = DBI.query('SELECT * FROM access_log') %]
```

The query method takes an SQL statement, which it issues against the underlying database, and returns an iterator that we can use to manipulate the data (see Example 9-4).

Example 9-4. Counting visitors

```
[% # Get a count of visits per address

  visitors = { };

  FOREACH log_entry IN log_entries;

    visitors.${log_entry.hostaddr} =

      visitors.${log_entry.hostaddr} + 1;

  END


  MACRO times(count)

    # "1 time" or "2 times"

    IF count =  = 1;

      "$count time";

    ELSE;

      "$count times";

    END

-%]

[% FOREACH visitor IN visitors.keys %]

  [% visitor %] visited [% times(visitors.$visitor) -%]

[% END %]
```

The simple template in Example 9-4 might give us something like the following:

```
134.174.141.2 visited 4 times

128.103.1.1 visited 1 time

206.33.106.134 visited 2 times

4.2.2.1 visited 3 times
```

Once we have the data, we can use one of the graph-generating plugins—for example, GD.Graph.pie, to generate a nice graph (see Example 9-5).

Example 9-5. Generating graphs

```
[%  USE graph = GD.Graph.pie(400, 300);

  FILTER null;

    data = [

      [ ]     # Array of addresses

      [ ]     # Array of visits
```

```
    ];

    FOREACH visitor IN visitors.keys;
      data.0.push(visitor);
      data.1.push(visitors.$visitor);
    END;

    dclrs = [ 'green' 'blue' 'red' 'cyan' ];
    graph.set(
      title    = 'Visits per address'
      transparent = 0,

      cycle_clrs  = 1
      dclrs       = dclrs
    );

    # plot data as a PNG, and send it to stdout
    # (recall the argument to the stdout filter
    # indicates that bindmode should be set).
    graph.plot(data).png | stdout(1);
  END;
-%]
```

Because the DBI plugin passes through to the underlying RDBMS, we can use any functions or stored procedures that database offers, as shown in Example 9-6.

Example 9-6. Using RDBMS-specific functions

```
[% query = DBI.query('SELECT sum(bytes_sent) as bytes_sent,
                      hostaddr FROM access_log group by
                      hostaddr'); %]
[% FOREACH hb = query %]
  We sent [% hb.bytes_sent %] bytes to [% hb.hostaddr %].
[% END %]
```

The query method returns an iterator that is similar, though not identical, to what is created within a FOREACH loop (the loop variable). This means that we have access to some of loop's methods, such as size, index, and max (see Example 9-7).

Example 9-7. Counting results

```
[% log_entries = DBI.query('SELECT hostaddr
                            FROM access_log
                            GROUP BY hostaddr');
-%]
There are [% log_entries.size %] unique addresses in the log.
```

Business folk like to have reports in CSV format so that they can manipulate the data in a spreadsheet program such as Excel or Gnumeric; producing a CSV file is pretty straightforward, as shown in Example 9-8:

Example 9-8. Producing a report as a CSV file

```
[% log_entries = DBI.query('SELECT * FROM access_log');
  FOREACH entry IN log_entries;
    FOREACH field IN entry.keys;
      field = entry.$field;
      field.match('[, ]') ? "\"$field\"" : field;
      "," UNLESS loop.last;
    END;
    "\n";
  END
-%]
```

If the field contains a comma (,) or a space, we quote it, using double quotes. Otherwise, it can be emitted as is.

Generating the report in XML format is similar, as shown in Example 9-9.

Example 9-9. Producing a report as XML

```
<?xml version="1.0" standalone="yes"?>
<access-log>
  [% log_entries = DBI.query('SELECT * FROM access_log') %]
  [% FOREACH entry IN log_entries %]
    <log-entry>
      [% FOREACH field IN entry.keys %]
        <[% field %]>[% entry.$field | html %]</[% field %]>
      [% END %]
    </log-entry>
  [% END %]
</access-log>
```

9.2 Using Class::DBI

Class::DBI is a convenient, easy-to-use database abstraction layer. It automates all the repetitive code that accompanies every database wrapper—accessors, mutators, constructors, search interfaces—and enforces efficient use of the DBI as well.

When using Class::DBI-based objects with the Template Toolkit, most of the work takes place in the Perl module that implements the class; once that part is written, templates can treat the object like any other variable. With Class::DBI taking care of most of the details of the database, fully functional modules can be implemented with very little actual code.

To illustrate using Class::DBI with the Template Toolkit, here is a very simple realestate application, based around a few tables:

```
CREATE TABLE listing (
    listing INTEGER PRIMARY KEY,
    location INTEGER,
    realtor INTEGER,
    price INTEGER,
    rooms INTEGER,
    bedrooms INTEGER,
    baths INTEGER,
    body VARCHAR
);

CREATE TABLE realtor (
    realtor INTEGER PRIMARY KEY,
    name VARCHAR,
    phone VARCHAR,
    url VARCHAR
);

CREATE TABLE location (
    location INTEGER PRIMARY KEY,
    city VARCHAR,
    state VARCHAR,
    postalcode VARCHAR
);
```

While this schema leaves obvious room for improvements, it will suffice for our needs. To illustrate how simple it is to integrate Class::DBI and the Template Toolkit, we'll start with a Class::DBI base class, as suggested in Example 9-10.

Example 9-10. Class::DBI

```
package TTBook::RealEstate::DBI;


use strict;

use vars qw($VERSION);

use base qw(Class::DBI);


TTBook::RealEstate::DBI->set_db('Main', 'dbi:SQLite:dbname=realestate.db');
```

This very simple module will be used as the base class by the other modules in our real-estate application. We set the main DSN here (the Main table)—it will be inherited by our subclasses.

The modules that sit on top of the listing, realtor, and location tables are almost as simple; they just need to declare the table upon which they sit, and list the columns in that table:

```
package TTBook::RealEstate::Listing;


use strict;

use base qw(TTBook::RealEstate::DBI);


# DB Table

TTBook::RealEstate::Listing->table('listing');


# Column groups

TTBook::RealEstate::Listing->columns(All =>

    qw(listing rooms body price bedrooms baths location realtor));


# Relationships with other objects

TTBook::RealEstate::Listing->has_a(location => 'TTBook::RealEstate::Location');

TTBook::RealEstate::Listing->has_a(realtor => 'TTBook::RealEstate::Realtor');
```

The TTBook::RealEstate::Listing table has relationships with data in other tables, and we indicate this with the has_a method. The TTBook::RealEstate::Realtor and TTBook::RealEstate::Location tables are very simple, and as a consequence can be represented very simply:

```
package TTBook::RealEstate::Realtor;


use strict;

use base qw(TTBook::RealEstate::DBI);


# DB Table

TTBook::RealEstate::Realtor->table('realtor');


# Columns

TTBook::RealEstate::Realtor->columns(All => qw(realtor name phone));
```

```
package TTBook::RealEstate::Location;

use strict;
use base qw(TTBook::RealEstate::DBI);

# DB Table
TTBook::RealEstate::Location->table('location');

# Columns
TTBook::RealEstate::Location->columns(All => qw(location city state postalcode));
```

Notice that these modules consist almost entirely of configuration, and not code. Such is the power of Class::DBI—only extraordinary situations require special-purpose code.

Using our new classes is simple. The simple CGI script in Example 9-11 either processes *listing.tt2* (if invoked with a listing_id parameter) or presents a search form, which will presumably call itself with a listing_id parameter.

Example 9-11. listing.cgi

```
#!/usr/bin/perl

use strict;
use warnings;

use CGI;
use Template;
use TTBook::RealEstate::Listing;

my $q = CGI->new( );
my $listing_id = $q->param('listing_id');
my $template = $listing_id ? 'listing.tt2' : 'form.tt2';

my $tt = Template->new( ) || die Template->error;

my $listing = TTBook::RealEstate::Listing->retrieve($listing_id);
$template = 'notfound.tt2' unless $listing;

my $vars = {
    'listing' => $listing,
```

```
};

print $q->header('text/html');

$tt->process($template, $vars)
    || die $tt->error;
```

Within *listing.tt2*, we can access methods of the listing variable (which is an instance of our Class::DBI subclass, TTBook::RealEstate::Listing) directly, as shown in Example 9-12.

Example 9-12. listing.tt2

```
[% USE wrap;
  realtor = listing.realtor;
  location = listing.location;
-%]
<h1>Look at this beautiful home in [% location.city %]!</h1>

[% PROCESS summary.tt2
  price       = listing.price
  rooms       = listing.rooms
  bedrooms    = listing.bedrooms
  baths       = listing.baths
%]

<p>
  [% listing.body | wrap %]
</p>

<p>
  For more information, contact [% realtor.name %] at
  [% realtor.phone %].
</p>
```

The *summary.tt2* template shown in Example 9-13 creates a simple table of attributes (price and number of rooms, bedrooms, and bathrooms). We can use the Template::Plugin::Number::Format plugin from CPAN,[1] to format the price nicely.

[1] You can find this plugin at http://search.cpan.org/dist/Template-Plugin-Number-Format/.

Example 9-13. summary.tt2

```
[% USE Number.Format %]

<table>

  <tr>

    <th>Price></th><td>[% price | format_price(0) %]</td>

  </tr>

  <tr>

    <th>Rooms</th><td>[% rooms %]</td>

  </tr>

  <tr>

    <th>Bedrooms</th><td>[% bedrooms %]</td>

  </tr>

  <tr>

    <th>Baths</th><td>[% baths %]</td>

  </tr>

</table>
```

The format_price filter takes a precision, which in this case we will set to 0—we probably don't need to see fractions of a quid when dealing with house prices.

It so happens that we can simplify our implementations even more. Because we are using SQLite for a database, our TTBook::RealEstate::DBI base class can subclass Class::DBI::SQLite instead of Class::DBI. Class::DBI::SQLite knows how to query the underlying SQLite database to get the schema for the appropriate tables automatically:

```
package TTBook::RealEstate::DBI;



use strict;

use vars qw($VERSION);

use base qw(Class::DBI::SQLite);



TTBook::RealEstate::DBI->set_db('Main', 'dbi:SQLite:dbname=realestate.db');
```

Using Class::DBI::SQLite enables us to simplify all of our subclasses, using the set_up_table method.[2] For example:

[2] This feature isn't specific to Class::DBI::SQLite; there are also versions for Oracle, Postgres, and MySQL.

```
package TTBook::RealEstate::Listing;



use strict;

use base qw(TTBook::RealEstate::DBI);



TTBook::RealEstate::Listing->set_up_table('listing');



# Relationships with other objects

TTBook::RealEstate::Listing->has_a(location => 'TTBook::RealEstate::Location');

TTBook::RealEstate::Listing->has_a(realtor => 'TTBook::RealEstate::Realtor');
```

9.3 Using DBIx::Table2Hash

The DBIx::Table2Hash module provides a simple way to turn a database table into a hash, turning SQL statements into simple lookups in a prepopulated table. DBIx::Table2Hash has methods to make this data available in a nested form as well as in a one-dimensional lookup table. While it doesn't allow for updates, it provides fast, convenient access to the data of a static table, such as a table containing postal codes and the cities to which they map. For this example, assume a simple table that looks like this (using SQLite again):

```
CREATE TABLE postal_code (
    code VARCHAR PRIMARY KEY,
    city VARCHAR
);
```

Using DBIx::Table2Hash, we can get a hash of our access_log data from within Perl like so:

```
my %args = ( dbh          => $dbh,
             table_name   => 'postal_codes',
             key_column   => 'city',
             value_column => 'code' );


my $t2h = DBIx::Table2Hash->new(%args)
my $data = $t2h->select;
```

Let's see how we can utilize this data. DBIx::Table2Hash expects to be passed a hash of items, including a connected database handle. Here's an example, adapted from the DBIx::Table2Hash documentation:

```
[% args = { dbh          = dbh
            table_name   = 'postal_code'
            key_column   = 'city'
            value_column = 'code' };


   USE t2h = Table2Hash(args);
   codes = t2h.select %]
```

The Template Toolkit will pass those hash values as a hashref, so we'll need to wrap this in a plugin.

Once we USE the plugin, we can call select, select_hashref, or select_tree to get our data. select returns a hash reference in which each element is a key_column => value_column pair (key_column and value_column are specified in arguments given to the constructor).

```
The postal code for Plymouth is [% codes.Plymouth %].
```

Keys with spaces in their names must be used indirectly:

```
[% ey = "East Yarmouth" -%]
The postal code for East Yarmouth is [% codes.$ey %].
```

select_hashref returns a hash of hashrefs, keyed by key_column:

```
[% codes = Table2Hash.select_hashref %]
[% FOREACH city = codes.keys %]
    [% city %] has postal code [% codes.$city.code %].
[% END %]
```

We've been ignoring where the dbh in this example comes from. There are several options here; for example, we could add code to TTBook::Template::Plugin::Table2Hash to accommodate a missing dbh parameter. Even simpler would be to use the DBI plugin:

```
[% USE DBI('dbi:SQLite:dbname=postal_codes.db');
   USE Table2Hash(dbh          = DBI.dbh
                  table_name   = 'postal_code'
                  key_column   = 'city'
                  value_column = 'code');
   codes = Table2Hash.select;
%]
```

The complete TTBook::Template::Plugin::Table2Hash: is shown in Example 9-14.

Example 9-14. TTBook::Template::Plugin::Table2Hash

```
package TTBook::Template::Plugin::Table2Hash;

use strict;
use vars qw($VERSION);
use base qw(Template::Plugin);

use DBIx::Table2Hash;

$VERSION = 1.00;

sub new {
    my ($class, $context, $args) = @_;
    my $dbix = DBIx::Table2Hash->new(%$args);

    return bless {
        _CONTEXT => $context,
        _T2H => $dbix,
        _ARGS => $args,
    } => $class;
}

sub select {
    my ($self, $args) = shift;
    return $self->{_T2H}->select(%$args);
}

sub select_hashref {
    my ($self, $args) = shift;
```

```
    return $self->{_T2H}->select_hashref(%$args);
}

sub select_tree {
    my ($self, $args) = shift;
    return $self->{_T2H}->select_tree(%$args);
}

1;
```

9.3.1 Writing Your Own Database Abstraction Layer

When all else fails, you can always write your own abstraction layer. Sometimes, this is the only alternative that makes sense. When dealing with content developers who have no understanding of SQL, it can be easier to provide them with a foolproof method of retrieving dynamic data from a database. Creating an abstraction layer to handle query generation also means that you can change the underlying database—for example, from SQLite to Postgres—without anyone having to know, and without any of the templates that access it having to be changed.

One of the most basic elements of a database abstraction layer is figuring out how to turn a collection of data into SQL. Luckily, several modules are on CPAN that do exactly that. My favorite is Nathan Wiger's SQL::Abstract (http://search.cpan.org/dist/SQL-Abstract/). This powerful module takes search critera as a hash, and transforms it into a WHERE clause.

We can create a search interface for the access_log database we defined earlier. Recall our access_log table:

```
CREATE TABLE access_log (
    id INTEGER PRIMARY KEY,
    hostaddr VARCHAR,
    hostname VARCHAR,
    logname VARCHAR,
    req_time VARCHAR,
    request VARCHAR,
    uri VARCHAR,
    method VARCHAR,
    http_version VARCHAR,
    status VARCHAR,
    bytes_sent VARCHAR
);
```

The key to creating a useable database query module is making it simple to use—you can't get much more powerful than DBI, but it is unintuitive for people who don't already know both SQL and Perl. SQL::Abstract is a small, powerful module with methods designed to generate SQL from a hash of parameters, such as those that might come in via a CGI form submission.

Ideally, we'll be able to provide a robust search interface, using only a few simple constructs in the template (see Example 9-15).

Example 9-15. Searching with the AccessLogSearch plugin

```
[%  # Our search plugin is called AccessLogSearch
  USE als = AccessLogSearch('dbi:SQLite:dbname=access_log');
```

```
  search.terms = {
    uri    = '*/index.htm?'
    status = 404,
    };

    fields = [ 'hostname' 'uri' 'status' ];

    results = als.query(fields, search.terms);
%]

Found [% results.size %] results for your search terms!
[% FOREACH result IN results %]
  ...
[% END %]
```

Given these search terms, results would contain all requests for *index.htm* or *index.html* pages that generated a status of 404 (Not Found). Note the * and ? wildcards, which make globbing simpler for users who might not know that % and _ are the SQL wildcard characters. More importantly, it abstracts the implementation; if we change the underlying data source to a different database, or to something other than database, the user-facing interface isn't coupled to an irrelevant wildcard convention.

We begin by subclassing the DBI plugin because it does almost all of what we want. Specifically, it handles connecting to the database and creating an efficient iterator object so that we don't have to read all of our results into memory.

```
package TTBook::Template::Plugin::AccessLogSearch;

use strict;
use vars qw($VERSION $DEBUG);
use base qw(Template::Plugin::DBI);

$VERSION = 1.00;
$DEBUG = 0 unless defined $DEBUG;

use SQL::Abstract;
use Template::Plugin::DBI;
```

The new method defers to the DBI plugin's new method, but also needs to create a SQL::Abstract instance:

```
sub new {
    my $class = shift;
    my $self = $class->SUPER::new(@_);
    my $sql = SQL::Abstract->new;
```

```
$self->{ _SQL } = $sql;


    return $self;

}
```

The AccessLogSearch plugin keeps a similar interface to the DBI plugin, but adds a little syntactic sugar to the query method:

```
[% # How many hits from Harvard's medical library this month?

   results = als.query('hostname' 'status' 'uri'

                     hostaddr = '134.174.151.*'

                     req_time = '%Aug%2003%);

%]
```

The new query method handles these criteria easily: name => value pairs are search parameters, and any other values are the fields to be selected:

```
sub query {

    my ($self, @fields) = @_;

    my $terms = ref($fields[-1]) eq 'HASH' ? pop(@fields) : {  };

    my ($sql, @bind, $sth, $result, @results);
```

We can specify the fields that we want back, such as hostname, uri, and status, but if fields is empty, we use *, which means to select all fields. If the user passes in an array from the template, it will come to our method as an arrayref, so we dereference it here.

```
@fields = ('*') unless @fields;

@fields = @{$fields[0]} if ref($fields[0]) eq 'ARRAY';



$self->expand($terms);

($sql, @bind) = $self->{ _SQL }->select('access_log', \@fields, $terms);
```

If we are in $DEBUG mode—for example, during development—we emit the compiled SQL statement to the standard error stream, via the debug method (inherited from Template::Base, by way of Template::Plugin::DBI). Because SQL::Abstract generates SQL with placeholders, we need to fill them into the debugging string:

```
if ($DEBUG) {

    my @local_bind = @bind;

   (my $local_sql = $sql) =~ s/\?/"'" . shift(@local_bind) . "'"/eg;

    $self->debug("Generated SQL: '$local_sql'")

}
```

Now that we've generated the SQL, we can pass that to the DBI plugin's query method, which does the right thing—executes the query and returns a reference to an Iterator:

```
return $self->SUPER::query($sql, @bind);

    }
```

The expand method is responsible for turning * and ? into the SQL wildcards % and _ as shown here:

```
sub expand {

    my ($self, $terms) = @_;
```

```
    for my $term (keys %$terms) {
        my $like = 0;

        for ($terms->{$term}) {
            s/*/%/g && $like++;
            s/\?/_/g && $like++;
        }

        $terms->{$term} = $like ? { 'LIKE' => $terms->{$term} }
                                : { '='    => $terms->{$term} }
    }

    return $terms;
}
```

SQL::Abstract also knows how to deal with wildcard SQL, as long as we tell it to emit LIKE instead of =, so we count occurrences of the wildcard characters and use that to determine the appropriate test to use.

The complete TTBook::Template::Plugin::AccessLogSearch is shown in Example 9-16.

Example 9-16. TTBook::Template::Plugin::AccessLogSearch

```
package TTBook::Template::Plugin::AccessLogSearch;

use strict;
use vars qw($VERSION $DEBUG);
use base qw(Template::Plugin::DBI);

$VERSION = 1.00;
$DEBUG = 0 unless defined $DEBUG;

use SQL::Abstract;
use Template::Plugin::DBI;

# ----------------------------------------------------------------------
# new($context, @args)
#
# Pass @args directly to the superclass.
# ----------------------------------------------------------------------
sub new {
    my $class = shift;
    my $self = $class->SUPER::new(@_);
```

```
    my $sql = SQL::Abstract->new;

    $self->{ _SQL } = $sql;

    return $self;
}

# -------------------------------------------------------------------
# query(@fields, \%terms)
# -------------------------------------------------------------------
sub query {
    my ($self, @fields) = @_;
    my $terms = ref($fields[-1]) eq 'HASH' ? pop(@fields) : {  };
    my ($sql, @bind, $sth, $result, @results);

    @fields = ('*') unless @fields;
    @fields = @{$fields[0]} if ref($fields[0]) eq 'ARRAY';

    $self->expand($terms);
    ($sql, @bind) = $self->{ _SQL }->select('access_log', \@fields, $terms);

    if ($DEBUG) {
        my @local_bind = @bind;
       (my $local_sql = $sql) =~ s/\?/"" . shift(@local_bind) . ""/eg;
        $self->debug("Generated SQL: '$local_sql'")
    }

    return $self->SUPER::query($sql, @bind);
}

# -------------------------------------------------------------------
# expand(\%terms)
#
# Expand * and ? wildcards into SQL wildcards % and _.  Expects a
# reference to a hash, and operates on each value.  If a value is
# expanded, use LIKE instead of =.
# -------------------------------------------------------------------
sub expand {
    my ($self, $terms) = @_;
```

```
    for my $term (keys %$terms) {
        my $like = 0;

        for ($terms->{$term}) {
            s/*/%/g && $like++;
            s/\?/_/g && $like++;
        }

        $terms->{$term} = $like ? { 'LIKE' => $terms->{$term} }
                                : { '='    => $terms->{$term} }
    }

    return $terms;
}

1;
```

Chapter 10. XML

XML is becoming one of the most ubiquitous data formats. It is used for both data storage and data exchange. The Template Toolkit can be used to both create XML documents and convert them into other formats.

In this chapter, we'll take a look at some of the tools that the Template Toolkit provides for working with XML. We show how to populate template variables with fields from XML, how to generate XML, how to process RSS, how to extract information with the Document Object Model (DOM) and XPath, and even how to use XML transforms.

Before we get into some of the more complex tools for processing XML, let's start simply by looking at Template::Plugin::XML::Simple, which allows us to take a very simple approach to our XML.

10.1 Simple XML Processsing

Example 10-1 shows an XML file that contains details of a company's current inventory of widgets. We have each widget's part number, name, price, and current stock. This data might be generated by a stock control system.

Example 10-1. Stock control data

```
<inventory>

  <product id="0050">

    <name>Basic Widget</name>

    <price>49.99</price>

    <stock>2500</stock>

  </product>

  <product id="0051">

    <name>Cheap Widget</name>

    <price>29.99</price>

    <stock>5000</stock>

  </product>

  <product id="0101">

    <name>Super Widget</name>

    <price>99.99</price>

    <stock>1000</stock>

  </product>

  <product id="0102">

    <name>Ultra Widget</name>

    <price>149.99</price>

    <stock>500</stock>

  </product>

</inventory>
```

Suppose that we want to produce a report based on this data and also want to include the value of the stock. We can use the XML.Simple plugin to do this. Example 10-2 shows one way that we might do it.

Example 10-2. Template to create a stock report

```
[% USE inventory = XML.Simple('products.xml') -%]

[% FOREACH product = inventory.product.keys.sort;

   current = inventory.product.$product -%]

[% current.id %] [% product %]

[%- current.stock | format('%5d') %] units @

[%- current.price | format('%6.2f') -%] =
```

```
[%- current.stock * current.price | format('%10.2f') %]

[%- total = total + current.stock * current.price %]

[% END -%]

          Total value: [% total | format('%10.2f')%]
```

XML.Simple is given the name of an XML document and it builds a data structure that contains all of the data from that document. The USE directive returns a reference to this data structure, which we can then access using standard Template Toolkit techniques. In this case, the data structure it builds is a multilevel hash.

At the top level, the hash has only one key, product (representing the <product> tags from the original document). The value is a reference to another hash. The keys in this second hash are the names of the products, and the values are references to other hashes containing the details of the product. We can therefore use the expression inventory.product.keys.sort to get a list of the product names in alphabetical order.

To cut down on typing, we create a temporary variable, current, which contains the hash representing the current product. We can then access various parts of that hash to get the data that we want. Notice that we calculate the value of the current stock in each product and also keep a running total (in total) that we can display in the end. We also make use of the format filter to ensure that all of the numbers line up neatly.

The output generated by Example 10-2 is shown in Example 10-3.

Example 10-3. Generated stock report

```
0050 Basic Widget 2500 units @  49.99 =  124975.00

0051 Cheap Widget 5000 units @  29.99 =  149950.00

0101 Super Widget 1000 units @  99.99 =   99990.00

0102 Ultra Widget  500 units @ 149.99 =   74995.00

          Total value:  449910.00
```

For many tasks, XML.Simple is a perfectly adequate approach, however there will certainly be times when you need something that is a little more sophisticated. We'll look at XML.DOM and XML.XPath later in this chapter, but first we'll take a short detour to look at how we might create XML documents using the Template Toolkit.

10.2 Creating XML Documents

In order to demonstrate how to create XML documents using the Template Toolkit, we will use the example of creating an XML document that contains data about a TV show. Let's use (to pick a show at random) *Buffy the Vampire Slayer*.

10.2.1 Modeling Data About a TV Show

A TV show consists of a number of seasons. Generally, one season is made each year. Each season will have a regular cast. A season consists of a number of episodes. We want to create an XML file that contains all of this data.

We won't go into the details of how we access the data about the TV show. We'll just assume the existence of a module called TVShow.pm that will be our interface to details about a show. TVShow.pm has a constructor, new, which is passed the name of a show and returns an object that contains all of the data we need. It also has access methods that return all of these values.

We'll further assume the existence of Template::Plugin::TVShow, which allows us to use a TVShow object in our templates.

10.2.2 DTD for a TV Show

When designing an XML document, it's useful to create a *Document Type Definition* (or DTD) that defines what the XML document will look like. A DTD simply helps you to focus on the structure of the document. None of the Template Toolkit XML tools currently makes any use of the DTD.

Here's the DTD that we'll be using for our XML:

```
<!ELEMENT show (name, creator, seasons)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT creator (#PCDATA)>

<!ELEMENT seasons (season+)>

<!ELEMENT season (cast, episodes)>

<!ATTLIST season number CDATA>

<!ATTLIST season year CDATA>

<!ELEMENT cast (regular+)>

<!ELEMENT regular (character, actor)>

<!ELEMENT character (#PCDATA)>

<!ELEMENT actor (#PCDATA)>

<!ELEMENT episodes (episode+)>

<!ELEMENT episode (name, summary)>

<!ATTLIST episode number CDATA>

<!ATTLIST episode date CDATA>
```

While there are a large number of elements in this DTD, it isn't very complex. In English, the description looks something like this:

- A TV show consists of a name, a creator, and a list of seasons.
- A list of seasons consists of one or more seasons.
- A season consists of a cast and a list of episodes. It has two attributes—the season number and the year of broadcast.
- A cast consists of one or more regulars.

- A regular has a character name and an actor name.
- An episode list consists of one or more episodes.
- An episode has a name and a summary. It has two attributes—the episode number and the date of first transmission.

For more information on creating and interpreting DTDs, see *XML in a Nutshell* by Elliotte Rusty Harold and W. Scott Means, or *Learning XML* by Eric T. Ray (both by O'Reilly).

10.2.3 XML Template

Example 10-4 shows a simple template that will use the TVShow module to create an XML document conforming to our DTD.

Example 10-4. Sample template to create an XML document

```
[% USE show = TVShow(name) -%]
<?xml version="1.0"?>
<show>
  <name>[% show.name | html %]</name>
  <creator>[% show.creator | html %]</creator>
  <seasons>
    [%- FOREACH season = show.seasons %]
    <season number="[% loop.count %]"
      year="[% season.year %]">
      <cast>
      [%- FOREACH part = season.regulars %]
      <regular>
        <character>[% part.character | html %]</character>
        <actor>[% part.actor | html %]</actor>
      </regular>
      [%- END %]
      </cast>

      <episodes>
      [%- FOREACH episode = season.episodes %]
        <episode number="[% loop.count %]"
                date="[% episode.date %]">
          <name>[% episode.name | html %]</name>
          <summary>[% episode.summary | html %]</summary>
        </episode>
      [%- END %]
```

```
      </episodes>
    </season>
    [% END -%]
  </seasons>
</show>
```

This template takes one parameter, name, which can be passed in on the command line, so we can create a document for *Buffy the Vampire Slayer* using *tpage* like this:

```
$ tpage --define name='Buffy the Vampire Slayer' show.tt > show.xml
```

Example 10-5 shows the XML created. Repeated sections have been replaced with ellipses.

Example 10-5. XML document describing Buffy

```
<?xml version="1.0"?>
<show>
  <name>Buffy the Vampire Slayer</name>
  <creator>Joss Whedon</creator>
  <seasons>
    <season number="1"
            year="1997">
      <cast>
        <regular>
          <character>Buffy Summers</character>
          <actor>Sarah Michelle Gellar</actor>
        </regular>
        <regular>
          <character>Xander Harris</character>
          <actor>Nicholas Brendon</actor>
        </regular>

      ...

      </cast>

      <episodes>
        <episode number="1"
                 date="00:00:00 10-03-1997">
          <name>Welcome to the Hellmouth</name>
          <summary>Buffy Summers moves to Sunnydale</summary>
        </episode>
        <episode number="2"
```

```
            date="00:00:00 17-03-1997">

          <name>The Harvest</name>

          <summary>The Master plans to escape by harvesting people</summary>

        </episode>


...


      </episodes>

    </season>


...


  </seasons>

</show>
```

The template itself doesn't do anything complex. It simply uses access methods on the TVShow object to get the data that it needs. Notice that it uses the Date plugin to format the date and the loop.count variable to create the season and episode numbers.

Notice also that anywhere we are displaying text that could possibly include characters that have a special meaning in XML (&, <, >, or "), we use the html filter to convert these characters into their equivalent XML entity (&, <, >, and ", respectively).

10.3 Processing RSS Files with XML.RSS

Before we start looking at using the Template Toolkit to process arbitrary XML documents, let's take a look at a plugin that can be used to handle an industry-standard XML format: RSS.

RSS[1] is a method that web sites can use to exchange headlines and other data with each other. Web sites can produce RSS files that other web sites can periodically download and process. These files contain information that the subscriber web sites can display along with links to more detailed information on the publisher's web site. This gives the subscribers a relatively simple way to have frequently updated information on their web sites. A good example of this concept are the "slashboxes" that appear on the front page of http://slashdot.org/. You can get more information about RSS from *Content Syndication with RSS* by Ben Hammersley (O'Reilly).

> [1] RSS stands for Rich Site Summary, although exact translations of the abbreviation seem to vary on a daily basis.

An RSS file consists of a small number of tags that describe the web site that produced the file, together with a list of items. Example 10-6 is a sample RSS file. It is taken from CPAN and lists the most recent module uploads. You can see the most recent version of this file at http://search.cpan.org/rss/search.rss. We've removed all but two of the modules from the file to keep the example to a manageable size.

Example 10-6. Example RSS file from CPAN

```
<rss version="0.91">
<channel>
 <title>search.cpan.org</title>
 <link>http://search.cpan.org</link>
 <description>The CPAN search site</description>
 <language>en</language>
  <image>
   <title>searchDOTcpan</title>
   <url>http://search.cpan.org/s/img/cpanrdf.gif</url>
   <link>http://search.cpan.org</link>
   <width>88</width>
   <height>31</height>
   <description>All Modules, All the time</description>
 </image>
 <item>
  <title>DateTime-Format-Builder-0.62</title>
  <link>http://search.cpan.org/author/SPOON/DateTime-Format-Builder-0.62</link>
 </item>
 <item>
  <title>VCS-Lite-0.04</title>
  <link>http://search.cpan.org/author/IVORW/VCS-Lite-0.04</link>
 </item>
</channel>
</rss>
```

The structure of this file is easy to understand. The <channel> element contains a number of details about the web site providing the file in the <title>, <link>, <description>, and <language> tags. Then we see the <image> tag, which contains details of an image that we can use to illustrate our display of the information. Following this are a number of <item> tags, each of which includes information about one recently uploaded CPAN module.

The Template Toolkit's support for RSS is provided by Template::Plugin::XML::RSS, which is, in turn, a thin wrapper round Jonathan Eisenzopf's XML::RSS.

The RSS plugin makes it very simple to use RSS files in your templates. To use it, you need to add the line:

```
[% USE rss = XML.RSS(rssfile) %]
```

where rssfile is a variable that is set to the filename of the RSS file you want to use. You can then access individual items from the file using access methods on the rss object. Here is a very simple template to extract a list of the newest modules:

```
[% rss.channel.title -%]


[%- FOREACH item = rss.items %]
* [% item.title -%]
[% END %]
```

It's only a little more complex to build an HTML page, as shown in Example 10-7.

Example 10-7. Template to build HTML from an RSS file

```
[% USE rss = XML.RSS(rssfile) -%]
<html>
  <head>
    <title>[% rss.channel.title | html %]</title>
  </head>
  <body>
    <h1>[% rss.channel.title | html%]</h1>
    <p><a href="[% rss.image.link | html %]"><img
          src="[% rss.image.url | html %]"
          title="[% rss.image.title | html %]"
          alt="[% rss.image.title | html %]" /></a></p>
    <ul>
    [%- FOREACH item = rss.items %]
      <li><a href="[% item.link | html %]">[% item.title |html %]</a></li>
    [% END %]
    </ul>
  </body>
</html>
```

Notice that, as with the XML document we produced in the previous section, any text displayed is passed through the html filter to turn dangerous characters into HTML entities.

From processing one RSS file link, it's easy to move to processing a number of them on one page to create your own news page.

There is one slight complication with this scenario. You will find a number of different versions of the RSS file on the Internet. You will come across Versions 0.91, 0.92, 1.0, and 2.0.

The simple templates we've shown up to now will work with all versions equally well, but Versions 1.0 and 2.0 have a number of extensions that allow them to contain more information. The extensions in Version 1.0 are incompatible with those in 2.0. Luckily, the XML::RSS plugin gives us access to the version attribute from the RSS file, so our templates can make intelligent decisions on what data to expect to find.

For more details on support of the extensions to RSS 1.0 and 2.0, see the documentation for XML::RSS at http://search.cpan.org/dist/XML-RSS/.

10.4 Processing XML Documents with XML.DOM

There are a number of standards for XML document processing. One of the most popular is the DOM. The Template Toolkit supports this method through the plugin Template::Plugin::XML::DOM, which is, in turn, a thin wrapper around the XML::DOM module written by Enno Derksen.

Because the DOM is a mature standard, there are stable implementations of it in many languages. For this reason, it is very popular with programmers who often switch between different languages. XML::DOM parses the XML document into a tree structure that you can then query using a large set of defined method calls.

To demonstrate the use of the XML.DOM plugin, let's go back to the TV show XML document that we created earlier in this chapter. Example 10-8 shows a basic template that will transform that XML into an HTML page that describes a particular TV show.

Example 10-8. Creating HTML from XML using Template::Plugin::XML::DOM

```
[% USE date (format = '%d %b %Y') -%]
[% USE dom = XML.DOM;
     show = dom.parse('show.xml');
     name = show.getElementsByTagName('name').0.getFirstChild.getNodeValue
-%]
<html>
  <head>
    <title>[% name | html %]</title>
  </head>
  <body>
    <ul>
    [%- FOREACH season = show.getElementsByTagName('season');
              number = season.getAttribute('number') %]
      <li><a href="#season[% number %]">Season [% number %]</a></li>
    [% END -%]
    </ul>

    <h1>[% name | html %]</h1>
      <p>
        Created by
        [% show.getElementsByTagName('creator').getFirstChild.getNodeValue
         | html
         %]
      </p>

    [% FOREACH season = show.getElementsByTagName('season');
             number = season.getAttribute('number') -%]
    <h2><a name="season[% number %]">Season [% number %]</a>
```

```
    ([% season.getAttribute('year') %])</h2>

    <h3>Regular Cast</h3>
    <ul>
    [% FOREACH part = season.getElementsByTagName('regular', 1) -%]
    <li><b>[% part.getElementsByTagName('actor').getFirstChild.getNodeValue
           | html %]</b> as
      <i>[% part.getElementsByTagName('character').getFirstChild.getNodeValue
           | html %]</i></li>
    [%- END %]
    </ul>

    <h3>Episodes</h3>
    [%- FOREACH episode = season.getElementsByTagName('episode',1) %]
    <h4>[% episode.getAttribute('number') %] -
    [% episode.getElementsByTagName('name').getFirstChild.getNodeValue
         | html %]</h4>
    <p>
      <i>First broadcast
      [% date.format(episode.getAttribute('date')) %]</i><br />
      [% episode.getElementsByTagName('summary',1).getFirstChild.getNodeValue
         | html %]
    </p>
    [% END %]
    [% END %]
  </body>
</html>
```

The first thing to notice is that we parse the XML document in two stages:

```
[% USE dom = XML.DOM;
   show = dom.parse('show.xml') %]
```

On the first line, we create a DOM parser object called dom; on the second line, we use that object to parse our input file and create a DOM tree that we store in the variable show. We can then call various XML::DOM methods on this object to extract information about the show. You'll notice that you will often need to string several method calls together to get the information that you need. For example, to get the name of the show, we use the expression:

```
name = show.getElementsByTagName('name').0.getFirstChild.getNodeValue
```

The method getElementsByTagName returns a list of all of the elements that are children of the show element and have the name name. We then take the first node from that list (using the index 0) and get the first child of that node. This will be the text node that contains the name of the show. We can then use getNodeValue to get the value (i.e., the text) of that node.

As always, when we display any text extracted from the XML document, we pass it through the html filter to convert dangerous characters to their HTML entity equivalents.

The output from this code is shown in Example 10-9.

Example 10-9. HTML created from XML using Template::Plugin::XML::DOM

```
<html>
 <head>
  <title>Buffy the Vampire Slayer</title>
 </head>
 <body>
  <ul>
   <li><a href="#season1">Season 1</a></li>
  </ul>

  <h1>Buffy the Vampire Slayer</h1>
  <p>
   Created by
   Joss Whedon
  </p>

  <h2><a name="season1">Season 1</a>
   (1997)</h2>

  <h3>Regular Cast</h3>
  <ul>
   <li><b>Sarah Michelle Gellar</b> as
     <i>Buffy Summers</i></li>
   <li><b>Nicholas Brendon</b> as
     <i>Xander Harris</i></li>
  </ul>

  <h3>Episodes</h3>
  <h4>1 -
   Welcome to the Hellmouth</h4>
  <p>
   <i>First broadcast
     10 Mar 1997</i><br />
     Buffy Summers moves to Sunnydale
  </p>

  <h4>2 -
   The Harvest</h4>
```

```
    <p>
      <i>First broadcast
        17 Mar 1997</i><br />
        The Master plans to escape by harvesting people
    </p>
  </body>
</html>
```

You can get more details on using the DOM from the Template Toolkit by reading the module documentation for Template::Plugin::XML::DOM (at http://www.template-toolkit.org/docs/plain/Modules/Template/Plugin/XML/DOM.html) and XML::DOM (at http://search.cpan.org/dist/XML-DOM/). There is more information about the DOM standard in *XML in a Nutshell* by Elliotte Rusty Harold and W. Scott Means (O'Reilly).

As you can see, using the DOM to extract data from an XML document can get a little long-winded. Luckily, there are other ways to handle XML documents in the Template Toolkit. In the next section, we will look at another.

10.5 Processing XML Documents with XML.XPath

Another common standard for extracting data from XML documents is called *XPath*. XPath is structured vaguely like a filesystem path: consecutive elements are joined with a forward slash (/), beginning at the root, and each element in the path is nested below the previous. The XPath statement:

```
/html/head/title/text( )
```

retrieves "Welcome to Foo.com" from the following XML:

```
<html>

  <head>

    <title>Welcome to Foo.com</title>

  </head>

</html>
```

The Template Toolkit has support for XPath via the XML.XPath plugin, which wraps around Matt Sergeant's excellent XML::XPath module, available from CPAN (see http://search.cpan.org/dist/XML-XPath/). The XML.XPath plugin is given either the name of an XML document or a string containing XML.

Example 10-10 shows a template that uses the XPath plugin to create an HTML page from our XML file containing information about *Buffy the Vampire Slayer*. This is identical to the one we created in the previous section using the DOM (see Example 10-9).

Example 10-10. Creating HTML from XML using Template::Plugin::XML::XPath

```
[% USE date (format = '%d %b %Y') -%]

[% USE show = XML.XPath('show.xml') -%]

[% name = show.findvalue('/show/name/text( )') -%]

<html>

  <head>

    <title>[% name | html %]</title>

  </head>

  <body>

    <ul>

    [%- FOREACH season = show.findnodes('/show/seasons/season');

              number = season.findvalue('@number') %]

      <li><a href="#season[% number %]">Season [% number %]</a></li>

    [% END -%]

    </ul>


    <h1>[% name | html %]</h1>

    <p>

      Created by

    [% show.findvalue('show/creator/text( )') | html %]
```

```
  </p>

 [% FOREACH season = show.findnodes('/show/seasons/season');
          number = season.findvalue('@number') -%]
  <h2><a name="season[% number %]">Season [% number %]</a>
   ([% season.findvalue('@year') %])</h2>

  <h3>Regular Cast</h3>
  <ul>
  [% FOREACH part = season.findnodes('cast/regular') -%]
   <li><b>[% part.findvalue('actor/text( )') | html %]</b> as
     <i>[% part.findvalue('character/text( )') | html %]</i></li>
  [%- END %]
  </ul>

  <h3>Episodes</h3>
 [% FOREACH episode = season.findnodes('episodes/episode') -%]
  <h4>[% episode.findvalue('@number') %] -
     [% episode.findvalue('name/text( )') | html %]</h4>

  <p>
   <i>First broadcast
     [% date.format(episode.findvalue('@date')) %]</i><br />
     [% episode.findvalue('summary/text( )') | html %]
  </p>
 [% END %]
 [% END %]
 </body>
</html>
```

We are basically using three methods from the XML.XPath plugin. The line:

```
[% USE show = XML.XPath('show.xml') -%]
```

creates a new XML::XPath object based on the file *show.xml*. This object is a tree structure that models the XML structure of the XML document. We can then use the methods findvalue and findnodes to run XPath queries against this object. findvalue takes an XPath expression that will return a single value and returns the result of evaluating that expression. For example, we use:

```
[% name = show.findvalue('/show/name/text( )') -%]
```

to get the name of the show from the current document. The XPath query translates as "get the text for contained in the <name> element, which is a child of the <show> element, which is a child of the root." Any kind of XPath expression can be used. For example, we use @number to get the number attribute of the current node (which just happens to be an episode node at that point).

The findnode method is used to loop over a list of nodes. For example, we use:

```
[% FOREACH season = show.findnodes('/show/seasons/season') %]
```

to get each <season> node that is contained in the document, and use:

```
[% FOREACH episode = season.findnodes('episodes/episode') %]
```

to get each episode in a season. Notice that as findnodes returns a list of nodes, we use a variable to store each node in return as we work our way across the loop. These nodes are also XML::XPath objects and we can therefore run XPath queries against them in exactly the same way as we can with the original show object.

The current node that we are working from is called the *context node*. Continuing the filesystem analogy that we mentioned earlier, using a context node is like changing your current directory. Any XPath query that doesn't start with / is taken to be relative to your context node in the same way as a directory path that doesn't start with / is taken to be relative to your current directory. Any XPath query that starts with / is taken to be relative to the root node in the same way as a directory path that starts with / is taken as relative to the root directory.

You can get more details on using XPath from the Template Toolkit by reading the module documentation for Template::Plugin::XML::XPath (at http://www.template-toolkit.org/docs/plain/Modules/Template/Plugin/XML/Path.html) and XML::XPath (at http://search.cpan.org/dist/XML-XPath/). There is more information about the XPath standard in *XML in a Nutshell* by Elliotte Rusty Harold and W. Scott Means.

10.6 Processing XML Documents with XML.LibXML

All of the XML processors that we have seen up to now are based on the Perl module XML::Parser, which is, in turn, based on James Clark's *expat* XML parser. However, *expat* doesn't have support for newer XML features such as namespaces, so another parser has emerged as the first choice for many XML processing tasks. It is called *libxml2*, and you can find more details about it at http://www.libxml.org/.

Perl has a module, XML::LibXML, that gives access to the *libxml2* API, and Mark Fowler has written Template::Plugin::XML::LibXML, which allows the API to be used from the Template Toolkit. Both of these modules can be downloaded from CPAN at http://search.cpan.org/dist/XML-LibXML/ and http://search.cpan.org/Template-Plugin-XML-LibXML/, respectively.

libxml2 contains support for both DOM and XPath, so both of the previous examples will work almost unchanged. You will just need to alter the lines that load and parse the XML document.

10.7 Using Views to Transform XML Content

The XML processing methods that we have seen so far are very useful for *data-centric* XML documents. These are documents whose structure is very well-defined. This type of file is commonly seen when the file is modeling some kind of data structure, and is usually used for transferring data between different systems. The TV show example was a good example of this, as the relationships between the various data items in the document were well understood and unlikely to change.

There is another type of XML file, known as *narrative-centric*. In these files, the data is less well structured. A good example of this kind of document is a book. Although a book will have some high-level structure (table of contents, chapters, appendixes, and index), once you get down to the text in a chapter, the structure is much less defined. A paragraph can contain italic text, bold text, references to footnotes, URLs, and any number of other types of text, all of which will need to be processed differently.

While it is possible to handle these kinds of documents using the techniques we have seen previously, using the *VIEW* directive makes it far easier to process narrative-centric XML.

Example 10-11 shows a narrative-centric XML document.

Example 10-11. A narrative-centric XML document

```
<faq>
  <qna id="q1">
    <question>
      What is the ultimate answer to life, the universe and everything?
    </question>

    <answer author="Deep Thought">
      <para>42</para>
      <note>The problem may well be that you don't <i>actually</i>
         know what the question is!</note>
    </answer>
  </qna>

  <qna id="q2">
    <question>
      Where shall we have lunch?
    </question>

    <answer author="Milliways Marketing Dept.">
      <para>Have you considered <froody>Milliways</froody>, the restaurant
        at the end of the universe.</para>

      <quote>If you've done six impossible things today then why
        not top it off with dinner at Milliways?</quote>
```

```
    </answer>

  </qna>

</faq>
```

Notice that while the higher levels of the document are well structured, once you get into the answer tag, the text is unstructured. The para, note, and quote tags are used interchangeably, and other tags are used as well—you can see i and froody.

To process this file, we will create a *VIEW* called faq_html that will convert the FAQ to HTML. For our first attempt, we will create a "do nothing" view that will simply pass the document through unchanged. This view is shown in Example 10-12.

Example 10-12. faq_view1

```
[% VIEW faq_html

  notfound='passthru';



  BLOCK text;

    item;

  END ;



  BLOCK passthru;

    item.starttag;

    item.content(view);

    item.endtag;

  END;

END

%]
```

The *[% VIEW %]* directive defines a block that can contain other named blocks. In this *VIEW*, we defined two blocks. The first is called text. This is the default name for a block that will be called to process text nodes from the document. Our text block is simple and just displays the current item. Note that from within a VIEW template, the current node is available in the item variable and the current view is in the view variable.

The other block we defined is the block that is called if no matching block is found for a node. This is defined using the notfound parameter to the *VIEW* directive. Our passthru block displays the start and end tags for the node, and between them it calls the current node's content method, passing it the current view. The content method finds all of the current node's child nodes and displays them using the given view. This is an important method. If you want child nodes to be processed, your template must call it.

In order to use this template, we need to have a parsed XML document. VIEWs work well with any of the XML modules that we have seen before, but support for the XPath plugin is the most advanced. We can create and process an XML::XPath object with code like this:

```
[% USE doc=XML.XPath(file => 'faq.xml');

   node = doc.findnodes('/faq');

   faq_html.print(node) %]
```

Calling the print method on the VIEW and passing it the starting node starts the VIEW processing the document. Each type of node in the document is handled by the block with the same name. Any type of node that doesn't match a block in the VIEW is handled by the notfound block.

Currently our template has no named blocks, so all nodes are handled by the notfound block. We can add blocks that handle any nodes that need more than this default processing. Example 10-13 fills in processing for a number of tags.

Example 10-13. A more complex view

```
[% VIEW faq_html notfound='xmlstring' %]

[% BLOCK faq -%]
<h1>Frequently Asked Questions</h1>
[%- item.content(view) %]
[%- END %]

[% BLOCK question -%]
<h2>[% item.content(view) %]</h2>
[%- END %]

[% BLOCK answer %]
[% item.content(view) %]
<p>Answer by [% item.getAttribute('author') %]</p>
[% END %]

[% BLOCK para -%]
<p>[% item.content(view) %]</p>
[%- END %]

[% BLOCK note -%]
<p>Note: [% item.content(view) %]</p>
[%- END %]

[% BLOCK quote -%]
<blockquote><i>[% item.content(view) %]</i></blockquote>
[%- END %]

[% BLOCK qna;
  item.content(view);
  END;

  BLOCK text;
    item;
  END;

  BLOCK xmlstring;
    item.starttag;
```

```
    item.content(view);
    item.endtag;
  END
%]
[% END %]

[% USE doc = XML.XPath(file => 'faq.xml');
   node = doc.findnodes('/faq');
   faq_html.print(node)
%]
```

We should note a couple of points. First, we have created a block for the qna node, which does nothing but process its children. This is because if we left it to the default block, the opening and closing qna tags would be displayed, and we don't want that. Second, we haven't defined a block for the i tag. This is because we are happy for it to pass through unchanged, so it becomes part of the HTML page that is created.

Our input document also contains a froody tag. Currently this tag is passed through untouched (and presumably is ignored by the browser that displays the finished page). But when the management of Milliway's complain that their text should be displayed in a certain manner, it will be simple for us to add a block that handles it. For example:

```
[% BLOCK froody -%]
<font size="20" color="red"><i>[% item.content(view) %]</i></font>
[%- END %]
```

It is this extensibility that makes VIEW a perfect tool for processing narrative-centric XML documents. It is very simple to add processing for new tags, and it doesn't matter where they appear in the document structure.

Chapter 11. Advanced Static Web Page Techniques

In Chapter 2, we looked at some simple examples of using the Template Toolkit to generate web content. In this chapter, we will look at some more advanced techniques for building web sites and manipulating HTML page content. We will start out with a minimal setup that illustrates some useful techniques that can easily be adapted and applied to any web site. The basic system will be extended throughout the chapter as we add functionality to address more complex requirements and provide more advanced features.

The emphasis in this chapter will be on generating static HTML web content. The examples will be loosely based around the Template Toolkit web site, http://template-toolkit.org/. However, we're not going to be looking at content of any of the individual pages in any great detail, so the subject matter is largely immaterial.

Most of the techniques demonstrated are equally applicable to web sites delivering dynamically generated content and running web applications. More generally, this chapter shows how a general-purpose *presentation framework* can be built using the Template Toolkit. This can then be used to apply a consistent look and feel to all pages in a site, including static HTML pages (as discussed later in this chapter) and dynamic content (described in Chapter 12).

11.1 Getting Started

A few basic tasks need to be done when starting out a project for a Template Toolkit-driven web site. The first thing is to create somewhere for the project files to go. It's a good idea to keep everything related to the project in one place. If all the files are located in subdirectories of one common parent directory, the entire project can easily be relocated to another server, or perhaps to another directory on the same machine. It is much harder to keep track of files when they are dotted around a filesystem.

For this project, we will generate static HTML pages from templates. All the output files will be written to an *html* subdirectory of the project directory. From here they can be accessed via an appropriately configured web server. We'll be looking at a simple configuration for the Apache web server that demonstrates this.

The tool of choice for this kind of project is *ttree*. It also needs a configuration file detailing the various directories and other Template Toolkit options in effect. In this file, we can also specify which templates should be used as headers, footers, or wrappers to be automatically applied to each generated page. With these configuration files and standard templates in place, we can then begin to generate HTML pages.

So let's walk through the complete process, from creating the project directory to generating the first HTML page.

11.1.1 Directory Structure

The first task is to create a directory structure for the web site project. We'll be using */home/dent/web/ttbook* as the base directory in these examples:

```
$ mkdir /home/dent/web/ttbook

$ cd /home/dent/web/ttbook
```

Some further subdirectories are required underneath the new project directory:

```
$ mkdir bin etc templates html images
```

The directories follow a fairly standard naming convention. Here *bin* will be used to store executable programs or scripts to assist in building the site or performing other housekeeping tasks. The *etc* directory is for configuration and other miscellaneous files. The *templates* directory is for source templates from which HTML pages are generated. These are written to the *html* directory from where they are ready to be accessed by a web server, along with any images or other binary files for the site, stored under the *images* directory.

Two more subdirectories are required under the *templates* directory:

```
$ mkdir templates/src templates/lib
```

The *templates* directory is where most of the action takes places. The *templates/src* directory contains the source templates for the pages of the web site, or more generally, the site *content*. The *templates/lib* directory alongside it contains the library of general-purpose template components: headers, footers, menus, and so on. These typically relate to the user interface or *presentation* aspects.

You'll need to create further directories for content and component templates as we progress through the examples in this chapter. We'll assume from now on that you can do that without us having to tell you.

One final thing to note is that the names of templates cited in `INCLUDE`, `PROCESS`, and `WRAPPER` directives in these examples relate to files in the *templates/lib* directory, as defined in the `lib` option in *etc/ttree.cfg*. So a directive such as `[% PROCESS menu/item %]`, for example, refers to the *templates/lib/menu/item* template.

11.1.2 Web Server Configuration

The Template Toolkit isn't tied into any particular web server. At the simplest level, it is just a tool for generating content that can be read directly by a file editor or web browser, or can be served across a network by a web server. It operates independent of any delivery mechanism.

We will be using the Apache web server in these examples. A sample configuration file for Apache is shown in Example 11-1. This file should be created in the project *etc* directory.

Example 11-1. etc/httpd.conf

```
Alias /ttbook/images/   /home/dent/web/ttbook/images/

Alias /ttbook/          /home/dent/web/ttbook/html/
```

```
<Directory /home/dent/web/ttbook/>
    Options MultiViews Indexes FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

You will also need to edit your main Apache *httpd.conf* file (typically */usr/local/apache/conf/httpd.conf* or */etc/httpd.conf*) to Include the project configuration file. Example 11-2 shows the relevant line that is added for our configuration file, */home/dent/web/ttbook/etc/httpd.conf*.

Example 11-2. Addition to Apache httpd.conf configuration file

```
Include /home/dent/web/ttbook/etc/httpd.conf
```

You will need to restart Apache for these changes to take effect. For an Apache installation in */usr/local/apache*, the command would be as follows:

```
# /usr/local/apache/bin/apachectl restart
```

Another approach is to use symbolic links from an existing location that is already visible to the web server. For example, if the directory */home/dent/public_html/* can be accessed via the URL *http://localhost/~dent/*, you can create a symbolic link from here to the project *html* directory. On a Unix machine, the relevant command would be something like this:

```
$ cd /home/dent/public_html
$ ln -s /home/dent/web/ttbook/html ttbook
```

The *html* directory would then be accessible via the web server URL *http://localhost/~dent/ttbook/*.

Be warned that Apache doesn't follow symbolic links by default, so you'll need to add FollowSymLinks to the relevant section of the *httpd.conf* configuration file if you choose this approach:

```
<Directory /home/*/public_html>
    ...
    Options FollowSymLinks
    ...
</Directory>
```

With this directive in place, you can also use a symbolic link in the *html* directory to make the *images* directory accessible:

```
$ cd /home/dent/web/ttbook/html
$ ln -s ../images images
```

If you're not using Apache, you'll need to consult the documentation for your own web server to find out how to make the contents of the *html* and *images* directories accessible.

We'll assume in the following examples that the root document URL is */ttbook/* and the root images URL is */ttbook/images/*, in both cases assuming the default host, *http://localhost/*.

11.1.3 ttree Configuration

We need to provide a configuration file to tell *ttree* everything it needs to know to build the site content. Example 11-3 shows the complete file.

Example 11-3. etc/ttree.cfg

```
# directories
src  = /home/dent/web/ttbook/templates/src
lib  = /home/dent/web/ttbook/templates/lib
dest = /home/dent/web/ttbook/html

# copy images and other binary files
copy = \.(png|gif|jpg|pdf)$

# ignore CVS, RCS, and Emacs temporary files
ignore = \b(CVS|RCS)\b
ignore = ^#

# misc options
verbose
recurse
recursion

# TT options
pre_process = config/main
wrapper     = site/wrapper

# define some location variables
define rootdir = /home/dent/web/ttbook
define rooturl = /ttbook
define debug   = 0
```

The configuration file is very similar to the example we saw in Chapter 2. The first section defines the three important template directories:

```
# directories
src  = /home/dent/web/ttbook/templates/src
lib  = /home/dent/web/ttbook/templates/lib
dest = /home/dent/web/ttbook/html
```

The src directory contains the source templates for HTML pages. Each is processed by *ttree*, and the output is written to the corresponding file in the dest directory. The lib directory contains the library of various template components that don't comprise complete page templates in their own right. This directory is added to the INCLUDE_PATH option that *ttree* passes to the Template Toolkit. You can specify multiple lib directories in the configuration file, and each will be added to the INCLUDE_PATH in the order defined.

For now we plan to keep all images under the *images* directory, separate from the source templates in *templates/src*. However, there may be occasions when we want to put an image or other binary file in the same directory as an HTML page. To accommodate this, we set the copy option to a regular expression matching any filename extensions that indicate files that should be copied directly from *templates/src* to *html* without being processed through the Template Toolkit:

```
# copy images and other binary files
copy = \.(png|gif|jpg|pdf)$
```

We can also tell *tree* to look out for certain files that should be completely ignored—in this case, any CVS or RCS files that we may be using for version control, and also any temporary files that our favorite editor may have left lying around:

```
# ignore CVS, RCS, and Emacs temporary files

ignore = \b(CVS|RCS)\b

ignore = ^#
```

The next section sets some basic *ttree* flags:

```
# misc options

verbose

recurse

recursion
```

The first is verbose, which enables various useful messages so that we can see what's going on as *ttree* is doing its work. The second is recurse, which tells *ttree* to recurse into any directories it finds under the src directory and process any templates and further subdirectories it finds therein. The last option, recursion, is confusingly similar to recurse but serves a slightly different purpose. This tells the Template Toolkit that it's OK for a template to recursively process itself. Don't worry if you're not sure what that means right now. We're going to be using this feature later on when it comes to building a menu for the site, so all will become clear.

The next section defines two options that are passed to the Template Toolkit as the PRE_PROCESS and WRAPPER options:

```
# TT options

pre_process = config/main

wrapper     = site/wrapper
```

The pre_process option denotes that the *config/main* template should be preprocessed before each source page template. The wrapper option gives the name of a template that is used to provide a wrapper around the generated page output—in this case, to add HTML headers, footers, and any other user interface elements common to all pages in the site.

The final section defines two template variables that indicate the root directory for the project and the root URL for accessing the pages. The third defines a debug flag, which we'll leave disabled for now:

```
# define some location variables

define rootdir = /home/dent/web/ttbook

define rooturl = /ttbook/index.html

define debug   = 0
```

It is common (and sensible) practice to develop and test a web site offline, uploading it to its final URL only when it is finished and ready for public consumption. The only drawback to this is that the URLs you use to access pages under development will be different from those you use when the site goes live. One workaround to this problem is to use relative URLs when linking between pages. This approach works fine for small and simple sites but doesn't scale very well to larger, more complex sites, which can become more fragile when held together by relative links.

A better approach is to use a variable such as rooturl to define a root URL from which all other relative URLs in the site are constructed. If we need to relocate our site to be served under a different URL, we need only change this value and have *ttree* rebuild the site.

We'll see how this works in practice when we define some URLs a little later on in this chapter.

11.1.4 Simple pre_process and wrapper Templates

We now need to provide the pre_process and wrapper templates that were named in the *etc/ttree.cfg* configuration file.

For now we can just use some simple templates to get started and test that everything is working. The configuration template is shown in Example 11-4. It sets a single variable, msg. We will be displaying this value in a test page later on to demonstrate that the template is being preprocessed and the value correctly set.

Example 11-4. templates/lib/config/main

```
[% message = 'Hello World' -%]
```

The wrapper template displays the content inside a minimal set of HTML elements required for a valid HTML page (see Example 11-5).

Example 11-5. templates/lib/site/wrapper

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>[% template.title %]</title>
  </head>
  <body>
    [% content %]
  </body>
</html>
```

We'll start off by defining a title for each page in a META tag in the source template. In the *wrapper* template, this value is accessed as the template.title variable.

11.1.5 Creating the Build Script

Building the site is now a simple matter of invoking *ttree* using the -f option to tell it where to find the configuration file:

```
$ ttree -f /home/dent/web/ttbook/etc/ttree.cfg
```

The configuration file can be specified using an absolute filename as shown earlier, or a relative filename as shown in the following examples. Note the leading dot character (.) on the first example, which is required.

```
$ cd ~/web/ttbook
$ ttree -f ./etc/ttree.cfg    # OK
$ cd src
$ ttree -f ../etc/ttree.cfg   # OK
```

This can get a little tiresome when you have to type it several dozen times in a day, especially if the path to the configuration file is long and complicated. So to make life a little easier, we create a simple build script that calls *ttree* with the right -f option along with any other command-line arguments we specify, as shown in Example 11-6.

Example 11-6. bin/build

```
    ttree -f /home/abw/web/ttbook/etc/ttree.cfg $@
```

The build script is just a thin wrapper of convenience around *ttree* (for now). You can continue to use any of the usual *ttree* command-line options. For example:

```
$ bin/build             # build any modified pages
$ bin/build -a          # build all pages
$ bin/build index.html  # build just this page
$ bin/build -h          # show help
```

See Chapter 2 for further examples of using a build script.

11.1.6 A First HTML Page

With our basic presentation system in place, we can now start to create content for the web site. Each HTML page starts off as a source template in *templates/src*. All the headers, footers, menus and other user interface components are added automatically, so these templates need to provide only the core content for the page.

It is traditional to begin any demonstration such as this with the universal greeting to all of humanity. Example 11-7 shows a page template that displays the familiar "Hello World" message.

Example 11-7. templates/src/index.html

```
[% META title = 'Template Toolkit Test' %]



<p>

  This is the index page.  Testing!  Testing!

  <br />

  The message is '[% message %]'.

</p>
```

The page contains two directives. The first defines a title in a META tag. This value will then be displayed in the HTML head tag by the *templates/lib/site/wrapper* template that we defined earlier. The title is accessed, as are all META items, through the template variable—e.g., template.title.

The second directive prints the value of the message variable that we defined in the preprocessed *config/main* template.

Run *bin/build* to process the source template and generate the HTML page:

```
$ bin/build



ttree 2.63 (Template Toolkit version 2.10)



      Source: /home/dent/web/ttbook/templates/src

 Destination: /home/dent/web/ttbook/html

Include Path: [ /home/dent/web/ttbook/templates/lib ]

      Ignore: [ \b(CVS|RCS)\b, ^# ]

        Copy: [ \.(png|gif|jpg|pdf)$ ]

      Accept: [ * ]



  + index.html
```

The + to the left of index.html on the last line indicates that the file was processed successfully. This creates an *index.html* file in the *html* directory that looks like Example 11-8.

Example 11-8. html/index.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

  <head>

    <title>Template Toolkit Test</title>

  </head>

  <body>

    <p>

      This is the index page.  Testing!  Testing!

      <br />

      The message is 'Hello World'.
```

```
    </p>
  </body>
</html>
```

The source template has been processed and the [% message %] directive resolved to provide the familiar Hello World greeting. The page content has also been enclosed in the HTML wrapper template with the title of the page template (template.title) correctly inserted.

If your web server configuration is working as expected, you should now also be able to view this page as *index.html* under the root URL you specified (e.g., */ttbook/index.html*).

11.1.7 Automating the Project Configuration Process

If you take a look back over the files that we've created for the purposes of project administration—*bin/build*, *etc/ttree.cfg*, and *etc/httpd.conf*—you'll notice that all three reference the project root directory, */home/dent/web/ttbook*, and two use the base URL, */ttbook*. As when we want to move the project to another directory or URL, we need only edit these three files, and everything else should fall into place as part of the build process.

Three files may not sound like many, but that number will most likely grow as you add more functionality to your system. Sooner or later you'll relocate the site and forget to update one of the critical files. Much head scratching will ensue while you try to figure out why the site isn't building properly or the pages aren't being displayed.

If alarm bells aren't already ringing in your head, they should be because this is a perfect application area for some template processing. We said earlier that the Template Toolkit wasn't just for processing HTML, and this is a great example of what we mean. Rather than hardcoding a directory and URL in several configuration files, we can define them as templates, and have these and any other project-related variables inserted automatically to construct the build script and configuration files for us.

Here's how we do it. First, we create a directory for storing the skeleton templates for our project files. We'll call this directory *skeleton* to avoid confusing it with our HTML templates in *templates*. Under this directory, we also add *bin* and *etc* sub-directories.

```
$ cd /home/dent/web/ttbook
$ mkdir skeleton
$ mkdir skeleton/bin skeleton/etc
```

Copy the files *bin/build*, *etc/ttree.cfg*, and *etc/httpd.conf* (if you're using it, that is) into the relevant *skeleton* directories:

```
$ cp bin/build skeleton/bin
$ cp etc/ttree.cfg skeleton/etc
$ cp etc/httpd.conf skeleton/etc
```

Now use your favorite text editor to peform a global search for the project directory (e.g., */home/dent/web/ttbook*) and replace it with the Template Toolkit directive [% dir %]. Similarly, replace the base URL (e.g., */ttbook*) with [% url %]. Finally, replace the 0 for the debug value defined in *skeleton/etc/ttree.cfg* with [% debug %]. You can use Perl to do this if you prefer, using something like the following incantation:

```
$ perl -pi -e 's{/home/dent/web/ttbook}{[% dir %]}g; \
>            s{/ttbook}{[% url %]}g; \
>            s{(debug\s*=)\s*0}{$1 [% debug %]}' \
>            skeleton/*/*
```

The files should now look like those shown in Examples Example 11-9, Example 11-10, and Example 11-11.

Example 11-9. skeleton/bin/build

```
ttree -f [% dir %]/etc/ttree.cfg $*
```

Example 11-10. skeleton/etc/ttree.cfg

```
# directories
src  = [% dir %]/templates/src
lib  = [% dir %]/templates/lib
dest = [% dir %]/html

# copy images and other binary files
copy = \.(png|gif|jpg|pdf)$

# ignore CVS, RCS, and Emacs temporary files
ignore = \b(CVS|RCS)\b
ignore = ^#

# misc options
verbose
recurse
recursion

# TT options
pre_process = config/main
wrapper     = site/wrapper

# define some location variables
define rootdir = [% dir %]
define rooturl = [% url %]
define debug   = [% debug %]
```

Example 11-11. skeleton/etc/httpd.conf

```
Alias [% url %]/images/   [% dir %]/images/
Alias [% url %]/          [% dir %]/html/

<Directory [% dir %]/>
    Options MultiViews Indexes FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

Now all we need is a configuration script to figure out what the right values should be and process the templates. Another wrapper around *ttree* will do the job nicely, as shown in Example 11-12.

Example 11-12. bin/configure

```
#!/usr/bin/perl -w                                   # -*- perl -*-

#

# configure

#

# This script determines the correct root directory

# for the project (the parent of the 'bin' directory

# in which it is located), prompts for some configuration

# values if not set via command-line options, and then

# calls ttree to process all files under the skeleton

# directory, storing output relative to the project root

# directory (e.g., skeleton/bin/build => bin/build).

#

# Copyright 2003 Andy Wardley.

#

# This is free software distributed under the same terms as Perl.

#


use strict;

use warnings;

use FindBin qw( $Bin );

use Getopt::Std;

local $|=1;


# defaults

my $URL  = '/ttbook';


# get options

our ($opt_d, $opt_u, $opt_y, $opt_h);

getopts('yhdu:');


# display usage and exit on -h

die <<END_USAGE if $opt_h;

usage: configure [options]


options:

    -u url     url for HTML pages  (default: $URL)

    -d debug  set debug flag      (default: 0)

    -y         Accept defaults
```

```
    -h       This help
END_USAGE

# work out where we are in the filesystem
my @dirs = File::Spec->splitdir($Bin);
pop @dirs;  # remove 'bin'
my $dir  = File::Spec->catdir(@dirs);
my $skel = File::Spec->catfile($dir, 'skeleton');

# prompt for root URL
my $url  = prompt('root page URL', $opt_u || $URL);
my $dbg  = prompt('enable debugging?', $opt_d ? 'yes' : 'no')
         =~ /^y(es)?/ ? 1 : 0;

# hand over to ttree
my @args = ( 'ttree',
             '-r', '-p', '-v', '-a',
             '-s', $skel,
             '-d', $dir,
             '--ignore', '\b(CVS|RCS)\b',
             '--define', "dir=$dir",
             '--define', "url=$url",
             '--define', "debug=$dbg",
             @ARGV );

system(@args) =  = 0
    or die "ttree failed: $?\n";

#------------------------------------------------------------------------
# prompt($message, $default)
#
# Prompt user to input value or accept default.
#------------------------------------------------------------------------

sub prompt {
    my ($msg, $def) = @_;
    my $ans = '';
    $def = '' unless defined $def;
```

```
    print "$msg [$def] ";

    if ($opt_y) {     # accept default
        print "$def\n";
    }
    else {            # read user input
        chomp($ans = <STDIN>);
    }

    return length($ans) ? $ans : $def;
}
```

The script first determines the root directory of the project and then prompts the user for the base URL, defaulting to */ttbook*.

```
$ bin/configure
root page URL [/ttbook]
```

It also prompts the user to confirm the debugging option. Answer y or yes to set the debugging option, or press Enter to accept the default, leaving debugging disabled:

```
enable debugging? [no]
```

This flag doesn't have any effect on the Template Toolkit, although there are plenty of others that do. We're just defining another template variable, this time called debug, which we'll be using later.

The script then calls *ttree*, passing the various options required to have it process the files under the *skeleton* directory and copy the generated output into place under the project root directory:

```
ttree 2.63 (Template Toolkit version 2.10)

      Source: /home/dent/web/ttbook/skeleton
 Destination: /home/dent/web/ttbook
Include Path: [  ]
      Ignore: [ \b(CVS|RCS)\b ]
        Copy: [  ]
      Accept: [ * ]

  + bin/build
  + etc/ttree.cfg
  + etc/httpd.conf
```

The output files generated—*bin/build*, *etc/ttree.cfg*, and *etc/httpd.conf*—will contain exactly the same content as they did before. However, we can now easily move the project to a new directory or locate it under a different URL. Instead of editing the configuration files by hand, we let the *bin/configure* script take care of it.

An illustration of this is shown in the first line of the following example. Command-line options are used to define the new root URL (-u) and to accept all defaults (-y). The *bin/configure* script then regenerates the configuration files for the project. The second command then calls on the *bin/build* script to rebuild all the pages in the site (-a) using the new values defined.

```
$ bin/configure -u /newtturl -y
$ bin/build -a
```

Even the Apache configuration file, *etc/httpd.conf*, has been updated to account for the new URL, as shown in Example 11-13.

Example 11-13. etc/httpd.conf

```
Alias /newtturl/images/   /home/abw/web/ttbook/images/

Alias /newtturl/          /home/abw/web/ttbook/html/



<Directory /home/abw/web/ttbook/>

    Options MultiViews Indexes FollowSymLinks

    AllowOverride None

    Order allow,deny

    Allow from all

</Directory>
```

All you need to do is to restart Apache to have it read the new configuration. The web site will then be accessible via the URL *http://localhost/newtturl/*.

11.2 Library Templates

The templates for this project fall into two categories. Each HTML page has a corresponding source template in *templates/src* such as that shown in Example 2-1 in Chapter 2. These are referred to as *page templates* and generally map one-to-one with each static page in the site.

The other templates are *library templates*, also known as *template components*. Rather than defining complete HTML pages, these templates encode smaller chunks of HTML markup or Template Toolkit code to perform one task. We've also seen some simple examples of these in Examples Example 2-3 and Example 2-4. We're going to be looking at these in more detail now.

11.2.1 Configuration Templates

The purpose of the PRE_PROCESS configuration template, *config/main*, is to define any sitewide variables required to specify URLs, colors, images, and anything that we don't want to hardcode in the HTML page content or user interface components.

Rather than define everything in one monolithic configuration file, something that would quickly lead to a poor separation of concerns, a separate *config* directory will be used to contain various different configuration templates, each one representing one particular aspect of the site. These templates are loaded by one master template, *config/main*, shown in Example 11-14, using the PROCESS directive.

Example 11-14. templates/lib/config/main

```
[%  PROCESS config/page
          + config/site
          + config/url
          + config/col
          + config/images
-%]
```

This approach allows you to easily change one configuration file without affecting the others. This is particularly useful when you want to customize a web site to provide different presentation styles, a process known as *branding* or *skinning,* which we will be covering later in this chapter.

Now let us look at each configuration file in turn to find out what they do. The first, *config/page*, defines a page data structure containing various bits of information relating to the current page (i.e., source template) being processed. This is shown in Example 11-15.

Example 11-15. templates/lib/config/page

```
[%  USE Date;

   # define page data structure
   page = {
     file     = template.name
     title    = template.title
     about     = template.about
     type      = template.type or 'html'
     date      = template.date or Date.format(template.modtime)
```

```
   };


-%]
```

We're using the template variable here that references the Template::Document object for the current page template being processed (or about to be processed, given that this is all happening during the preprocess phase). Through the template variable we can access details about the template file itself, including the filename, template.name (specified relative to the *templates/src* directory in this case) and the modification time, template.modtime. Any metadata items defined in META tags within the template are also made available through the template variable—here we look specifically for title, about, and type. We also look for a date item, and otherwise construct human-readable data from the template modification time (template.modtime) formatted using the Date plugin.

The remaining templates define configuration data that relates to the site as a whole. The *config/site* template, shown in Example 11-16, defines a site data structure that contains some miscellaneous items.

Example 11-16. templates/lib/config/site

```
[%  site = {
       name      = 'Template Toolkit Web Site'
       server    = 'http://template-toolkit.org'
       admin     = 'webmaster@template-toolkit.org'
       copyright = '1996-2003 Andy Wardley'
   }
-%]
```

Example 11-17 shows the *config/url* template. This uses the rooturl variable to construct a set of page and section URLs that are stored in the site.url hash. Recall that the value for rooturl is defined as a ttree configuration option in the *etc/ttree.cfg* file.

Example 11-17. templates/lib/config/url

```
[%  site.url = {
       root    = rooturl
       home    = "$rooturl/index.html"
       images  = "$rooturl/images"
       logo    = "$rooturl/images/logo"
       css     = "$rooturl/css"
   }
-%]
```

The *config/col* template defines an rgb hash mapping color names to RGB hex triplets in the format required for HTML pages. This is also aliased to site.rgb. The template then defines a site.col hash that maps various style names to specific rgb colors. This is shown in Example 11-18.

Example 11-18. templates/lib/config/col

```
[%  rgb = {
       white    = '#FFFFFF'
       black    = '#000000'
       red      = '#ED2328'
       orange   = '#F08900'
       skyblue  = '#00AAF0'
```

```
        paleblue = '#80C0F0'
        midblue  = '#6080C0'
        darkblue = '#202060'
        misty    = '#C0C0F0'
        ltgrey   = '#E0E0E0'
        vltgrey  = '#F0F0F0'
    }

    site.rgb = rgb
    site.col = {
        back  = rgb.white
        text  = rgb.black
        link  = rgb.skyblue
        vlink = rgb.midblue
        alink = rgb.red
        mlink = rgb.orange
        line  = rgb.skyblue
        head  = rgb.darkblue
    }
-%]
```

The color names being used here are entirely arbitrary. It should be obvious that you can extend and adapt these and all the other data structures for your own use.

The *config/images* template, shown in Example 11-19, defines a site.image data structure containing some useful information about the logos that we're using in the site in various sizes.

Example 11-19. templates/lib/config/images

```
[%  site.image = {
        logo = {
            large = {
                src    = "$site.url.logo/tt2_180x60.gif"
                alt    = "TT2 Logo"
                width  = 180
                height = 60
            }
            small = {
                src    = "$site.url.logo/tt2_120x40.gif"
                alt    = "TT2 Logo"
                width  = 120
                height = 40
            }
```

```
        }
        name = {
            src    = "$site.url.logo/ttdotorg.gif"
            alt    = "template-toolkit.org"
            width  = 180
            height = 24
        }
    }

    site.logo = site.image.logo.large

-%]
```

The configuration templates collectively define two data structures: site and page. It is a good idea to define as few "top-level" variables like this as possible. The more variables you have, the harder it is to keep track of them, and the more likely you are to overwrite an important piece of predefined configuration data with a temporary or "local" variable of the same name.

Another benefit to this approach is that it allows us to replace the site or page data structures at a later date with alternate implementations. For example, we might decide to define the site data in an XML file, in an SQL database, or as a Perl module. All we have to do is arrange the data in the right format and make it available as the site and page variables, and it will integrate seamlessly into the existing structure.

Finally, defining all your sitewide data in a single site variable makes it easy to use compile-time constant folding at a later date if you need to optimize your templates for efficiency. As described in Chapter 3, the constant folding feature allows you to provide a set of variables in a namespace (constants by default, but you can easily change it to site, for example), which should be resolved once when the template is compiled instead of being resolved each time the template is processed. This can be particularly benefical when generating large amounts of template-driven dynamic content through a web server. It effectively gives each template less work to do each time it is processed by doing some of the work when the template is compiled.

11.2.2 Layout Templates

Now we can start to define the overall look and feel of the web site, using the same techniques that we introduced in Chapter 2.

11.2.2.1 Page wrappers

The wrapper option is used in the *etc/ttree.cfg* file to denote the name of a template that is used to automatically enclose the content generated from each page template. In this, the template is *site/wrapper*, shown in Example 11-20.

Example 11-20. template/lib/site/wrapper

```
[% content WRAPPER site/html + site/layout -%]
```

Two templates are being used to wrap the generated page content. The first and outermost wrapper in this case is *site/html*, shown in Example 11-21.

Example 11-21. templates/lib/site/html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
 <head>
  <title>[% page.title %]</title>
```

```
  <link rel="stylesheet" href="[% site.url.css %]/tt2.css" />
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
  <meta name="robots" content="all" />
 </head>

 <body bgcolor="[% site.col.back %]"
      text="[% site.col.text %]" link="[% site.col.link %]"
      vlink="[% site.col.vlink %]" alink="[% site.col.alink %]">
  [% content %]
 </body>
</html>
```

It adds the standard headers and footers required to construct a valid HTML page, interpolating a number of variables along the way. These include the page title from page.title and several colors from site.col.

Example 11-22 shows the second and innermost template, *site/layout*. It defines an overall layout for the page content and other sitewide user interface components.

Example 11-22. templates/lib/site/layout

```
<table width="100%" border="0" cellpadding="0" cellspacing="5">

  <tr valign="middle">
    <td width="[% site.logo.width + 10 %]" align="center">
      [% PROCESS site/logo %]
    </td>
    <td>
      [% PROCESS site/header %]
    </td>
  </tr>

  <tr>
    <td></td>
    <td>[% PROCESS misc/line %]</td>
  </tr>

  <tr valign="top">
    <td align="center">
      [% PROCESS site/menu %]
    </td>
    <td>
      [% content %]
    </td>
```

```
  </tr>

  <tr>
    <td></td>
    <td>[% PROCESS misc/line %]</td>
  </tr>

  <tr valign="bottom">
    <td></td>
    <td align="center">
      [% PROCESS site/footer %]
    </td>
  </tr>

</table>
```

It does this by combining them in an HTML table to define the overall layout, but leaves the implementation specifics of each element to be handled by other template components. This approach allows you to get a clear overview of the layout without the distraction of too much messy detail. Each component does just one thing, making it easy to understand, modify, or replace.

11.2.2.2 Layout components

Example 11-23 shows the other user interface components that we're using in the overall layout for the site.

Example 11-23. templates/lib/site/logo

```
<a href="[% site.url.home %]">
[%- INCLUDE misc/image image=site.logo | trim -%]
</a>
```

The *site/logo* template shown in Example 11-23 uses *misc/image* to generate an appropriate image tag. This has leading and trailing whitespace removed with the trim filter and is enclosed in an element making it a link to the site home page. The *misc/image* template in Example 11-24 simply generates an HTML image tag.

Example 11-24. templates/lib/misc/image

```
<img src="[% image.src %]" alt="[% image.alt %]"
     width="[% image.width %]" height="[% image.height %]" border="0" />
```

The *misc/line* template in Example 11-25 is so simple that you might wonder why we're using it at all. It contains only an hr element to create a horizontal rule (i.e., line) across the page.

Example 11-25. templates/lib/misc/line

```
<hr />
```

This example is rather trivial but it illustrates the principle of creating a library of reusable presentation components. They define a particular look and feel for the site that can easily be changed at a later date. Although it is slightly more tedious in this case to write [% PROCESS misc/line %] than to embed the <hr/> HTML element directly in a template, it has the benefit of allowing us to make it more complicated later.[1]

[1] As indeed we will, later on in this chapter.

As indeed we will, later on in this chapter.

Using a template component from the start to generate this feature means that we will have to make changes in only one place. When we do make a change, all the templates that use the component will get the benefit of the update. You don't have to generate your entire user interface like this, only the parts that you think you might want to do differently at a later date.

When you're designing the look and feel for a site, you'll probably want to try out a few different combinations of user interface elements in various styles, colors, positions, and so on. If you create each as a separate template component, you can easily switch between them to find something that you like. This is also ideal for showing different possibilities to your customer, manager, or whoever has the ultimate responsibility for how the site should look. They may not care too much about how the bike shed was built, but you can be sure they will have some opinion on what color it should be painted.[2]

[2] See http://www.unixguide.net/freebsd/faq/16.19.shtml for the origins of this analogy.

The *site/header* template is also very simple. It displays the page title and any information about the page, defined in page.title and page.about, respectively. This is shown in Example 11-26.

Example 11-26. templates/lib/site/header

```
<h1 class="title">[% page.title %]</h1>

[% IF page.about -%]

<div class="info">

  [% page.about %]

</div>

[% END -%]
```

We will be looking at generating menus and other navigation components later in this chapter. For now we'll start with something simple such as the template in Example 11-27, which provides a basic menu linking to various pages in the site.

Example 11-27. templates/lib/site/menu

```
[%  menu = {

      index = 'Home'

      about = 'About'

      news  = 'News'

    };



    order = ['index' 'about' 'news'];



    FOREACH item IN order;

-%]

<a href="[% site.url.root %]/[% item %].html">[% menu.$item %]</a>

<br />

[%  END -%]
```

Last but not least we have the *site/footer* template in Example 11-28. This adds a standard copyright message and some general information about the page.

Example 11-28. templates/lib/site/footer

```
<p class="info">
  &copy; Copyright [% site.copyright %].
  All Rights Reserved.
 <br />
  [% page.file %] last modified [% page.date %]
</p>
```

11.3 Content Templates

We now have a library of template components in place that defines a common configuration and presentation for our web site content. This is applied automatically by *ttree* for each page template it processes so that we don't have to worry about it. Our page templates can concentrate on defining core page content without being obscured by elements of the user interface.

11.3.1 HTML Pages

In Example 11-7, we saw how a title for a page can be defined in a META directive. In addition to this, we can now also provide an about item, as shown in Example 11-29.

Example 11-29. *templates/src/index.htm*

```
[% META title = 'Template Toolkit Home'
        about = 'Home page for the Template Toolkit'
%]

<p>
  Welcome to the Template Toolkit web site.
</p>

<p>
  This page would have more content but the editor
  is currently out enjoying an extended lunch break.
</p>

<p>
  We expect him back before the end of the year.
</p>
```

The title and about items are extracted automatically and displayed by the *site/header* template, along with the logo, menu, and footer. The rest of the template provides the page content, clean and simple.

Now you can run the *bin/build* script to generate the HTML output page:

```
$ bin/build
```

The output file *html/index.html* is generated. Figure 11-1 shows what it looks like when viewed using a web browser.[3]

[3] In the screenshots in this chapter, we have deliberately increased the size of the user interface in proportion to the page content. On the real site, the logo, menu, and other navigation components are much smaller, leaving more room for the core page content, which is of course the most important thing.

Figure 11-1. The generated HTML index page

The benefits of separating the common user interface elements from the core page content should by now be obvious. Adding a new page to the web site is a simple matter of adding a page template to the *templates/src* directory. These templates contain only the core content of the page, and authors don't need to concern themselves with adding headers, footers, menus, or anything else that is common to the site as a whole. The only requirement is that they define the title and about values in a META tag, although both of these are strictly optional. If they don't define either the title or about, the relevant page.title or page.about values will be empty. If we want to be more strict, we could easily modify our *config/page* template to throw an error if one or another was undefined.

You will of course need to run the *bin/build* script whenever you add new pages. Assuming they process without error, the generated HTML output pages will then be accessible via the relevant URL for your web server. When you're happy with the new pages, you can then go and update your *site/menu* template to make them accessible via the menu. Remember that you'll need to rebuild the entire site when you make a change to a sitewide component such as *site/menu*, so invoke *bin/build* with the -a option.

11.3.2 CSS and Other Non-HTML Pages

With the wrapper and layout templates in place, we can enjoy the benefits of having the user interface elements added automatically. However, there may be pages for which we don't want this window dressing automatically added. We're going to look at a Cascading Style Sheet (CSS) as an example of such a page, but the principle applies equally well to JavaScript libraries, text files, XML files, and so on.

We could just define these files outside of the *templates/src* directory so that they bypass the regular build process. We would of course need to manually copy them into the *html* directory or configure the web server to locate them correctly. Or we could store them in the *templates/src* directory along with all the other page templates, but add *css, js, txt*, and any other relevant file extensions to the copy option in the *etc/ttree.cfg* configuration file, indicating the files that should be copied into place rather than processed.

However, these approaches bypass the Template Toolkit processing stage, which isn't what we want in this case. We have already defined various colors in the pre-processed configuration template *templates/lib/config/col,* and we would like to use these values in the CSS file. Assuming then that we are going to process the CSS file through the Template Toolkit, we can take advantage of this by adding any other directives that will simplify the job of maintaining the document—for example, by defining font information in one place at the start of the file and then using it by variable reference in numerous different places throughout the file.

Example 11-30 shows the start of the CSS file to illustrate the principle. For a detailed discussion of CSS, see *Cascading Style Sheets: The Definitive Guide* by Eric Meyer (O'Reilly). As far as the Template Toolkit is concerned, it is just another text format.

Example 11-30. templates/src/css/tt2.css

```
[% META type = 'text' %]


[%  font = {

      text = 'Verdana, Arial, Helvetica, sans-serif'

      mono = '"Courier New", Courier, monospace'

    }

-%]


body {
```

```
    font-family: [% font.text %];
    font-size: 12px;
}

.info {
    font-size: 10px;
}

.title {
    font-family: [% font.text %];
    font-size: 24px;
    line-height: 30px;
    font-weight: bold;
    color: [% site.col.text %];
    margin-top: 0px;
    margin-bottom: 2px;
}

a {
    font-family: [% font.text %];
    font-size: 12px;
    line-height: 14px;
    text-decoration: none;
    color: [% site.col.link %];
}

a:hover {
    color: [% site.col.alink %];
}

a.menu {
    white-space:nowrap;
}

a.menu:hover {
    color: [% site.col.alink %];
```

```
}

a.menuselect {
    font-weight: bold;
    color: [% site.col.mlink %];
    white-space:nowrap;
}

a.menuselect:hover {
    font-weight: bold;
    color: [% site.col.alink %];
}

...etc...
```

The META directive in the first line declaring a text template type is the key to bypassing the usual wrapper mechanism. You may recall it was one of the items that the *config/page* template examined, in this case copying it into the page.type variable. The default value, if not explicitly set in a META directive, is html.

All that needs to be done is a quick change to the *site/wrapper* template to handle different values for page.type. This is shown in Example 11-31.

Example 11-31. templates/lib/site/wrapper

```
[%  SWITCH page.type;
      CASE 'text';
        content;

      CASE 'html';
        content WRAPPER site/html
                      + site/layout;
      CASE;
        THROW page.type "Invalid page type: $page.type";
    END;
-%]
```

If the page type is text, the page content is passed through unaltered. If the page type is html, we apply the usual wrappers. Otherwise we throw an error reporting that we can't handle pages of whatever unknown type they claim to have.

You can achieve the same effect in other ways without using a META item. For example, the *config/page* template could examine the template path or file extension to determine the file type, or consult a lookup table or database mapping filenames to type.

11.3.3 Content Components

As you develop more content for your site you'll undoubtedly find yourself doing the same kinds of things over and over again. At this point it might be a good idea to see whether you can isolate what you're doing and create a template component or components that do it for you.

We're going to look at an example of laying out information in a table. The HTML table element is a complex beast with many options, but we're not going to try and emulate or replicate it. Instead, we're going to define one particular table style and a few different cell styles according to the look and feel of our site.

The first thing we need to do is to define some colors for our table. Example 11-32 shows the definition of a site.col.table data structure, added to the bottom of the *config/col* template.

Example 11-32. templates/lib/config/col

```
[%
  .
  .
  .
  site.col = {
    .
    .
    .
    line  = rgb.skyblue
    head  = rgb.darkblue
    table = {
      edge = rgb.skyblue
      back = rgb.white
      head = rgb.misty
      cell = rgb.ltgrey
    }
  }
-%]
```

The *table/edge* template shown in Example 11-33 generates a table element nested inside another. This provide us with a colored border (site.col.table.edge) around the table. The template is designed to be used with the WRAPPER directive, so it expects the contents of the table to be defined in the content variable.

Example 11-33. templates/lib/table/edge

```
<table border="0" cellspacing="1" cellpadding="0"
       bgcolor="[% site.col.table.edge %]">
  <tr>
    <td>
      <table border="0" bgcolor="[% site.col.table.back %]"
             cellspacing="2" cellpadding="4">
        [%- content -%]
      </table>
    </td>
  </tr>
</table>
```

Here's a simple way in which you would use the template defined in Example 11-33:

```
[% WRAPPER table/edge %]
<tr>
 <th>Forename</th>
 <td>Arthur</td>
</tr>
<tr>
 <th>Surname</th>
 <td>Dent</td>
</tr>
[% END %]
```

The *table/row* template, also designed for use with WRAPPER, generates an HTML tr element with the content embedded inside. This is shown in Example 11-34.

Example 11-34. templates/lib/table/row

```
<tr valign="top">
  [% content %]
</tr>
```

The *table/head* and *table/cell* templates both generate HTML td elements, but use different background colors from the site.col.table hash (see Examples Example 11-35 and Example 11-36).

Example 11-35. templates/lib/table/head

```
<td bgcolor="[% site.col.table.head %]">
  <b>[% content %]:</b>
</td>
```

Example 11-36. templates/lib/table/cell

```
<td bgcolor="[% site.col.table.cell %]">
  [% content %]
</td>
```

Now we can use these different components to do the hard work of generating HTML tables in a consistent style.

11.3.4 Debugging Pages

When you're creating components such as these you'll want somewhere to test them and get them working just right. It's a good idea to create a separate directory in your site for doing this, but don't throw the test pages away when you're done. If you create a page for debugging a component or set of components in isolation, you can use it to check that the components are working as expected right now, as well as in the future, when you decide to change the layout style and modify the template components.

So let's start by creating a page for debugging the table components that we've just created. Example 11-37 shows three different examples of tables created using these components.

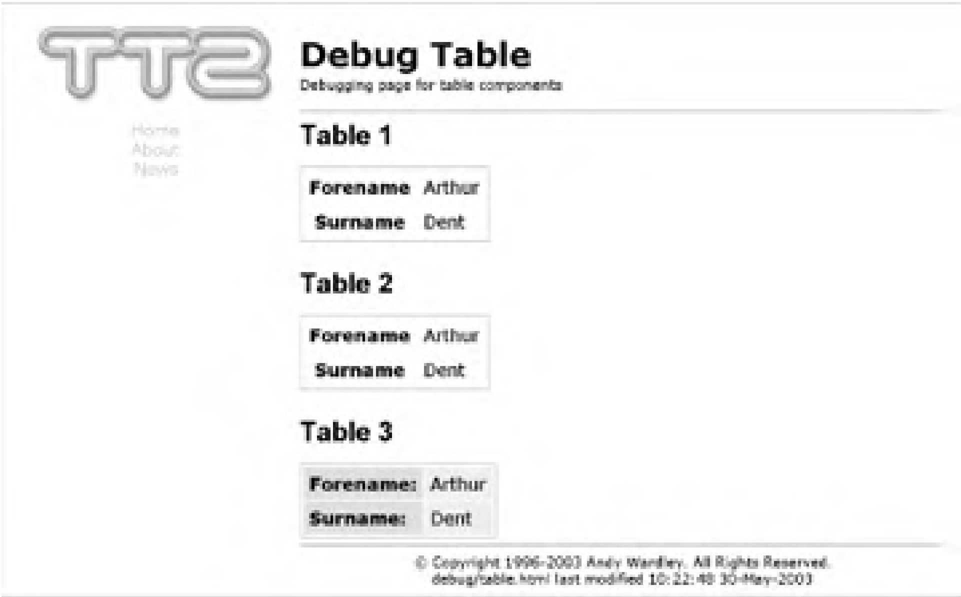
Example 11-37. templates/src/debug/table.html

```
<h2>Table 1</h2>

[% WRAPPER table/edge %]
  <tr>
    <th>Forename</th>
    <td>Arthur</td>
  </tr>
  <tr>
    <th>Surname</th>
    <td>Dent</td>
  </tr>
[% END %]

<h2>Table 2</h2>

[% WRAPPER table/edge %]
[%   WRAPPER table/row %]
       <th>Forename</th>
       <td>Arthur</td>
[%   END %]
[%   WRAPPER table/row %]
       <th>Surname</th>
       <td>Dent</td>
[%   END %]
[% END %]

<h2>Table 3</h2>

[% WRAPPER table/edge;
     WRAPPER table/row;
       INCLUDE table/head content='Forename';
       INCLUDE table/cell content='Arthur';
     END;
     WRAPPER table/row;
       INCLUDE table/head content='Surname';
       INCLUDE table/cell content='Dent';
```

```
    END;
  END
%]
```

Figure 11-2 shows the page generated by Example 11-37. Everything seems to be working as expected.

Figure 11-2. Debugging page for table components



Now let's add a page showing the contents of the site data structure. Or rather, let's write a *generic* template component that displays the contents of any hashlike data structure (see Example 11-38).

Example 11-38. templates/lib/debug/hash

```
[%  WRAPPER table/edge;
      FOREACH key = hash.keys;
          val = hash.$key;
        WRAPPER table/row;
          INCLUDE table/head content=key;
          WRAPPER table/cell;
            IF val.keys;
              INCLUDE debug/hash hash=val;
            ELSE;
              val;
            END;
          END;
        END;
      END;
    END;
-%]
```

Then all we need to do is to call the component passing the site data structure as the hash variable (see Example 11-

39).

Example 11-39. templates/src/debug/site.html

```
[% META title = 'Debug Site'
        about = 'Debugging page for the site data'
-%]


[% INCLUDE debug/hash hash=site %]
```

Figure 11-3 shows part of the page generated by the template in Example 11-39.

Figure 11-3. Debugging page for site data

It is a good idea to create a few debugging pages such as this that test any nontrivial template components you create. Whenever you make any changes to a component, you can check the relevant test page to ensure that it is still working as expected. Think of these pages as your test suite, designed to alert you quickly to any problems that may arise.

11.4 Navigation Components

Good navigation components are critical to making your web site accessible and allowing your visitors to find what they're looking for. A good general rule of user interface design is that a menu should have between three and seven items. Any more, and the user is faced with a daunting list of options to read through. Any fewer, and it's hardly a menu at all.

Given that a typical web site is likely to have more than seven pages, we need to consider how the pages and menus will be organized into some kind of structure. We'll look first at how a configuration file can be used to predefine this structure, automatically compute certain parts of it such as the URL for each page, and then make it accessible as part of the global site data. Then we'll create some template components that use this data structure to generate a menu and other navigation components.

We'll be keeping this example fairly simple so that we can concentrate on how the menus are constructed without getting bogged down in too much detail. Nevertheless, we will show how the site structure can be nested to any depth (within a reasonable limit), and also how it can be extended at runtime based on certain conditions, such as the value of the debug variable we set earlier in the *etc/ttree.cfg* file.

11.4.1 Adding Site Structure

The first rule of navigation is to have a good map.

Mapmaking is generally a laborious and time-consuming task, so we're going to get the Template Toolkit to do as much of the tedious work as possible. The map will be defined in the *config/map* template, so we need to modify the *config/main* template to PROCESS it, as shown in Example 11-40.

Example 11-40. Additions to templates/lib/config/main

```
[%  PROCESS config/page

         + config/site

         + config/url

         + config/col

         + config/images

         + config/map    # add this line

-%]
```

The *config/map* template is shown in Example 11-41.

Example 11-41. templates/lib/config/map

```
[%  # define map of pages in site

    map = {

        name = 'template-toolkit.org'

        menu = [ 'index', 'about', 'news', 'docs' ]

        page = {

            index    = { name = 'Home'     }

            about    = { name = 'About'    }

            news     = { name = 'News'     }

            docs     = {

                name = 'Documentation'

                menu = [ 'index', 'faq', 'manual' ]
```

```
            page = {
                index  = { name = 'Introduction' }
                faq    = { name = 'FAQ' }
                manual = {
                    name = 'Manual'
                    menu = [ 'index', 'syntax', 'directives' ]
                    page = {
                        index      = { name = 'Introduction' }
                        syntax     = { name = 'Syntax'      }
                        directives = { name = 'Directives'  }
                    }
                }
            }
        }
    }
};

IF debug;
    # add debugging pages
    map.page.debug = {
        name = 'Debug'
        menu = [ 'site' 'table']
        page = {
            site  = { name = 'Site'  }
            table = { name = 'Table' }
        }
    };

    # add debug item to main menu
    map.menu.push('debug');
END;

# save map in site
site.map = map;

# expand map recursively...
PROCESS config/expand;

-%]
```

11.4.1.1 Map nodes

The first section defines a nested map data structure:

```
map = {
    .
    .
    .
}
```

Each node in the map is represented by a hash array. This corresponds to a section or page in the site that has a unique location and a page associated with it. For example, the syntax page toward the bottom of the map correponds to the path *docs/manual/syntax.html* relative to the *templates/src* directory, and hence also to the */ttbook* URL or equivalent.

The one item that each node must contain is a name. This provides a short, readable name suitable for use in a menu.

```
syntax = { name = 'Syntax' }
```

If a node is a container for other pages, such as the manual node that contains the syntax page, the pages should be defined in a page hash:

```
manual = {
    name = 'Manual'
    menu = [ 'index', 'syntax', 'directives' ]
    page = {
        index      = { name = 'Introduction' }
        syntax     = { name = 'Syntax'       }
        directives = { name = 'Directives'   }
    }
}
```

The final addition is the menu item, also shown in this example. This defines the order in which the pages should be displayed in a menu. Remember that hash arrays don't retain the order of the items they contain, so we need to add this to make it explicit.

What we end up with is a complete page node that can be added to the page hash of a parent container:

```
docs = {
    name = 'Documentation'
    menu = [ 'index', 'faq', 'manual' ]
    page = {
        index  = { name = 'Introduction' }
        faq    = { name = 'FAQ' }
        manual = {
            .
            .  [ the manual node ]
            .
        }
    }
}
```

That node can then be added to another, which is added to another, and so on, until the complex site structure, or the part that is currently relevant to you, is defined.

11.4.1.2 XML site map

For a large site, the map could quickly become complex and difficult to maintain. However, you don't have to define it all at once, or all in the same place. You can just as easily store the information in an external XML file or SQL database and use one of the XML or DBI plugins to load it into place.

Example 11-42 shows how the same data information could be defined in an XML file.

Example 11-42. xml/sitemap.xml

```
<map>
  <name>template-toolkit.org</name>
  <menu>index</menu>
  <menu>about</menu>
  <menu>news</menu>
  <menu>docs</menu>
  <page id="index" name="Home" />
  <page id="about" name="About" />
  <page id="news"  name="News" />
  <page id="docs"  name="Documentation">
    <menu>index</menu>
    <menu>faq</menu>
    <menu>manual</menu>
    <page id="index"  name="Introduction" />
    <page id="faq"    name="FAQ" />
    <page id="manual" name="Manual">
      <menu>index</menu>
      <menu>syntax</menu>
      <menu>directives</menu>
      <page id="index"      name="Introduction" />
      <page id="syntax"     name="Syntax" />
      <page id="directives" name="Directives" />
    </page>
  </page>
</map>
```

Example 11-43 shows a variation of the *lib/map* template from Example 11-42. It uses the XML::Simple plugin to load the XML file and define the map variable. The KeyAttr parameter tells it to use the id attribute to index items.

Example 11-43. templates/lib/config/mapx

```
[%  USE map = XML.Simple(
        "$rootdir/xml/sitemap.xml"
```

```
        KeyAttr = ['id']
    );

    IF debug;
        # as before
        .
        .
        .

-%]
```

11.4.1.3 Selective mapmaking

Another approach to making a complex sitemap easier to maintain is to add bits in stages—for example, by defining the structure of each major section of the site in separate files. These can then be loaded via PROCESS and merged into a single map, much in the same way that we use several different configuration templates to build up the site data structure.

The next section of the *site/map* template shows one way this can be done. Here we define a submenu for our debugging pages, but only if the debug variable is set to true.

```
IF debug;
    # add debugging pages
    map.page.debug = {
        name = 'Debug'
        menu = [ 'site' 'table']
        page = {
            site  = { name = 'Site'  }
            table = { name = 'Table' }
        }
    };

    # add debug item to main menu
    map.menu.push('debug');
END;
```

If you want to enable the debugging pages, run bin/configure with the -d command-line option, or answer yes when prompted. Then run bin/build -a to rebuild the site with the debugging menu enabled.

The final part of the file saves the map structure in site.map and then calls *config/expand* to walk the map structure and expand it with additional items:

```
# save map in site
site.map = map;

# expand map recursively...
PROCESS config/expand;
```

11.4.2 Walking the Structure

The *config/expand* template is where all the deep magic behind our navigation system takes place. We're cramming a lot into a small space, and the template is rather complex as a result. In fact, this is probably the most complicated template that we're using to build the site.

Templates such as this often start simple and grow more complex as you develop the site further. For a real web site, we would probably implement this complex functionality as a Perl subroutine or plugin module. More likely, we would prototype it as a template and later implement it in Perl when we have a better idea about exactly what we want.

Nevertheless, we'll continue to use this as an example of the kind of complicated task that can be undertaken using the Template Toolkit, should you choose to do so.

Example 11-44 shows what the *config/expand* template looks like.

Example 11-44. templates/lib/config/expand

```
[%  # page.trail tracks path to the current page
    DEFAULT page.trail = [ ];

    # list of menu items we're constructing
    map.items = [ ];

    # walk through item names in map.menu
    FOREACH id IN map.menu;
        # fetch page from map.page
        THROW map "Invalid menu item in $map.name: $id"
            UNLESS (item = map.page.$id);

        # add location data
        item.id   = id;
        item.path = path ? "$path/$id" : id;
        item.file = item.page
                  ? "${item.path}/index.html"
                  : "${item.path}.html";
        item.url  = "$site.url.root/$item.file";

        # is this item on the path to the current page?
        item.hot  = page.file.match("^$item.path");
        item.subs = item.hot and item.menu;
        item.here = (item.file =  = page.file);

        # set next/last if this is the actual page
        IF item.here;
            page.prev = map.page.${loop.last};
            page.next = map.page.${loop.next};
```

```
        END;

        # add item to map items list
        map.items.push(item);

        # also to the trail if the page is hot
        page.trail.push(item) IF item.hot;

        # expand any submenu for this item
        IF item.subs;
           INCLUDE config/expand
              map  = item
              path = item.path;
        END;
    END;

-%]
```

It expects to be passed a map variable referencing a page node in the format defined in config/map. It walks through each page element defined within it in the order specified in the menu item. It calls itself recursively to process all the pages within pages within pages, to ensure that each node in the map is visited.

The purpose of visiting each node is to define additional data items that we are too lazy to add by hand. It's not just that we can't be bothered to go to the effort of adding relative paths, full URLs, and so on to each page. The real reason is that there is so much repetition of the same values that it's going to be tedious, time-consuming, and error-prone work that can be much better handled by a machine. Furthermore, some of these items are based on values that we will want to change from time to time (such as the base URL), so it makes sense to compute them at runtime.

Another reason for visiting each node is to construct an items list within the map that contains references to the pages in page in the order defined by menu. This will allow us to iterate directly through the page items in a map node in the correct order, without having to explicitly reference the page using an identifier each time. In other words, we're making life easier for ourselves later on.

The final reason is to determine which nodes are on the path to the current page and which pages, if any, come before or after the page in the menu. We'll be using this later to create a "bread-crumb trail" and links to the previous and next pages.

The list of page nodes on the path to the current file will be stored in page.trail, so the first thing *config/expand* does is to make sure it exists:

```
DEFAULT page.trail = [ ];
```

Then it creates a new items list in the current map node:

```
map.items = [ ];
```

Then it iterates through each page identifier, id, in the menu, map.menu:

```
FOREACH id IN map.menu;
    # fetch page
    THROW map "Invalid menu item in $map.name: $id"
        UNLESS (item = map.page.$id);
```

```
    .
    .
    .
END
```

It uses the identifier to index into the page map, map.page.$id, to fetch a page hash. This is then stored in the item variable, or an error is thrown if an invalid identifier is used. The id, path, file, and url items are then computed and added to item.

```
# add location data
item.id   = id;
item.path = path ? "$path/$id" : id;
item.file = item.page
          ? "${item.path}/index.html"
          : "${item.path}.html";
item.url  = "$site.url.root/$item.file";
```

Notice how the path variable is being used to construct the item.path, which is then used in item.file and item.url. We'll see how this works when we look at how the config/expand template calls itself recursively. But first, we should look at the other values that are added to each item.

```
# is this item on the path to the current page?
item.hot  = page.file.match("^$item.path");
item.subs = item.hot and item.menu;
item.here = (item.file =  = page.file);
```

The item.hot flag is set if the path to the item matches the beginning (or all) of the path for the current page being processed. In other words, it indicates that the node is on the path to the current page. For example, if the page.file variable contains the value docs/manual/index.html, the nodes marked as hot in the map would be docs, manual, and index, whose paths are docs, docs/manual, and docs/manual/index, respectively.

The item.subs flag goes a little further, indicating that the node is hot and also has further items contained within it. The last flag, item.here, indicates that the item is the actual node for the current page being processed.

If the item.here flag is set, we've found the node for the page we're processing, in which case we can set page.prev and page.next to point to the data structures for the previous and next pages:

```
# set next/last if this is the actual page
IF item.here;
   page.prev = map.page.${loop.last};
   page.next = map.page.${loop.next};
END;
```

The loop.last and loop.next variables provide us with the identifiers for the previous and next pages in the FOREACH loop. We use these to key into the map.page structure to fetch references to the hash arrays for the pages, if they exist.

Now that we've got a complete item we can add it to the map.items list:

```
# add item to list
map.items.push(item);
```

If the item is hot, we also add it to page.trail:

```
# also to the trail if the page is hot
page.trail.push(item) IF item.hot;
```

Then if the item.subs flag is set, the *config/expand* template recursively processes itself to expand the children and further descendants of the item:

```
# expand any submenu for this item
IF item.subs;
  INCLUDE config/expand
    map  = item
    path = item.path;
END;
```

The current item variable is passed as map and a new value for path is provided so that all the paths generated within it will be relative to the path for the current item.

As we already mentioned, this is perhaps the most complicated template in the site, so don't be surprised if you find it daunting. It can take a little time and patience to get something as complicated as this working properly, but it is usually something you have to do only once and can then forget.

It is also worth reiterating that when things start getting complicated, you can always recode in Perl and load the functionality in using a plugin, for example. That would certainly be the approach we would adopt if this template needed to become any more complex than it already is.

11.4.3 Building a Nested Menu

Now that we have a complete map defined, we can write a template that builds a menu from this data structure. Example 11-45 shows one way this can be done.

Example 11-45. templates/lib/menu/nest

```
[%  DEFAULT pad = '';

  FOREACH item = menu.items;
    pad;

    INCLUDE menu/text
     link = {
       text     = item.name
       url      = item.url
       class    = item.hot ? 'menuselect' : 'menu'
     };

    IF item.subs;
       "<br />\n";
       INCLUDE menu/nest
         menu = item
         pad  = pad ? "   $pad"
                    : " - ";
    END;

    "<br />\n";
```

```
  END
-%]
```

The *menu/nest* template also calls itself recursively to generate nested menus representing the structure of the site. For each invocation, the menu variable references the current site map node being processed. The pad variable contains a string used to indent each item by an amount appropriate to the current nesting depth.

The template iterates through each item in the menu.items list that now contains references to complete page structures, thanks to the work of the *config/expand* template:

```
FOREACH item = menu.items;
  .
  .
  .
END
```

Inside the loop, it prints the current pad string and then calls *menu/text* to generate a text link for the menu item:

```
pad;

INCLUDE menu/text
  link = {
    text    = item.name
    url     = item.url
    class   = item.hot ? 'menuselect' : 'menu'
  };
```

The *menu/text* template is passed a link hash that contains values extracted from the current menu item. The class value is set to correspond to one of the styles defined in the *templates/src/css/tt2.css* file, according to whether the item is hot and on the path to the current page. Example 11-46 shows the *menu/text* template.

Example 11-46. templates/lib/menu/text

```
<a href="[% link.url %]"
[%- " class=\"$link.class\""
      IF link.class
-%]
>[%- link.text -%]</a>
```

The final task of the *menu/nest* template is to process any nested items if the item.subs flag is set:

```
IF item.subs;
  "<br />\n";
  INCLUDE menu/nest
    menu = item
    pad  = pad ? "   $pad"
               : " - ";
END;
```

When the *menu/nest* template is called recursively, the item is passed as the new menu target and the pad is set to provide a deeper level of indenting.

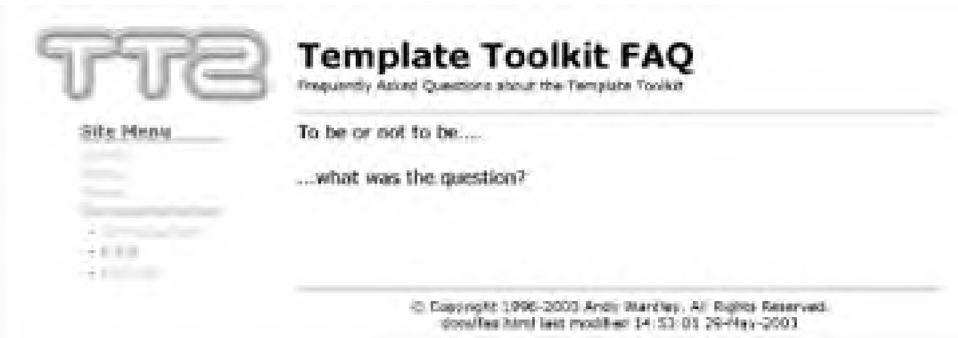
Now all we need to do is to modify the *site/menu* template to use the new *menu/nest* component, passing the top-level site map node, site.map, as the starting value for menu. While we're at it, we'll also add a title bar for the menu. Example 11-47 shows the changes made to *site/menu*.

Example 11-47. Changes made to templates/lib/site/menu

```
<table border="0" cellpadding="0" cellspacing="0">
  <tr>
    <td align="left" class="menutitle">
      Site Menu
    </td>
  </tr>
  <tr>
    <td>[% PROCESS misc/line %]</td>
  </tr>
  <tr valign="top">
    <td align="left">
      [% INCLUDE menu/nest
           menu = site.map
      -%]
    </td>
  </tr>
</table>
```

Figure 11-4 shows a screenshot containing the new menu.

Figure 11-4. Page with nested menu



Notice how the hot items in the menu are shown in bold orange[4] text as defined by the menuselect CSS style. Other menu items are displayed in the normal menu style.

[4] Not that you can tell in a grayscale image, but trust us, they're orange.

11.4.4 A Stacked Menu

The nested menu style works well when we need to nest menus that are only two or perhaps three levels deep. Any more than that and the menu will start to occupy more horizontal space that will cut into the page content.

We can easily create a new menu component that stacks menus on top of each other instead of nesting them. This is shown in Example 11-48.

Example 11-48. templates/lib/menu/stack

```
[%  pending = [ menu ];


    WHILE pending.size;

        menu = pending.shift;


        "<p>\n";

        FOREACH item = menu.items;

            PROCESS menu/text

              link = {

                text     = item.name

                url      = item.url

                class    = item.hot ? 'menuselect' : 'menu'

              };


            "<br />\n";


            pending.push(item)

              IF item.subs;

        END;

        "</p>\n";

    END;

-%]
```

The pending variable is used to keep a list of the menus that require processing, starting with the menu passed in as an argument, as per menu/nest:

```
pending = [ menu ];
```

The WHILE block repeats while there are menus in the pending list, removing the first menu in the list each time around:

```
WHILE pending.size;

    menu = pending.shift;



    .

    .

    .



END;
```

Other than adding a few HTML elements, the main part of the body of the WHILE block simply iterates over the items in the current menu, calling *menu/text* to process each:

```
FOREACH item = menu.items;

    PROCESS menu/text

      link = {
```

```
        text     = item.name
        url      = item.url
        class    = item.hot ? 'menuselect' : 'menu'
      };

    "<br />\n";

    pending.push(item)
      IF item.subs;
END;
```

When an item is found that has the subs flag set, it is added to the list of pending items. It will be processed after the current menu is complete, and will appear underneath it.

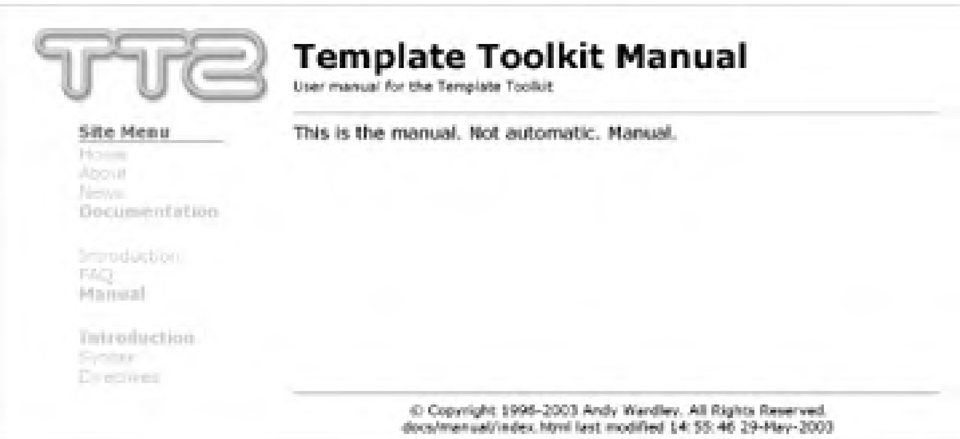
A quick change in *site/menu* from menu/nest to menu/stack is all that is required to use the new menu, as shown in Example 11-49.

Example 11-49. Changes to templates/lib/site/menu

```
  .
  .
  .
  <tr valign="top">
    <td align="left">
      [% INCLUDE menu/stack
           menu = site.map
      -%]
    </td>
  </tr>
</table>
```

Figure 11-5 shows a page with stacked menus.

Figure 11-5. Page with stacked menus



11.4.5 Bread-Crumb Trail

The name *bread-crumb trail* is borrowed by web developers from the story of Hansel and Gretel. They left a trail of bread-crumbs through the woods to help them find their way back from the wicked witch's edible house.[5] In the context of a web site, it refers to a commonly used navigation component that shows the steps a visitor has taken from the site home page down to the current page location.

[5] Alas, the hungry birds ate the bread-crumbs, but things turned out alright for them in the end.

The config/expand template has already stored the list of hot page nodes in the page.trail list. All we need is a template to display the information. This is shown in Example 11-50.

Example 11-50. templates/lib/menu/trail

```
<table border="0" cellpadding="0" cellspacing="2">
  <tr valign="middle">
[% FOREACH item IN trail %]
    <td class="info"></td>
    <td>[% PROCESS menu/text
            link = {
              text     = item.name
              url      = item.url
              class    = 'menu'
            };
        %]</td>
[% END %]
  </tr>
</table>
```

Then we can update the site/layout to include it in an appropriate place, as shown in Example 11-51.

Example 11-51. Adding the bread-crumb trail to templates/lib/site/layout

```
.
    .
    .
    <td width="100%">
      [% PROCESS site/header %]
    </td>
  </tr>

  <!-- new section added -->
  <tr>
    <td align="center">
      [% PROCESS site/name %]
```

```
    </td>
    <td>
      [% PROCESS site/navigate %]
    </td>
  </tr>
  <!-- end of new section -->

  <tr>
    <td></td>
    <td>[% PROCESS misc/line %]</td>
    .
    .
    .
```

Two new templates are being added, *site/name* and *site/navigate*. The first adds a nameplate underneath the logo (see Example 11-52).

Example 11-52. templates/lib/site/name

```
<a href="[% site.url.home %]">
[%- INCLUDE misc/image image=site.image.name | trim -%]
</a>
```

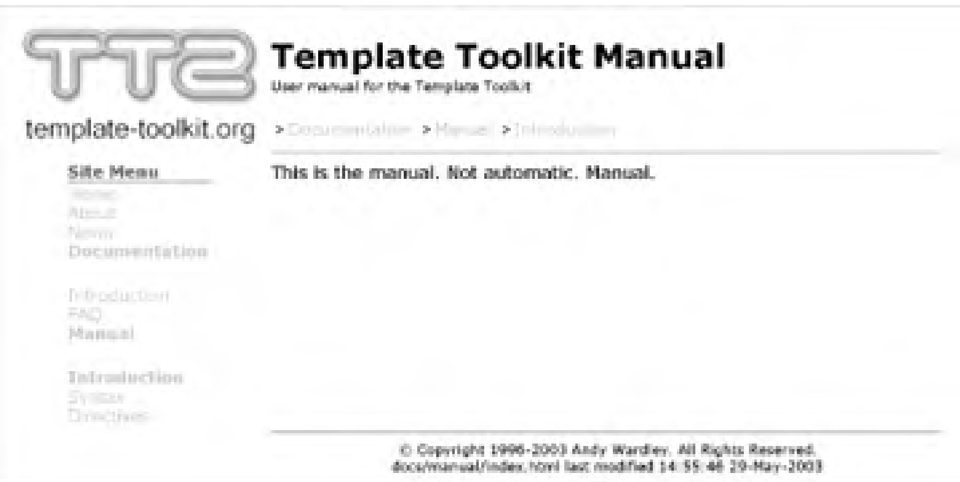
This is purely for aesthetic reasons to help keep the layout balanced when we add the bread-crumb trail. The *site/navigate* component does nothing more than display the bread-crumb trail (see Example 11-53). However, we will be adding more to this template shortly.

Example 11-53. templates/lib/site/navigate

```
[% PROCESS menu/trail trail=page.trail %]
```

Now you can run bin/build -a to rebuild the entire site and see the pages with the bread-crumb trail added. Figure 11-6 shows a screenshot of a page containing the new bread-crumb trail.

Figure 11-6. Bread-crumb trail



11.4.6 Previous and Next Pages

We can also add a navigation component to add links to the previous and next pages relative to the current one. These were also determined by the *config/expand* template and set in the page.prev and page.next variables. Either of these values could be undefined, so we need to be sure to cover those cases. Example 11-54 shows the *menu/prevnext* template component that generates these links.

Example 11-54. templates/lib/menu/prevnext

```
<table border="0" cellpadding="2" cellspacing="2">
  <tr valign="middle">
  [% IF page.prev -%]
    <td align="right">
      [% PROCESS menu/text
           link = {
             text  = page.prev.name
             url   = page.prev.url
             class = 'menu'
           };
     -%]
    </td>

    [% IF page.next -%]
    <td>|</td>
    [% END -%]
  [% END %]

  [% IF page.next %]
    <td align="left">
      [%- PROCESS menu/text
            link = {
              text  = page.next.name
              url   = page.next.url
              class = 'menu'
            };
      %]
    </td>
  [% END %]
  </tr>
</table>
```

Once again, *menu/text* is being used to generate the individual text links. This template is mostly just providing the layout logic.

The *site/navigate* template can now be modified to incorporate the new navigation component, as shown in Example

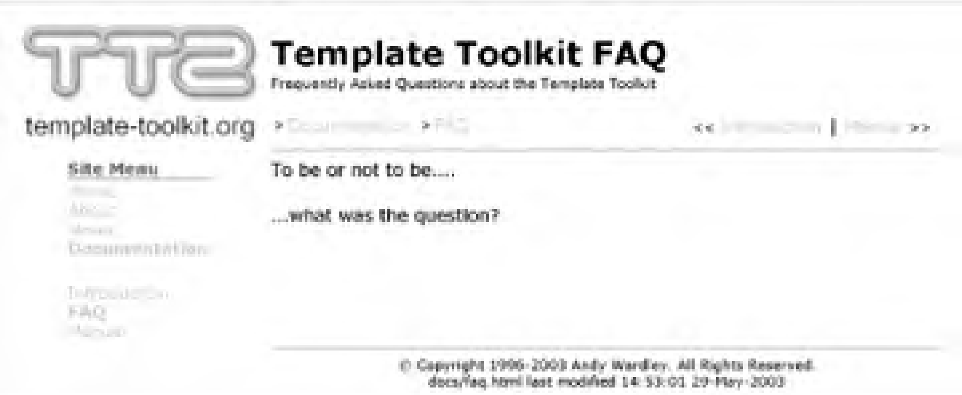
11-55.

Example 11-55. Adding menu/prevnext to templates/lib/site/navigate

```
<table width="100%" border="0" cellpadding="0" cellspacing="0">
  <tr valign="middle">
    <td align="left">
      [% PROCESS menu/trail trail=page.trail %]
    </td>
    <td align="right">
      [% PROCESS menu/prevnext %]
    </td>
  </tr>
</table>
```

Figure 11-7 shows a page with the bread-crumb trail on the left, with links to the previous and next pages on the right of the page header.

Figure 11-7. Previous and next pages



11.5 Structuring Page Content

We've looked at different ways that template components can be used to generate shared user interface components such as headers, menus, and footers. Now we are going to turn our attention to the page content itself, showing how the Template Toolkit can be used to help structure and present content in different ways.

11.5.1 Defining Sections

Web pages containing any more than a few paragraphs will typically be organized into sections, subsections, or some other kind of logical division. A simple HTML page may use nothing more than <h1> and <h2> elements to break up a document into small chunks. A more complex page might add all manner of fancy HTML markup to indicate section breaks or other structural parts of a document. You might also want to include a table of contents at the top of the page, linking to sections of the document below.

Needless to say, all this involves extra work that requires a lot of repetition. We want to make it easy to add and update site content, and don't want to burden page authors with the task of adding presentation markup, generating and maintaining tables of contents, and so on. Furthermore, we want to keep the presentation aspects separate so that we can restyle the site at a later date without having to rewrite all the content.

The solution is of course to use templates to define the presentation elements, which are then automatically applied to the page content. We will also show how a table of contents can be automatically generated from the structure of the content.

11.5.1.1 Section headers

Adding a standard block of HTML markup at the start of each section in a page is as easy as calling a template component. Example 11-56 shows a page that uses the INCLUDE directive to add a section header in two places.

Example 11-56. Adding section headers

```
[% META title = 'About the Template Toolkit'
        about = 'A brief overview of and introduction
                 to the Template Toolkit'
%]

[% INCLUDE section/header
     title = 'Overview'
%]
<p>
  The <b>Template Toolkit</b> is a fast,
  powerful, and easily extensible template
  processing system written in <b>Perl</b>...
</p>

[% INCLUDE section/header
     title = 'Mailing Lists'
%]
<p>
  A number of mailing lists are provided for discussing
```

```
  the Template Toolkit...

</p>
```

A simple template for generating each section header is shown in Example 11-57. Here we are using the *misc/line* template component to add a line across the page, followed by the section title in a <h1> element.

Example 11-57. templates/lib/section/header

```
[% PROCESS misc/line %]



<h1>[% title %]</h1>
```

You might also want to define macros to make using these components as easy as possible. These can be defined at the top of the page or, better still, in a preprocessed configuration template. For example:

```
[% MACRO Section(title) INCLUDE section/header %]
```

With this macro defined, the page content can be simplified, as shown in Example 11-58.

Example 11-58. Using a section macro

```
[% META title = 'About the Template Toolkit'

        about = 'A brief overview of and introduction

                 to the Template Toolkit'

%]



[% Section('Overview') %]

<p>

  The <b>Template Toolkit</b> is a fast,

  powerful, and easily extensible template

  processing system written in <b>Perl</b>...

</p>



[% Section('Mailing Lists') %]

<p>

  A number of mailing lists are provided for discussing

  the Template Toolkit...

</p>
```

11.5.1.2 Section wrappers

If you want to add some markup at the start of a section and some more at the end, you could use separate *section/header* and *section/footer* templates. But as we know from looking at page headers and footers, a better approach is to create a single wrapper template.

Let's say that we want to add the title at the start of the section, but move the line generated by *misc/line* to come *after* the content for the section. Example 11-59 shows a wrapper template to do this.

Example 11-59. templates/lib/section/wrapper

```
<h1>[% title %]</h1>

[% content %]

[% PROCESS misc/line %]
```

To use this component, the page template should use the WRAPPER directive, enclosing the content for each section between WRAPPER and END. This can be seen in Example 11-60.

Example 11-60. Using a section wrapper

```
[% META title = 'About the Template Toolkit'
        about = 'A brief overview of and introduction
                 to the Template Toolkit'
%]

[% WRAPPER section/wrapper
     title = 'Overview'
%]
<p>
  The <b>Template Toolkit</b> is a fast,
  powerful, and easily extensible template
  processing system written in <b>Perl</b>...
</p>
[% END %]

[% WRAPPER section/wrapper
     title = 'Mailing Lists'
%]
<p>
  A number of mailing lists are provided for discussing
  the Template Toolkit...
</p>
[% END %]
```

11.5.1.3 Sections and subsections

You can create as many different components as you require for sections, subsections, subsubsections, and any other page elements. Example 11-61 shows a page with a more complex structure, including subsections nested within sections.

Example 11-61. Sections and subsections

```
[% META title = 'About the Template Toolkit'
        about = 'A brief overview of and introduction
                 to the Template Toolkit'
```

```
%]
[% MACRO Section(title)    INCLUDE page/section;
   MACRO Subsection(title) INCLUDE page/subsection
%]

[% Section('Overview') %]
<p>
  The <b>Template Toolkit</b> is a fast,
  powerful, and easily extensible template
  processing system written in <b>Perl</b>...
</p>

[% Subsection('Features' ) %]
<ul>
  <li>Fast, powerful, and extensible...</li>
  <li>Powerful presentation language...</li>
  <li>And so on...</li>
</ul>

[% Section('Mailing Lists') %]
<p>
  A number of mailing lists are provided for discussing
  the Template Toolkit.
</p>

[% Subsection('templates') %]
<p>
  The <b>templates</b> mailing list exists
  for reporting information, asking questions, and
  discussing development or any other topic
  relevant to the Template Toolkit.
</p>

[% Subsection('templates-announce') %]
<p>
  The <b>templates-announce</b> mailing list
  is a low-volume list used for announcing
```

```
  new versions of the Template Toolkit
  or other related information.
</p>
```

Example 11-62 shows the *page/section* template and Example 11-63 shows the *page/subsection* template.

Example 11-62. templates/lib/page/section

```
<table width="100%" border="0" cellpadding="0" cellspacing="4">
  <tr>
    <td align="left">
      <a name="[% id %]">
        <h2 class="section">[% title %]</h2>
      </a>
    </td>
    <td align="right">
      [% UNLESS no_top -%]
      <a href="#top" class="navlink">Top</a>
      [% END -%]
    </td>
  </tr>
</table>
```

Example 11-63. templates/lib/page/subsection

```
<a name="[% id %]">
<h3 class="subsection">[% title %]</h3>
</a>
```

The template in *page/section* is a little more involved than the simpler *page/subsection* template. Both templates generate an HTML anchor around the title using an optional id variable as the identifier. We'll be looking at this in the next section when we build a table of contents to link down to the different sections and subsections in a document.

11.5.2 A Table of Contents

We now have the page content defined in terms of sections and subsections. From this, we can generate a table of contents to help the reader navigate around the document structure.

11.5.2.1 Anchor points

We saw in the previous section how the *page/section* and *page/subsection* templates in Examples Example 11-62 and Example 11-63, respectively, generate an HTML <a> element to create an anchor point in the document. To use this feature, a value must be provided for the id variable:

```
[% INCLUDE page/subsection
     title = 'Testing 123'
        id = 'testing'
%]
```

This generates the following HTML:

```
<a name="testing">

<h3 class="subsection">Testing 123</h3>

</a>
```

This subsection can now be linked to by appending #testing to the end of the page URL—e.g., *http://localhost/ttbook/about.html#testing*.

11.5.2.2 Better page macros

The first task is to enhance the Section and Subsection macros. We'll define these in a separate *config/macros* template, shown in Example 11-64.

Example 11-64. templates/lib/config/macros

```
[%  page.items = [ ];


    MACRO Section(title) BLOCK;

        id   = title.replace('\W+', '_');

        item = {

            url   = "#$id"

            name  = title

            items = [ ]

        };

        CALL page.items.push(item);

        PROCESS page/section;

    END;


    MACRO Subsection(title) BLOCK;

        id   = title.replace('\W+', '_');

        item = {

            url  = "#$id"

            name = title

        };

        CALL page.items.last.items.push(item);

        PROCESS page/subsection;

    END;

-%]
```

The first line creates a reference to an empty list and assigns it to page.items. This will be used to keep track of each section in the page.

```
page.items = [ ];
```

The Section expects a title argument, as before. The body of the macro is defined as a BLOCK continuing down to the corresponding END directive.

```
MACRO Section(title) BLOCK;

   # macro body

END;
```

The title is used to generate an HTML-compliant identifier for the section by replacing all sequences of one or more nonword characters with a single underscore:

```
id = title.replace('\W+', '_');
```

The item variable is then defined as a hash array containing values for url and name. It also defines an items list for storing information about any subsections contained within this section.

```
item = {

   url   = "#$id"

   name  = title

   items = [ ]

};
```

The new item is added to the page.items list:

```
CALL page.items.push(item);
```

Finally, the *page/section* template is processed to generate the appropriate HTML markup for the section heading:

```
PROCESS page/section;
```

The Subsection macro differs in a few minor details. To keep things simple for this example, we are not providing any support for nesting subsubsections within subsections, although it would be easy to add. As a result, there is no need for an items list in the item hash.

```
item = {

   url  = "#$id"

   name = title

};
```

Instead of being pushed onto the page.items list, the new item is added to the items list for the current section—that is, the last item on the page.items list:

```
CALL page.items.last.items.push(item);
```

Of course it uses the *page/subsection* template rather than the *page/section* template to generate the subsection header.

To make these MACRO definitions visible, we need to update the *config/main* template to add *config/macros* to the list of templates in the PROCESS directive. Example 11-65 shows the relevant change.

Example 11-65. Addition to config/main

```
[%  PROCESS config/page

         + config/site

         + config/url

         + config/col

         + config/images

         + config/map

         + config/macros   # add this line

-%]
```

11.5.2.3 Generating the table of contents

These macros build up information about the structure of the page content and store it in the page.sections list. Generating a table of contents is then a simple matter of iterating through this data and presenting it nicely as a set of formatted links.

Given that this data isn't complete until the page is processed in its entirety, you may be wondering how we can generate a table of contents to be inserted at the top of the page. The answer is that we use a WRAPPER around the page, as shown in Example 11-66. For the sake of clarity, we removed the page content to show only the directives in question.

Example 11-66. Page layout wrapper

```
[% META title = 'About the Template Toolkit'

       about = 'A brief overview of and introduction

              to the Template Toolkit'

%]


[% WRAPPER page/tocpage %]


[% Section('Overview') %]

  ...


[% Subsection('Features' ) %]

  ...


[% Section('Mailing Lists') %]

  ...


[% Subsection('templates') %]

  ...


[% Subsection('templates-announce') %]

  ...


[% END %]
```

The page content is enclosed in a WRAPPER ... END block. The content is processed first, thereby triggering the Section and Subsection macros, and is then passed off to the *page/tocpage* template for presentation (see Example 11-67).

Example 11-67. templates/lib/page/tocpage

```
<h2>Contents</h2>


<ul>

[% FOREACH section IN page.items -%]

<li><a href="[% section.url %]">[% section.name %]</a></li>

[% PROCESS subs IF section.items.size -%]
```

```
[% END -%]

</ul>


[% BLOCK subs -%]

<ul>

[% FOREACH sub IN section.items -%]

<li><a href="[% sub.url %]">[% sub.name %]</a></li>

[% END -%]

</ul>

[% END %]


[% content %]
```

The first section generates the main table of contents using a FOREACH loop to iterate through each section in the page.items list:

```
<ul>

[% FOREACH section IN page.items -%]

<li><a href="[% section.url %]">[% section.name %]</a></li>

[% PROCESS subs IF section.items.size -%]

[% END -%]

</ul>
```

If a section contains subsections, the subs block is called to create a nested menu. This works in an identical way to the main body, but iterates over the items in section.items rather than page.items.

```
[% BLOCK subs -%]

<ul>

[% FOREACH sub IN section.items -%]

<li><a href="[% sub.url %]">[% sub.name %]</a></li>

[% END -%]

</ul>

[% END %]
```

The page content then follows after the table of contents:

```
[% content %]
```

11.5.2.4 Reusing menu components

You may have noticed that page.items data defined by the Section and Subsection macros has the same basic structure as for our site menu. Each item has a name, a url, and a list of nested items. This choice was deliberate. It allows us to reuse our menu template components to generate the table of contents.

Example 11-68 shows a different version of the *page/tocpage* template from what we saw in Example 11-67.

Example 11-68. Table of contents generated using menu/nest

```
[% FOREACH section IN page.items;

    SET section.subs = 1
```

```
        IF section.items.size;
   END
-%]

<h2>Contents</h2>

[% INCLUDE menu/nest menu=page %]

[% content %]
```

There is one modification we need to make to the data. The *menu/nest* template is programmed to descend into nested items if the subs value is set. The first block of the template in Example 11-68 uses a FOREACH directive to iterate through each item, setting the subs value to 1 if it contains any items:

```
[% FOREACH section IN page.items;
       SET section.subs = 1
        IF section.items.size;
   END
-%]
```

This ensures that the *menu/nest* template will display the entire table of contents, including nested subsections. The *menu/nest* template is called, passing page as the local value for the menu variable. It will then walk through the entries in the page.items list, and also through any nested items within them.

```
<h2>Contents</h2>

[% INCLUDE menu/nest menu=page %]
```

As before, we display the page content after the table of contents. Figure 11-8 shows an HTML page built this way.

Figure 11-8. HTML page with table of contents

11.5.2.5 Adding the table of contents automatically

To make life as easy as possible, we can modify the *site/wrapper* template to automatically wrap the page in the *page/tocpage* template. So that we have some control over which pages this is applied to, we will add a new page type, tocpage. Example 11-69 includes a new CASE for this page type that adds page/tocpage to the list of wrappers for the page.

Example 11-69. Adding a tocpage page type to site/wrapper

```
[%  SWITCH page.type;

      CASE 'text';

         content;


      CASE 'html';

         content WRAPPER site/html

                  + site/layout;


      CASE 'tocpage';

         content WRAPPER site/html

                  + site/layout

                  + page/tocpage;


      CASE;

         THROW page.type "Invalid page type: $page.type";

   END;

-%]
```

With this in place, there is no need for a page to explicitly wrap itself in the *page/tocpage* template. Instead, it should define a type of tocpage in a META directive and leave it to *site/wrapper* to add the table of contents:

```
[% META type  = 'tocpage'

      title = 'About the Template Toolkit'

      about = 'A brief overview of and introduction

            to the Template Toolkit'

%]


[% Section('Overview') %]

   ...etc...
```

11.5.3 Declarative Markup Using XML

The Template Toolkit allows you to decouple your core content from any particular presentation style. However, the techniques that we've shown in this section are very much specific to the Template Toolkit and to a particular way of generating pages.

That isn't going to be a problem in many cases, but you might prefer to define your content in a format that can be read and manipulated by other tools as well as by the Template Toolkit. XML is of course the perfect example of an open format that you might like to use.

XML allows you to write declarative markup instead of the more procedural markup of the Template Toolkit. Rather than embedding a set of instructions in the document that say "add a section header here" or "generate a table of contents over there," XML simply states things for the record. It says "this is a section" or "this is a subsection," and allows you to do what you like with the information.

The Template Toolkit is quite happy working with XML. It will do the hard work of transforming it into HTML, using template components to apply the current presentation style for your site along the way.

11.5.3.1 XML page content

Example 11-70 shows a page template that uses XML to define the core content.

Example 11-70. XML page template

```
[% META type  = 'xml'
       title = 'About the Template Toolkit'
       about = 'A brief overview of and introduction
                to the Template Toolkit'
%]

<page>
  <section title="Overview">
    <p>
      The <b>Template Toolkit</b> is a fast,
      powerful, and easily extensible template
      processing system written in <b>Perl</b>...
    </p>

    <subsection title="Features">
      <ul>
        <li>Fast, powerful, and so on...</li>
      </ul>
    </subsection>
  </section>

  <section title="Mailing Lists">
    <p>
      A number of mailing lists are provided for discussing
      the Template Toolkit.
    </p>

    <subsection title="templates">
      <p>
        The <b>templates</b> mailing list...
```

```
    </p>

   </subsection>


   <subsection title="templates-announce">

    <p>

     The <b>templates-announce</b> mailing list...

    </p>

   </subsection>

  </section>

</page>
```

The page content is enclosed within a <page> element. Sections and subsections are declared using the appropriate <section> and <subsection> elements. We can include any kind of valid XHTML markup within these elements.

11.5.3.2 XML page wrapper

A minor change is required to our presentation framework for it to handle XML files. We've declared the page type for Example 11-70 to be xml in the META tag. We must therefore add the appropriate handler to the *site/wrapper* template, as shown in Example 11-71.

Example 11-71. Adding an XML page type to site/wrapper

```
[%  SWITCH page.type;

     CASE 'text';

        content;


     CASE 'html';

        content WRAPPER site/html

                      + site/layout;


     CASE 'tocpage';

        content WRAPPER site/html

                      + site/layout

                      + page/tocpage;


     CASE 'xml';

        content WRAPPER site/html

                      + site/layout

                      + site/xmlpage;


     CASE;

        THROW page.type "Invalid page type: $page.type";

   END;
```

```
-%]
```

The *site/xmlpage* template is used as an additional wrapper to process XML page content. Example 11-72 shows how it works.

Example 11-72. templates/lib/site/xmlpage

```
[%  USE xmldoc  = XML.XPath( text = content );
    USE xmlview = view(
          prefix   = 'xmlpage/'
          notfound = 'xmltag'
    );

    FOREACH xnode = xmldoc.findnodes('/page');
        xmlview.print(xnode);
    END;
-%]
```

It uses the XML.XPath plugin, passing the XML content of the page as the text variable. The plugin then returns an object through which we can query the XML document, assigned to the xmldoc variable:

```
USE xmldoc  = XML.XPath( text = content );
```

It then creates a VIEW plugin object called xmlview. This will be used to map XML elements to corresponding templates in the *xmlpage/* subdirectory of *templates/lib*. The xmltag template will be used to render any XML elements for which no template is defined:

```
USE xmlview = view(
      prefix   = 'xmlpage/'
      notfound = 'xmltag'
);
```

The final section iterates through each page node,[6] calling on the xmlview view to print it using the appropriate template:

[6] There's only one in this case, but findnodes returns a list anyway.

```
FOREACH xnode = xmldoc.findnodes('/page');
    xmlview.print(xnode);
END;
```

11.5.3.3 XML view templates

The view first calls the *xmltag/page* template to process the outermost page XML node. It calls the item.content method passing the current view as an argument. This generates the view-specific content for the page that can then be wrapped using the existing *page/tocpage* template to add a table of contents (see Example 11-73).

Example 11-73. templates/lib/xmlpage/page

```
[% item.content(view)
     WRAPPER page/tocpage
-%]
```

The call to item.content(view) causes the view to iterate over the content of the page XML node. In this case, it will find section nodes, which are sent off to the *xmlpage/section* for processing (see Example 11-74).

Example 11-74. templates/lib/xmlpage/section

```
[%  Section( item.getAttribute('title') );

    item.content(view)

-%]
```

This template calls the Section macro, fetching the value for the title from the XML title attribute. The section content is then displayed, again by calling the item.content method.

The *xmlpage/subsection* template is called whenever a subsection XML element is encountered. It is almost identical to *xmlpage/section*, as shown in Example 11-75.

Example 11-75. templates/lib/xmlpage/subsection

```
[%  Subsection( item.getAttribute('title') );

    item.content(view)

-%]
```

Whenever the view finds an XML element that it doesn't have a template for, it calls on xmlpage/xmltag, which regenerates the original XML element. This allows us to pass XHTML content through without it requiring any further transformation (see Example 11-76).

Example 11-76. templates/lib/xmlpage/xmltag

```
[% item.starttag;

   item.content(view);

   item.endtag

-%]
```

We also need a simple template to reproduce any plain-text parts as they are (see Example 11-77).

Example 11-77. templates/lib/xmlpage/text

```
[% item -%]
```

That's all there is to it. Any time you want to define some specific handling for an XML element, simply add the appropriately named template to the *templates/lib/xmlpage* directory. The view will take care of the rest.

These simple templates don't do much in themselves. They just provide the glue between XML.XPath nodes and our existing Section and Subsection macros. We get to reuse all of our existing presentation framework, but can now define content in XML, HTML, and various other formats, all of which can be freely intermixed with Template Toolkit directives.

11.6 Creating a New Skin

In the final section of this chapter, we are going to show how a new set of template components can be created to rebrand, or *skin*, the site. Rather than modify our existing components, we will create a new set in a different directory. For these examples, the directory will be *templates/skin/droplet*, relative to the current project directory of */home/dent/web/ttbook*. We can create as many different skins as required as long as each has its own unique name and corresponding component directory. The name we are using for this skin is droplet, for no reason in particular.

11.6.1 Creating a Skin

First, we must create a directory for the skin-specific templates:

```
$ cd /home/dent/web/ttbook

$ mkdir templates/skin

$ mkdir templates/skin/droplet
```

The INCLUDE_PATH configuration option and the corresponding lib option for *ttree* allow multiple directories to be specified for the location of template files. The *templates/skin/droplet* directory should be added to *etc/ttree.cfg* as a new lib option coming before the existing one. Example 11-78 shows the new line added to the first block of the *etc/ttree.cfg* file.

Example 11-78. Adding a lib option to etc/ttree.cfg

```
src  = /home/dent/web/ttbook/templates/src

# add lib option for new skin

lib  = /home/dent/web/ttbook/templates/skin/droplet

lib  = /home/dent/web/ttbook/templates/lib

dest = /home/dent/web/ttbook/html
```

You may also want to update the corresponding skeleton template, *skeleton/etc/ttree.cfg*. Or you can update the skeleton file and then run the *bin/configure* script to have it regenerate *etc/tree.cfg*.

We will need to define some configuration data for the new skin, so we create a *config/skin* template and add it to the list in config/main (see Example 11-79).

Example 11-79. Adding config/skin to config/main

```
[%  PROCESS config/page

        + config/site

        + config/url

        + config/col

        + config/images

        + config/map

        + config/macros

        + config/skin     # add this line

-%]
```

The *config/skin* configuration template for the droplet skin is shown in Example 11-80. It defines a URL, some colors, and other information relating to a set of icons that will be used by various template components.

Example 11-80. templates/skin/droplet/config/skin

```
[%  site.url.icon = "$site.url.images/icon"
    site.col.icon = {
        on   = 'orange'
        off  = 'blue'
        roll = 'red'
        dead = 'gray'
    }
    site.image.icon = {
        large = {
            url    = "$site.url.icon/large"
            src    = "$site.url.icon/large/blue/dot.png"
            alt    = 'dot icon'
            width  = 36
            height = 36
        }
        small = {
            url    = "$site.url.icon/small"
            src    = "$site.url.icon/small/blue/dot.png"
            alt    = 'dot icon'
            width  = 24
            height = 24
        }
        tiny = {
            url    = "$site.url.icon/tiny"
            src    = "$site.url.icon/tiny/blue/dot.png"
            alt    = 'dot icon'
            width  = 18
            height = 18
        }
    }
-%]
```

In case we later decide to generate the site without this skin, we must also provide a dummy *config/skin* template in the default *templates/lib* directory (see Example 11-81).

Example 11-81. templates/lib/config/skin

```
[%# hook for skins to perform any
  # additional extra configuration
-%]
```

11.6.2 Custom Navigation Components

Now we can add our own custom components to the *templates/skin/droplet* directory. They will be used in preference to those in the default *templates/lib* directory.

We can start by defining a new *misc/line* component, as shown in Example 11-82.

Example 11-82. templates/skin/droplet/misc/line

```
<table border="0" width="100%" cellpadding="0" cellspacing="0">

  <tr>

    <td height="1" bgcolor="[% site.col.line %]"><img

        width="1" height="1" /></td>

  </tr>

</table>
```

The design of this skin is based around some simple icons. Example 11-83 shows a template component to generate the HTML for the various icons we are using.

Example 11-83. templates/skin/droplet/misc/icon

```
[%  # misc/icon - generate image tag for icon



    DEFAULT

      size = 'small'

      icon = 'dot'

      col  = 'blue';



    IF (image = site.image.icon.$size);

        PROCESS misc/image

          image.src  = "$image.url/$col/${icon}.png"

          image.alt  = "$icon icon";

    ELSE;

        THROW logo "invalid icon size: $size";

    END;

-%]
```

11.6.2.1 Nested menu

The *misc/icon* template can be used to spice up the *menu/nest* template that we introduced in Example 11-68. The new version can be seen in Example 11-84.

Example 11-84. templates/skin/droplet/menu/nest

```
[%  DEFAULT

        global.linkno = 0

        icon = site.image.icon.tiny;
```

```
    colroll = site.col.icon.roll;

    WRAPPER menu/table;
      FOREACH item = menu.items;
        linkno  = (global.linkno = global.linkno + 1);
        colicon = item.hot ? site.col.icon.on
                           : site.col.icon.off;

        INCLUDE menu/link
          link = {
            name     = "menu_$linkno"
            text     = item.name
            url      = item.url
            icon     = "$icon.url/$colicon/right.png"
            rollover = "$icon.url/$colroll/right.png"
            size     = icon.width
            class    = item.hot ? 'menuselect' : 'menu'
          };

        INCLUDE menu/submenu menu=item
          IF item.subs;
      END;
    END;
-%]
```

Figure 11-9 shows a screenshot of a page containing the droplet-style nested menu.

Figure 11-9. Droplet-style nested menu

11.6.2.2 Menu elements

Various HTML table elements and other components are used to generate the menu in this style. They have been moved into the templates shown in Examples 11-85 through 11-93 to promote modularity and to help keep the *menu/nest* template clutter-free.

Example 11-85. templates/skin/droplet/menu/table

```
<table border="0" cellpadding="0" cellspacing="2">

[%- content -%]

</table>
```

Example 11-86. templates/skin/droplet/menu/row

```
<tr valign="middle">

[%- content -%]

</tr>
```

Example 11-87. templates/skin/droplet/menu/blank

```
<tr>

  <td></td>

  <td> </td>

</tr>
```

Example 11-88. templates/skin/droplet/menu/line

```
<tr>

  <td colspan="2">[%- PROCESS misc/line -%]</td>

</tr>
```

Example 11-89. templates/skin/droplet/menu/name

```
[% PROCESS menu/blank -%]

<tr>

  <td colspan="2" class="menutitle">[% menu.name %]</td>

</tr>

[% PROCESS menu/line -%]
```

Example 11-90. templates/skin/droplet/menu/link

```
<tr valign="middle">
  <td align="middle" width="[% item.size %]" height="[% item.size %]">
[%- PROCESS menu/icon -%]</td>
  <td align="left">
[%- PROCESS menu/text -%]</td>
</tr>
```

Example 11-91. templates/skin/droplet/menu/submenu

```
<tr>
    <td></td>
    <td>
      [% PROCESS menu/nest %]
    </td>
  </tr>
```

Example 11-92. templates/skin/droplet/menu/icon

```
<a href="[% link.url %]"
[% IF link.target -%]
   target="[% link.target %]"
[% END -%]
[% IF link.rollover -%]
   onmouseover="[% link.name %].src = '[% link.rollover %]';"
   onmouseout="[% link.name %].src = '[% link.icon %]';"
[% END -%]
><img
   name="[% link.name %]" src="[% link.icon %]"
   width="[% link.size %]" height="[% link.size %]" border="0" /></a>
```

Example 11-93. templates/skin/droplet/menu/text

```
<a href="[% link.url %]"
[% IF link.class -%]
   class="[% link.class %]"
[% END -%]
[% IF link.target -%]
   target="[% link.target %]"
[% END -%]
[% IF link.rollover -%]
   onmouseover="[% link.name %].src = '[% link.rollover %]';"
```

```
  onmouseout="[% link.name %].src = '[% link.icon %]';"
[% END -%]
>
[%- link.text -%]
</a>
```

11.6.2.3 Stacked menu

We can also create a new version of the stacked menu by reusing these menu components, as shown in Example 11-94.

Example 11-94. templates/skin/droplet/menu/stack

```
[%  DEFAULT
        global.linkno = 0
        icon = site.image.icon.tiny;

    pending = [ menu ];
    colroll = site.col.icon.roll;

    WRAPPER menu/table;
        WHILE pending.size;
            menu = pending.shift;

            FOREACH item = menu.items;
                linkno  = (global.linkno = global.linkno + 1);
                colicon = item.hot ? site.col.icon.on
                                   : site.col.icon.off;

                INCLUDE menu/link
                    link = {
                        name     = "item_$linkno"
                        text     = item.name
                        url      = item.url
                        icon     = "$icon.url/$colicon/right.png"
                        rollover = "$icon.url/$colroll/right.png"
                        size     = icon.width
                        class    = item.hot ? 'menuselect' : 'menu'
                    };

                pending.push(item)
```

```
            IF item.subs;
        END;

        PROCESS menu/name menu=pending.first
          IF pending.size;
      END;

    END;
-%]
```

Figure 11-10 shows the end result.

Figure 11-10. Nested menu

11.6.2.4 Other page components

To complete the set, we can also define new templates for the bread-crumb trail, the next and previous page menu, and the page sections and subsections (see Examples 11-95 through 11-98).

Example 11-95. templates/skin/droplet/menu/trail

```
[%  DEFAULT
      icon  = site.image.icon.tiny;
      page.linkno = 0;

    colicon = site.col.icon.off;
    colroll = site.col.icon.roll;

    WRAPPER menu/table
```

```
         + menu/row;

       FOREACH item IN trail;
          INCLUDE menu/trail/crumb
             link = {
                name     = "trail_$loop.count"
                text     = item.name
                url      = item.url
                icon     = "$icon.url/$colicon/right.png"
                rollover = "$icon.url/$colroll/right.png"
                size     = icon.width
                class    = 'menu'
             };
       END;
    END;
-%]

[%- BLOCK menu/trail/crumb -%]
  <td align="middle" width="[% item.size %]" height="[% item.size %]">
[%- PROCESS menu/icon -%]</td>
  <td align="left">
[%- PROCESS menu/text -%]</td>
[%- END -%]
```

Example 11-96. templates/skin/droplet/menu/prevnext

```
[%  size  = 'tiny'
    icon  = site.image.icon.$size
    width = icon.width;

    colicon = site.col.icon.off;
    colroll = site.col.icon.roll;

    WRAPPER menu/table
         + menu/row;
%]

[% # is there a previous page?
   IF page.prev;
      link = {
```

```
            name     = "prev"
            text     = page.prev.name
            url      = page.prev.url
            icon     = "$icon.url/$colicon/left.png"
            rollover = "$icon.url/$colroll/left.png"
            size     = icon.width
            class    = 'menu'
        };
-%]
    <td align="right">
      [%- PROCESS menu/text -%]
    </td>
    <td width="[% width %]">
      [%- PROCESS menu/icon -%]
    </td>
[% ELSE %]
    <td></td>
    <td width="[% width %]">
      [%- INCLUDE misc/icon
            size = 'tiny'
            col  = site.col.icon.dead
            icon = 'left'
      %]
    </td>
[% END %]

    <td width="[% width %]">
      [%- INCLUDE misc/icon
            col  = site.col.icon.on
            icon = 'dot'
            size = 'tiny'
      -%]
    </td>

[% # is there a next page?
   IF page.next;
        link = {
            name     = "next"
            text     = page.next.name
```

```
          url      = page.next.url
          icon     = "$icon.url/blue/right.png"
          rollover = "$icon.url/red/right.png"
          size     = icon.width
          class    = 'menu'
      };
-%]
    <td width="[% width %]">
      [%- PROCESS menu/icon -%]
    </td>
    <td align="left">
      [%- PROCESS menu/text  -%]
    </td>
[% ELSE %]
    <td width="[% width %]">
      [%- INCLUDE misc/icon
           col  = site.col.icon.dead
           icon = 'right'
           size = 'tiny'
      %]
    </td>
    <td></td>
[% END %]

[% END   # WRAPPER %]
```

Example 11-97. templates/skin/droplet/page/section

```
[%  size = 'small';
    imgsize = site.image.icon.$size;
-%]
<p>
<table width="100%" border="0" cellpadding="0" cellspacing="4">
  <tr valign="middle">
    <td width="[% imgsize.width %]">
      [%- PROCESS misc/icon %]</td>
    <td align="left" width="100%">
      <a name="[% id %]"><b class="section">[% title %]</b></a>
```

```
    </td>
    <td align="right">
     [%- UNLESS no_top %]
     <a href="#top">[%
         INCLUDE misc/icon
           size = 'small'
           icon = 'up'
           col  = site.col.icon.off
      %]</a>
      [% END %]
    </td>
  </tr>
  <tr>
    <td></td>
    <td colspan="2">
     [% PROCESS misc/line %]
    </td>
  </tr>
  <tr valign="top">
    <td></td>
    <td colspan="2">
     [% content %]
    </td>
  </tr>
</table>
</p>
```

Example 11-98. templates/skin/droplet/page/subsection

```
[%  size = 'tiny';
    imgsize = site.image.icon.$size;
-%]
<p>
<table width="100%" border="0" cellpadding="0" cellspacing="4">
  <tr valign="middle">
    <td width="[% imgsize.width %]">
     [%- PROCESS misc/icon %]</td>
    <td align="left">
     <a name="[% id %]"><b class="subsection">[% title %]</b></a>
```

```
    </td>
  </tr>
  <tr valign="top">
    <td></td>
    <td>
      [% content %]
    </td>
  </tr>
</table>
```

Figure 11-11 is what it looks like when it is all put together. Remember that none of the core content has changed, only the template components that handle the presentation.

Figure 11-11. Complete page in the droplet style

Chapter 12. Dynamic Web Content and Web Applications

In Chapter 2 and Chapter 11, we looked at some basic, and then some more advanced techniques for generating static web content. The fundamental limitation of static web pages is, rather obviously, that they are static. The Template Toolkit allows you to incorporate any kind of dynamic data into a template as it is being processed. But once the page has been generated, the data is fixed. If you want to use different data, you must process the template again.

Most web content is *static*. The page is generated offline from a template, using a page design tool, or perhaps just typed in at a text editor. It is then uploaded to the web server where it is delivered time and time again without changing. Simple, fast, and efficient.

Some web content is *dynamic*. The results from a search engine are a perfect example of a dynamically generated page. There's no way of generating the page in advance because you don't know what search terms the user is going to enter. There are many other examples of dynamically generated web content to be found at news sites, in bulletin boards and chat rooms, and of course in e-commerce applications, where pages showing the latest offers or the contents of a user's shopping cart must be generated dynamically to incorporate the latest live data.

In this chapter, we will look at generating dynamic web pages using the Template Toolkit. We will start with some simple CGI scripts to illustrate the basic principles, and then move up to Apache and mod_perl. We'll be working toward a complete (but minimal) web application, concentrating particularly on achieving a clear separation of concerns between different functional aspects of the system: presentation, application, and storage.

12.1 CGI Scripts

The Common Gateway Interface (CGI) provides a simple mechanism for generating dynamic web content and running web applications. The web server receives a request and maps it to a CGI program, which is then run. These are often located in a special *cgi-bin* directory or have a particular file extension such as *.cgi*. Various parameters relating to the CGI request are passed to the program as environment variables. Additional data may be piped in through the program's standard input in the case of a POST request. The program does whatever it needs to do in the way of application processing, and then prints a simple header and then the content of the page to standard output. The web server sends this back to the client's browser as the response.

Perl is a very popular language for writing CGI scripts. The CGI module provides a wealth of functionality for CGI programming. For a full tour of CGI programming and the CGI module, see *CGI Programming with Perl* by Scott Guelich, Shishir Gundavaram, and Gunther Birznieks (O'Reilly).

12.1.1 Simple CGI Script

Using the Template Toolkit in a CGI script is easy. The Template `process( )` method prints its output to STDOUT by default. For simple cases, very little work is required on our part to turn any Perl program using the Template Toolkit into a CGI script. Example 12-1 shows such a script.

Example 12-1. ttcgi1.pl

```
#!/usr/bin/perl


use strict;

use warnings;

use Template;


$| = 1;

print "Content-type: text/html\n\n";


my $tt    = Template->new( );

my $input = 'destruction1.html';

my $vars  = {

    planet  => 'Earth',

    captain => 'Prostetnic Vogon Jeltz',

    time    => 'two of your earth minutes',

};


$tt->process($input, $vars)

    || die $tt->error( );
```

The only lines that are specific to CGI programming are these:

```
$| = 1;

print "Content-type: text/html\n\n";
```

The first of these lines disables buffering on standard output. This ensures that any content printed is sent back to the client right away. The second line prints a standard CGI header, telling the browser that we're sending it an HTML page. The other difference between this example and the simple text version that we first saw in Chapter 1 is that our template must now be marked up as valid HTML, as shown in Example 12-2.

Example 12-2. destruction1.html

```
<html>
 <head>
  <title>Destruction of [% planet %] is Imminent!</title>
 </head>
 <body>
  <p>
   People of [% planet %], your attention please.
  </p>
  <p>
   This is [% captain %] of the
   Galactic Hyperspace Planning Council.
  </p>
  <p>
   As you will no doubt be aware, the plans
   for development of the outlying regions
   of the Galaxy require the building of a
   hyperspatial express route through your
   star system, and regrettably your planet
   is one of those scheduled for destruction.
  </p>
  <p>
   The process will take slightly less than
   [% time %].
  </p>
 </body>
</html>
```

12.1.1.1 Using standard templates

The Template Toolkit provides a set of standard templates for adding HTML headers and footers to pages. On Unix systems, they are typically installed in */usr/local/tt2/templates*. On Windows platforms, they are installed in *C:\Program Files\Template Toolkit 2\templates*. The *Template::Config* module provides the instdir( ) method to determine the location in a portable way. By adding this directory to the INCLUDE_PATH configuration option, we can then use the standard *html/page* template as a WRAPPER for the page, as shown in Example 12-3.

Example 12-3. ttcgi2.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;
use Template::Config;

$| = 1;
print "Content-type: text/html\n\n";

my $tdir  = Template::Config->instdir('templates');
my $tt    = Template->new({
    INCLUDE_PATH => [ '.', $tdir ],
    WRAPPER      => 'html/page'
});
my $input  = 'destruction2.html';
my $vars   = {
    planet  => 'Earth',
    captain => 'Prostetnic Vogon Jeltz',
    time    => 'two of your earth minutes',
    html    => {
        head => {
            title => "Destruction of Earth is Imminent!",
        },
    },
};

$tt->process($input, $vars)
    || die $tt->error( );
```

The location of the *templates* directory is determined by the following line and stored in the $tdir variable:

```
my $tdir  = Template::Config->instdir('templates');
```

The $tdir directory is then added to the INCLUDE_PATH, along with the current working directory (.):

```
my $tt    = Template->new({
    INCLUDE_PATH => [ '.', $tdir ],
    WRAPPER      => 'html/page'
});
```

The *html/page* wrapper template adds the <html>, <head>, and <body> elements around the generated page content. It inserts the value of the html.head.title variable in the <title> of the <head> element, to set the page title. Accordingly, we define an appropriate title in the $vars hash:

```
my $vars   = {
    planet  => 'Earth',
    captain => 'Prostetnic Vogon Jeltz',
    time    => 'two of your earth minutes',
    html    => {
        head => {
            title => "Destruction of Earth is Imminent!",
        },
    },
};
```

The *destruction2.html* template can now be made much simpler, as shown in Example 12-4. The HTML headers and footers are all added automatically, leaving us to concentrate on the content. We're also using the html_para filter to add the <p> and </p> tags around each paragraph.

Example 12-4. destruction2.html

```
[% FILTER html_para %]
People of [% planet %], your attention please.


This is [% captain %] of the
Galactic Hyperspace Planning Council.


As you will no doubt be aware, the plans
for development of the outlying regions
of the Galaxy require the building of a
hyperspatial express route through your
star system, and regrettably your planet
is one of those scheduled for destruction.


The process will take slightly less than
[% time %].
[% END %]
```

If you've been working through the examples in Chapter 11, you'll probably have developed your own wrappers and other user interface templates that you can use in place of *html/page*.

12.1.2 Using the DATA Section

You can also define the main page template in a DATA section following the main part of the CGI script, as shown in Example 12-5.

Example 12-5. ttcgi3.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;

$| = 1;
print "Content-type: text/html\n\n";

my $tt   = Template->new({
    INCLUDE_PATH => '/home/dent/vogon/templates',
    WRAPPER      => 'vogon/page'
});
my $vars   = {
    planet  => 'Earth',
    captain => 'Prostetnic Vogon Jeltz',
    time    => 'two of your earth minutes',
};

$tt->process(\*DATA, $vars)
    || die $tt->error( );

_ _DATA_ _
[% FILTER html_para %]
People of [% planet %], your attention please.

This is [% captain %] of the
Galactic Hyperspace Planning Council.

As you will no doubt be aware, the plans
for development of the outlying regions
of the Galaxy require the building of a
hyperspatial express route through your
star system, and regrettably your planet
is one of those scheduled for destruction.
```

```
The process will take slightly less than

[% time %].

[% END %]
```

The _ _DATA_ _ (or _ _END_ _) marker indicates the point where the script stops and the template starts. Perl provides the DATA filehandle to read the text from this block. We pass a reference to the filehandle as the first argument to the process( ) method and leave it to do the rest:

```
$tt->process(\*DATA, $vars)

    || die $tt->error( );
```

The approach is great for small and simple CGI scripts. It allows you to keep everything together and contained in one file. You can see both the Perl code and the main page template in the same place, but they are still kept nicely separate from each other. Other components or layout templates such as *html/page* or the hypothetical *vogon/page* wrapper used in this example can be kept out of the way in separate files so that they don't obstruct the core content and can be reused between different CGI scripts.

Be warned that you can't use the DATA section if you want to run your CGI scripts under Apache::Registry. Apache:Registry allows you to run unaltered CGI scripts under mod_perl for a significant speedup. Instead of being loaded and compiled each time a request is made, the script is kept in compiled form in the memory space of the web server. It can then be executed quickly and repeatedly on demand.

However, a CGI script gets only one chance to read the DATA section. When it has been read once, there is no going back to read it again. If you plan to use Apache::Registry, you should use separate page template files rather than embedding them in a DATA section.

12.1.3 Using the CGI Module

The CGI module does everything you'll ever need to in CGI programming and a whole lot more. Example 12-6 shows how we create a CGI object and pass it to the template as the cgi variable.

Example 12-6. ttcgi4.pl

```
#!/usr/bin/perl


use strict;

use warnings;

use Template;

use CGI;


$| = 1;


my $cgi   = CGI->new( );

my $tt    = Template->new( );

my $input = 'cgiparams.html';

my $vars  = {

    cgi   => $cgi,
```

```
};

print $cgi->header;

$tt->process($input, $vars)
    || die $tt->error( );
```

The template processed by the script, *cgiparams.html*, is shown in Example 12-7. It calls the param( ) method of the CGI object first to fetch a list of request parameters, and then again to fetch the value for each parameter within the FOREACH loop.

Example 12-7. cgiparams.html

```
<h1>CGI Parameters</h1>

<ul>
[% FOREACH p = cgi.param -%]
   <li><b>[% p %]</b> [% cgi.param(p) %]</li>
[% END -%]
</ul>
```

Example 12-8 shows some typical output generated by the CGI script. In this case, the request URL used was */cgi-bin/ttcgi4.pl?pi=3.14&e=2.718&message=Hello%20World*. We didn't add any HTML page wrapper in this example to keep things simple. But that would of course be required for any CGI script operating in the real world.

Example 12-8. Output of cgiparams.html

```
<h1>CGI Parameters</h1>

<ul>
   <li><b>pi</b> 3.14</li>
   <li><b>e</b> 2.718</li>
   <li><b>message</b> Hello World</li>
</ul>
```

If you want to use the CGI object to manipulate headers, cookies, or anything else outside of generating content, you'll probably need to do it in the calling CGI script.

12.1.3.1 Setting cookies

Let's look at an example of how cookies can be set using values supplied from within a template. We start by defining a cookies template variable in the CGI script as a reference to an initially empty list. This will be used to store any cookies that should be added to the CGI header.

```
my @cookies;
my $vars  = {
    cgi     => $cgi,
    cookies => \@cookies,
};
```

The CGI object provides the cookie method for creating cookies. We call this from within the template to create a cookie object.

```
[% cookie = cgi.cookie(
     name    = 'SessionID',
     value   = 12345678,
     expires = '+1m'
   )
%]
```

The newly created cookie is then pushed onto the cookies list:

```
[% cookies.push(cookie) %]
```

Back in the CGI script, we need to process the template first and then check to see whether any cookies have been added to the list. Cookies must be added to the response header before any content is sent back to the client. Rather than let the Template process( ) method print its output directly to standard output, we provide it with a reference to an $output variable. This is used to store the generated HTML page until we have set the cookie headers and are ready to send a response back to the client.

```
my $output;


$tt->process($input, $vars, \$output)
    || die $tt->error( );
```

Then we check for any cookies and provide them as an option to the CGI header( ) method before printing the page content stored in $output:

```
if (@cookies) {
    @cookies = ("-cookie", [ @cookies ]);
}
print $cgi->header(@cookies), $output;
```

The complete CGI script is shown in Example 12-9.

Example 12-9. ttcgi5.pl

```
#!/usr/bin/perl


use strict;
use warnings;
use Template;
use CGI;


$| = 1;


my $cgi   = CGI->new( );
my $tt    = Template->new( );
my $input = 'cgicookie.html';
my @cookies;
my $vars  = {
    cgi     => $cgi,
```

```
    cookies => \@cookies,
};
my $output;

$tt->process($input, $vars, \$output)
    || die $tt->error( );

if (@cookies) {
    @cookies = ('-cookie', [ @cookies ]);
}

print $cgi->header(@cookies), $output;
```

The *cgicookie.html* template is listed in Example 12-10.

Example 12-10. cgicookie.html

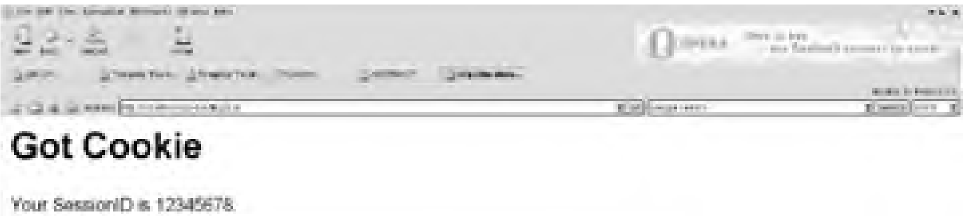
```
[% IF (cookie = cgi.cookie('SessionID')) %]

  <h1>Got Cookie</h1>

  <p>
    Your SessionID is [% cookie %].
  </p>

[% ELSE %]
  [% cookie = cgi.cookie(
        name    = 'SessionID',
        value   = 12345678,
        expires = '+1m'
     );
     cookies.push(cookie)
  %]

  <h1>Set Cookie</h1>

  <p>
    Cookie has been set.  Please reload page.
  </p>

[% END %]
```

Figure 12-1 shows the cookie being set the first time we access the page. We've enabled a feature on our browser that displays the details of each cookie being set so that we can confirm that the CGI script is working as expected.

Figure 12-1. cookieset.png

When the page is reloaded, the cookie is read and the value for SessionID printed, as shown in Figure 12-2.

Figure 12-2. cookieget.png



12.1.4 CGI Script Web Application

Now we're going to look at an example of a more complete CGI script that provides a simple web interface to a database containing entries for a fictional travel guide. Each entry has a name (e.g., Earth) as well as a unique numerical identifier (e.g., 42). We would like to be able to display an entry from the guide by specifying either the name or id. We would also like to be able to search the database to help find entries of interest. We'll be using MySQL in this example, but the techniques apply to any relational database.

12.1.4.1 CGI script

Let's start by walking through the CGI script to explain what each section of code does.

12.1.4.1.1 Preparation

The CGI script starts with the usual preamble. We first load the various modules that we are going to use:

```
#!/usr/bin/perl


use strict;

use warnings;


use DBI;
```

```
use CGI;
use CGI::Carp qw(fatalsToBrowser);
use Template;
$| = 1;
```

Then we define some configuration data:

```
my $ROOTDIR = '/home/dent/web/guide';
my $ROOTURL = '/~dent/guide';
my $ROOTCGI = '/cgi-bin/dent/guide.pl';
my $DBDSN   = 'DBI:mysql:guide';
my $DBUSER  = 'dent';
my $DBPASS  = 'ruhtra';
```

More preparation follows as we create a CGI object, make a connection to the database, and declare some variables, including the $vars hash containing template variables. The $template variable is used to store the name of the template that is processed to generate the page content. We'll be setting it shortly.

```
my $cgi = CGI->new( );
my $dbh = DBI->connect($DBDSN, $DBUSER, $DBPASS)
    || die "failed to connect to database: $DBI::errstr";


my ($param, $template);
my $vars = {
    rootdir => $ROOTDIR,
    rooturl => $ROOTURL,
    rootcgi => $ROOTCGI,
};
```

12.1.4.1.2 Application

Now we can get down to the application processing phase. The flow of control is determined by one of the request parameters being provided—name, id, or search. The if ... elsif ... else construct selects the right block of code accordingly.

```
if ($param = $cgi->param('name')) {
    # ...
}
elsif ($param = $cgi->param('id')) {
    # ...
}
elsif ($param = $cgi->param('search')) {
    # ...
}
else {
    # ...
}
```

If a name parameter is provided, the appropriate SELECT query is sent to the database. The entry is returned as a reference to a hash array, hopefully without error,[1] and is added to the $vars hash as the entry template variable. The $template variable is then set to entry.html.

[1] Note the use of the CGI::Carp module. This will catch our calls to die and generate an HTML page for sending back to the browser.

```
if ($param = $cgi->param('name')) {

    my $entry = $dbh->selectrow_hashref(

        "SELECT id, name, author, about, date

         FROM entry WHERE name=?", { }, $param)

        || die $DBI::errstr;

    $vars->{ entry } = $entry;

    $template = 'entry.html';

}
```

The handling of the id parameter is much the same as it is for name:

```
elsif ($param = $cgi->param('id')) {

    my $entry = $dbh->selectrow_hashref(

        "SELECT id, name, author, about, date

         FROM entry WHERE id=?", { }, $param)

        || die $DBI::errstr;

    $vars->{ entry } = $entry;

    $template = 'entry.html';

}
```

The search parameter requires a slightly different process to allow for the multiple entries that can be returned. Here the entries template variable is set to contain the list of entries returned, each of which is a hash reference, and the $template is set to entries.html:

```
elsif ($param = $cgi->param('search')) {

    $vars->{ search } = $param;

    $param =~ s/*/\%/g;  # change '*' to '%'

    my $sth = $dbh->prepare(

        'SELECT id, name, author, about, date

         FROM entry WHERE name LIKE ?' )

        || die $DBI::errstr;

    $sth->execute($param) || die $sth->errstr( );

    $vars->{ entries } = $sth->fetchall_arrayref({  });

    $template = 'entries.html';

}
```

This application allows the user to specify wildcards in a pattern using the * character—e.g., ear*. MySQL, on the other hand, uses % to denote wildcards. To cater for this, the appropriate transformation is made to the search term in $param before it is used in the query. A copy of the original search term is saved as the search template variable.

```
$vars->{ search } = $param;

$param =~ s/*/\%/g;  # change '*' to '%'
```

If none of the name, id, or search parameters is provided, the index page is displayed:

```
else {
    $template = 'index.html';
}
```

12.1.4.1.3 Presentation

At this point, the $template variable tells us which template needs to be processed, and $vars contains any variables required to process it. We create a Template object specifying various options indicating the location of templates, and naming a template for preprocessing (*config*) and another for wrapping around the page content (*wrapper*).

```
my $tt = Template->new({
    INCLUDE_PATH => [
        "$ROOTDIR/templates/cgi",
        "$ROOTDIR/templates/lib",
    ],
    PRE_PROCESS => 'config',
    WRAPPER     => 'wrapper',
});
```

Then we print the CGI header and process the template to generate the dynamic HTML page content:

```
print $cgi->header( );


$tt->process($template, $vars)
    || die $tt->error( );
```

All done! The complete CGI script is shown in Example 12-11.

Example 12-11. guide/cgi-bin/guide.pl

```
#!/usr/bin/perl


use strict;
use warnings;


use DBI;
use CGI;
use CGI::Carp qw(fatalsToBrowser);
use Template;
$| = 1;


#--------------------------------------------------
# configuration
#--------------------------------------------------


my $ROOTDIR = '/home/dent/web/guide';
```

```
my $ROOTURL = '/~dent/guide';
my $ROOTCGI = '/cgi-bin/dent/guide.pl';
my $DBDSN   = 'DBI:mysql:guide';
my $DBUSER  = 'dent';
my $DBPASS  = 'ruhtra';

my $cgi = CGI->new( );
my $dbh = DBI->connect($DBDSN, $DBUSER, $DBPASS)
    || die "failed to connect to database: $DBI::errstr";

my ($param, $template);
my $vars = {
    rootdir => $ROOTDIR,
    rooturl => $ROOTURL,
    rootcgi => $ROOTCGI,
};

#--------------------------------------------------
# application
#--------------------------------------------------

if ($param = $cgi->param('name')) {
    my $entry = $dbh->selectrow_hashref(
        "SELECT id, name, author, about, date
         FROM entry WHERE name=?", { }, $param)
        || die $DBI::errstr;
    $vars->{ entry } = $entry;
    $template = 'entry.html';
}
elsif ($param = $cgi->param('id')) {
    my $entry = $dbh->selectrow_hashref(
        "SELECT id, name, author, about, date
         FROM entry WHERE id=?", { }, $param)
        || die $DBI::errstr;
    $vars->{ entry } = $entry;
    $template = 'entry.html';
}
elsif ($param = $cgi->param('search')) {
    $vars->{ search } = $param;
```

```
    $param =~ s/*/\%/g;  # change '*' to '%'
    my $sth = $dbh->prepare(
        'SELECT id, name, author, about, date
         FROM entry WHERE name LIKE ?' )
        || die $DBI::errstr;
    $sth->execute($param) || die $sth->errstr( );
    $vars->{ entries } = $sth->fetchall_arrayref({  });
    $template = 'entries.html';
}
else {
    $template = 'index.html';
}

#------------------------------------------------------------------------
# presentation
#------------------------------------------------------------------------

my $tt  = Template->new({
    INCLUDE_PATH => [
        "$ROOTDIR/templates/cgi",
        "$ROOTDIR/templates/lib",
    ],
    PRE_PROCESS => 'config',
    WRAPPER     => 'wrapper',
});

print $cgi->header( );

$tt->process($template, $vars)
    || die $tt->error( );
```

12.1.4.2 Template components

The preprocessed *config* template, shown in Example 12-12, loads the Date plugin, defines a date MACRO that uses it, and then defines site and page data. See Chapter 11 for a full discussion on writing and using configuration templates.

Example 12-12. guide/templates/lib/config

```
[%  USE Date;

    MACRO date(d) BLOCK;
      # entry dates contain both date and
      # time, but we just want the date
      items = d.split('-');
      Date.format(
        "0:00:00 $items.2/$items.1/$items.0"
        format = '%d-%B-%Y'
      );
    END;

    site = {
      title     = "TT Hitch Hiker's Guide"
      admin     = 'webmaster@template-toolkit.org'
      copyright = '2003 Andy Wardley'
    }

    site.url = {
      guide  = rootcgi
      index  = "$rooturl/index"
      images = "$rooturl/images"
      css    = "$rooturl/css/tt2.css"
    }

    site.col = {
        back  = '#FFFFFF'  # white
        text  = '#000000'  # black
        line  = '#00AAF0'  # sky blue
    }

    site.logo = {
      src    = "$site.url.images/logo/tt2_120x40.gif"
      alt    = "TT2 Logo"
      width  = 120
      height = 40
```

```
    }

    page = {
      name  = template.name
      file  = template.name
      title = template.title
      about = template.about
      type  = template.type or 'html'
      date  = template.date or Date.format(template.modtime)
    }
-%]
```

Example 12-13 shows the *wrapper* template, which applies the *html* and *layout* templates as further wrappers around the generated page content. The use of wrapper templates is also discussed in Chapter 11.

Example 12-13. guide/templates/lib/wrapper

```
[%  SWITCH page.type;
        CASE 'text';
            content;

        CASE 'html';
            content WRAPPER html
                          + layout;
        CASE;
            THROW page.type "Invalid page type: $page.type";
    END;
-%]
```

The *html* and *layout* templates are shown in Examples Example 12-14 and Example 12-15, respectively.

Example 12-14. guide/templates/lib/html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
 <head>
  <title>
    [% site.title %]
    [% ": $page.title" IF page.title %]
  </title>
  <link rel="stylesheet"
        href="[% site.url.css %]" />
```

```
  <meta http-equiv="Content-Type"
        content="text/html; charset=iso-8859-1" />
 </head>

 <body bgcolor="[% site.col.back %]"
       text="[% site.col.text %]">
  [% content %]
 </body>
</html>
```

Example 12-15. guide/templates/lib/layout

```
[% MACRO line BLOCK -%]
  <tr>
    <td colspan="3">
      [% PROCESS line %]
    </td>
  </tr>
[%- END %]

<table width="100%" height="70%" border="0" cellpadding="4" cellspacing="4">
  <tr valign="middle">
    <td width="150" align="center">
      [% PROCESS logo %]
    </td>
    <td align="left">
      [% PROCESS header %]
    </td>
    <td align="right">
      [% PROCESS form %]
    </td>
  </tr>

  [% line %]

  <tr valign="top" height="100%">
    <td colspan="3">
      <!-- page content -->
      [% content %]
      <!-- end of page content -->
```

```
    </td>
  </tr>

  [% line %]

  <tr valign="bottom">
    <td colspan="3" align="center">
      [% PROCESS footer %]
    </td>
  </tr>
</table>
```

The *header* template uses the values defined in the page data structure to generate a page header, as shown in Example 12-16.

Example 12-16. guide/templates/lib/header

```
<h1 class="title">[% page.title %]</h1>
[% IF page.about -%]
<div class="info">
  [% page.about %]
</div>
[% END -%]
```

The *form* template, shown in Example 12-17, provides the search form. Any current value for the search template variable is displayed in the input field.

Example 12-17. guide/templates/lib/form

```
<form action="[% site.url.guide %]"
      method="POST" enctype="application/x-www-form-urlencoded">
  <table border="0">
    <tr valign="middle">
      <td>
        <input type="submit"
               name="submit"
               value=" Search " />
      </td>
      <td>
        <input type="text"
               name="search"
               size="30"
               value="[% search %]" />
```

```
      </td>
    </tr>
    <tr valign="middle">
      <td></td>
      <td class="info">
        e.g., <i>earth</i>, <i>magrethea</i>, <i>ear*</i>, <i>*th*</i>
      </td>
    </tr>
  </table>
</form>
```

The *logo* and *footer* templates, shown in Examples Example 12-18 and Example 12-19, respectively, also do what their names suggest.

Example 12-18. guide/templates/lib/logo

```
[% image = site.logo -%]
<a href="[% site.url.index %]"><img
   src="[% image.src %]" alt="[% image.alt %]"
   width="[% image.width %]" height="[% image.height %]"
   border="0" /></a>
```

Example 12-19. guide/templates/lib/footer

```
<p class="info">
  &copy; Copyright [% site.copyright %].
  All Rights Reserved.
 <br />
  [% page.name %] last modified [% page.date %]
</p>
```

12.1.4.3 Page templates

The *entry.html* page template is used to display a single entry. The template source is shown in Example 12-20. It sets the appropriate page values from the entry returned from the database. This allows the *header* template to display appropriate values when it is automatically added by the wrapper templates. In this simple example, the only real page content comes from the about.entry field.

Example 12-20. guide/templates/cgi/entry.html

```
[% # set various page items
   page.title = entry.name;
   page.name  = "Entry for $entry.name";
   page.date  = date(entry.date);
```

```
  page.about = "by $entry.author on $page.date"
%]

[% entry.about %]
```

Figure 12-3 shows a screenshot of an HTML page generated from this template.

Figure 12-3. earth.png

The *entries.html* page template, shown in Example 12-21, displays a list of the entries returned by a search.

Example 12-21. guide/templates/cgi/entries.html

```
[% page.title = 'Search Results' %]

[% n = entries.size or 'no' %]

<h3>There [% n =  = 1 ? 'is' : 'are' %] [% n %]
[% n =  = 1 ? 'entry' : 'entries' %] matching your search.</h3>

[% IF entries.size %]
  <ul>
  [%- FOREACH entry IN entries -%]
    <li><a href="[% site.url.guide %]?id=[% entry.id %]">[% entry.name %]</a>
      <br />
      <span class="info">by [% entry.author %] on [% date(entry.date) %].</span>
    </li>
  [%- END -%]
  </ul>
[% END %]
```

Figure 12-4 shows the results of a search for *th*.

Figure 12-4. search.png


Search Results
Search
There are 2 entries matching your search.

12.2 CGI Templates

Often dynamic content is generated as a response to a web query. The user types something into a form and a CGI program runs to extract the parameters, search the database, and generate the response. The easiest way to do this is to have the CGI program generate the HTML response. In this section we show a more maintainable way: use the CGI plugin from within a template to access query parameters.

12.2.1 Using the CGI Plugin

The Template Toolkit provides the CGI plugin as a simple wrapper around the CGI module. If you don't have any particular need to use a CGI object in the calling Perl program—say, to read request parameters or set headers—don't create one. Instead, use the CGI plugin to create a CGI object from within any templates that require access to it. Example 12-22 shows a template identical to that in Example 12-7, with the exception of the first line, which loads the CGI plugin.

Example 12-22. cgiplugin.html

```
[% USE cgi %]

<h1>CGI Parameters</h1>

<ul>
[% FOREACH p = cgi.param -%]
  <li><b>[% p %]</b> [% cgi.param(p) %]</li>
[% END -%]
</ul>
```

The plugin name can be specified in upper- or lowercase. The CGI object will be assigned to the variable of the same name in matching case. In Example 12-22, the lowercase cgi variable is used in keeping with Example 12-7. We could just as easily use the uppercase CGI name when the plugin is loaded:

```
[% USE CGI %]
```

and then again whenever the plugin is used:

```
[% FOREACH p = CGI.param %]
```

You cannot instantiate more than one CGI per request. If you create a CGI request in the controlling Perl script, you should pass it as a variable to the template instead of using a plugin. The plugin will create a second CGI object with unpredictable results.

12.2.2 Web Programming in Templates

The Template Toolkit gives you access to plugins and allows you to call out to subroutines and other objects from template code. This means that you can do a large amount of web programming entirely within the templates.

This is the style of popular web programming languages such as PHP and Microsoft's ASP. It is how HTML::Mason works, albeit by embedding Perl code rather than using a custom web programming language. It is a useful technique, particularly for small applications where you want to keep things together in one place, and don't want the overhead of a complex application framework that will only distract you from the task at hand.

The problem with this approach is that it often doesn't scale well to larger applications. HTML::Mason is perhaps the exception here, being very much based around a component architecture that naturally promotes modularity and scalability. However, it suffers from the same problem as PHP and ASP in binding the application code too tightly to presentation aspects, making it hard to change one without affecting the other.

The Template Toolkit approaches the problem from a different angle. Whereas PHP, ASP, and HTML::Mason are designed primarily for web programming, the Template Toolkit is more focused on web presentation. It deals mostly with making the generated pages look pretty (which may involve all manner of complex presentation logic) but doesn't worry itself too much about application programming issues. That is best left to a real programming language, namely Perl.

But as we have said, the technique is useful for smaller applications, and with a little careful organization can scale reasonably well. The Template Toolkit isn't fanatical about enforcing strict disciplines on anyone, and provides what you need to get the job done quickly, if that's what you want.

12.2.2.1 Dispatching CGI script

To illustrate this, we will take the Perl CGI script from Example 12-11 and implement the body of it in a template, making use of the CGI and DBI plugins. We still require a Perl CGI script to dispatch the template, as shown in Example 12-23.

Example 12-23. guide/cgi-bin/ttguide.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;

$| = 1;

my $ROOTDIR = '/home/dent/guide';
my $ROOTURL = '/~dent/guide';
my $ROOTCGI = '/cgi-bin/dent/ttguide.pl';
my $DBDSN   = 'DBI:mysql:guide';
my $DBUSER  = 'dent';
my $DBPASS  = 'ruhtra';

my $input = 'guide.html';
my $vars  = {
    rootdir => $ROOTDIR,
    rooturl => $ROOTURL,
    rootcgi => $ROOTCGI,
    dbdsn   => $DBDSN,
    dbuser  => $DBUSER,
    dbpass  => $DBPASS,
};

my $tt  = Template->new({
    INCLUDE_PATH => [
        "$ROOTDIR/templates/cgi",
```

```
        "$ROOTDIR/templates/lib",

    ],

    PRE_PROCESS => 'config',

    WRAPPER     => 'wrapper',

});


print "Content-type: text/html\n\n";


$tt->process($input, $vars)

    || die $tt->error( );
```

The script does little more than define some variables and create a Template object to process the *guide.html* file, located in the *templates/cgi* directory, relative to the $ROOTDIR, which in this example is /home/dent/guide.

12.2.2.2 Main control template

The application processing has now been moved into the *guide.html* template, shown in Example 12-24.

Example 12-24. guide/templates/cgi/guide.html

```
[%  USE cgi;

    USE dbi( dbdsn, dbuser, dbpass );


    # main control loop

    IF (param = cgi.param('name'));

        PROCESS entry/name;

    ELSIF (param = cgi.param('id'));

        PROCESS entry/id;

    ELSIF (param = cgi.param('search'));

        PROCESS entry/search;

    ELSE;

        PROCESS index.html;

    END;

%]
```

It first loads the CGI plugin, then the DBI plugin, passing the relevant configuration parameters for it to make a database connection. For both plugins, the lowercase names are used:

```
USE cgi;

USE dbi( dbdsn, dbuser, dbpass );
```

Then the control block follows. The request parameters are inspected and one of the relevant templates, *entry/name*, *entry/id*, or *<entry/search>*, is processed. If none of the parameters is provided, the *index.html* template is used.

```
IF (param = cgi.param('name'));

    PROCESS entry/name;

ELSIF (param = cgi.param('id'));
```

```
    PROCESS entry/id;
ELSIF (param = cgi.param('search'));
    PROCESS entry/search;
ELSE;
    PROCESS index.html;
END;
```

12.2.2.3 Additional control templates

The *entry/name* template, shown in Example 12-25, dispatches a database request to fetch an entry by name.

Example 12-25. guide/templates/cgi/entry/name

```
[%  entries = dbi.query(
        "SELECT id, name, author, about, date
         FROM entry WHERE name='$param'"
    );

    # entries is an iterator, so get first item
    entry = entries.get;

    IF entry;
        PROCESS entry.html;
    ELSE;
        PROCESS notfound.html;
    END;
%]
```

The query method of the DBI plugin returns a reference to an iterator object, which is assigned to entries. We're expecting only one item to be returned from this query, so we call the get method to fetch the first item from entries:

```
entry = entries.get;
```

If an entry is returned, the *entry.html* template is processed to present it. Otherwise, the *notfound.html* template is used to inform the user that the entry could not be found.

The *entry/id* template is very similar (see Example 12-26).

Example 12-26. guide/templates/cgi/entry/id

```
[%  entries = dbi.query(
        "SELECT id, name, author, about, date
         FROM entry WHERE id=$param"
    );
    entry = entries.get;
    IF entry;
        PROCESS entry.html;
```

```
    ELSE;
        PROCESS notfound.html;
    END;
%]
```

Example 12-27 shows the *entry/search* template.

Example 12-27. guide/templates/cgi/entry/search

```
[%  search = param.replace('*', '%');
    entries = dbi.query(
        "SELECT id, name, author, about, date
         FROM entry WHERE name LIKE '$search'"
    );
    PROCESS entries.html
        entries = entries.get_all;
%]
```

As before, we change any occurrences of * to % so that the user's idea of what constitutes a wildcard expression (e.g., ear*) matches the format that MySQL is expecting (e.g., ear%). This time, however, we do it using the replace virtual method:

```
search = param.replace('*', '%');
```

We are expecting a list of items to be returned from the search. The *entries.html* template generates an appropriate response even if no matches are found and the entries list is empty. We call the get_all method on the entries iterator to return a list of all matches found and then assign it back to entries. This effectively turns the iterator into a regular list so that the *entries.html* template can use the size list virtual method to determine whether there are any entries to display.

12.2.2.4 Perl or template?

We don't normally recommend putting too much application logic in templates as a general rule. But we do recognize that it can be useful from time to time, particularly when you have a small job to get done quickly and would rather have something basic working today than something elegant working next week.

In the example that we have just looked at, we created a CGI Perl script specifically to dispatch a single template. Given that we have gone to the effort of writing a Perl script, it would make more sense on this occasion to encode the application logic in Perl, leaving the templates to handle only presentation issues. This is the approach that we showed you in Example 12-7.

On the other hand, you may be using a generic template dispatcher such as Apache::Template. We saw an example in Chapter 11 where it was configured to process any *.tt2* that it finds before being returned to the client. It means you can simply drop a new *.tt2* file into your web directory to have Apache::Template automatically process it as a dynamically generated web page. There is no need to write a calling CGI script or custom mod_perl handler to cater for it. In cases such as this, the benefit of being able to perform some basic web programming tasks entirely within a template is more apparent.

So even though hardcore web programming in templates isn't usually encouraged, it certainly can be done. Furthermore, it is still possible to maintain a clear separation of concerns by using different templates for different parts of the system. In this example, we used one template for the main control loop and one for handling each query. All the presentation templates were borrowed without change from the previous example.

12.3 Apache and mod_perl

The biggest problem with CGI programming is that it is slow. Each request fires off a CGI script from scratch. Perl must first parse and compile the script and any modules you use (including the Template Toolkit, of course) before it can even start to generate content.

The mod_perl extension to Apache makes these problems go away. Rather than writing Perl CGI scripts, you write Perl handlers that sit "inside" the web server. The handlers and any modules they use are loaded and compiled when the server starts. Once Perl has compiled them into an internal "opcode" tree, they can be executed quickly, efficiently, and repeatedly with minimal overhead.

A second important benefit comes from using the Template Toolkit in a mod_perl-enabled Apache server. It allows you to create one Template object that is reused for all requests. When a template is first used, it is parsed by the Template Toolkit and converted to the equivalent Perl code. This is then passed to Perl, which compiles it into an opcode tree.

The Template Toolkit caches these compiled templates so that you can process them as many times as you like but only have to go through the relatively slow process of compiling them once. However, to get the benefit of this, you must use one Template object that remains persistent from one request to the next. The examples that follow all adopt this technique.

For a complete discussion of mod_perl and related topics, see *Practical mod_perl* by Stas Bekman and Eric Cholet (O'Reilly).

12.3.1 Apache::Template

Way back in Chapter 1, we looked at using Apache::Template to dispatch templates from a mod_perl-enabled Apache server. Example 12-28 shows an Apache/mod_perl configuration that uses Apache::Template to dispatch the web application template from Example 12-24.

Example 12-28. Apache::Template configuration

```
PerlModule        Apache::Template

TT2IncludePath     /home/dent/guide/templates/cgi
TT2IncludePath     /home/dent/guide/templates/lib
TT2PreProcess      config
TT2Process         process

TT2Variable        rooturl /~dent/guide
TT2Variable        rootcgi /ttguide
TT2Variable        dbdsn   DBI:mysql:guide
TT2Variable        dbuser  dent
TT2Variable        dbpass  ruhtra

Alias              /ttguide /home/dent/guide/templates/cgi

<Location /ttguide>
    SetHandler      perl-script
    PerlHandler     Apache::Template
</Location>
```

The Apache::Template module is loaded and then various TT2* parameters are set. At the time of this writing, *Apache::Template* is a version behind the Template Toolkit and doesn't yet support the TT2Wrapper (i.e., WRAPPER) configuration option. For now, we can emulate the behavior of TT2Wrapper with the TT2Process option. We tell Apache::Template to process the *process* template, shown in Example 12-29, in place of each main page template.

Example 12-29. templates/lib/process

```
[% PROCESS $template WRAPPER wrapper -%]
```

The *process* template processes the original page template.[2] The template variable contains a reference to the original page template (or rather, the Template::Document object used to represent it). The original template is processed and the output is wrapped in the *wrapper* template, thereby providing the equivalent functionality to the WRAPPER configuration option.

> [2] The leading $ on $template indicates that it is the template variable we want processed, rather than a template with the literal name "template."

The rooturl, rootcgi, dbdsn, dbuser, and dbpass template variables are set to their appropriate values using the TT2Variable directive. We also define an Apache Alias that maps the */ttguide* URL to the appropriate template files in the */home/dent/guide/templates/cgi* directory.

```
Alias           /ttguide /home/dent/guide/templates/cgi
```

Finally, we indicate that all files in this location and corresponding directory should be processed by Apache::Template:

```
<Location /ttguide>
    SetHandler      perl-script
    PerlHandler     Apache::Template
</Location>
```

The *guide.html* page template can now be accessed via the URL */ttguide/guide.html*. No changes to the template are required.

12.3.2 Custom Apache Handler

The Apache::Template module is good for simple things. If you want to do anything that doesn't count as simple, you will probably need to write your own custom *mod_perl* handler.

Example 12-30 shows an example of a module that defines such a handler.

Example 12-30. lib/TTBook/Apache/Handler.pm

```
package TTBook::Apache::Handler;

use strict;
use warnings;

use Template;
use Apache;
use Apache::Constants qw(OK SERVER_ERROR DECLINED);

our $VERSION = 1.00;
our $TT;

sub handler {
    my $r = shift;
```

```
    my $output;

    my %params = $r->method( ) eq 'POST'
        ? $r->content( )
        : $r->args( );

    my $template = $r->path_info( )
        || 'index.html';

    $template =~ s[^/][  ]g;

    $TT ||= do {
        my $rootdir = $r->dir_config('rootdir')
            || return error($r, "'rootdir' not defined");

        Template->new({
            INCLUDE_PATH => [
                "$rootdir/templates/cgi",
                "$rootdir/templates/lib",
            ],
            PRE_PROCESS => 'config',
            WRAPPER     => 'wrapper',
            ERROR       => 'error.html',
        });
    };

    $r->content_type('text/html');
    $r->send_http_header( );

    $TT->process($template, \%params, $r)
        || return error($r, $TT->error( ));

    return OK;
}

sub error {
```

```
    my $r = shift;
    $r->log_error(@_);
    return SERVER_ERROR;
}

1;
```

The interesting part is the handler method. It is called by mod_perl and passed a reference to an Apache::Request object. Through this, we can fetch the request parameters by calling the content( ) method for POST requests, or the args( ) method for GET (and other) requests:

```
sub handler {
    my $r = shift;
    my $output;

    my %params = $r->method( ) eq 'POST'
        ? $r->content( )
        : $r->args( );
```

In this handler, we are using PATH_INFO to determine which template to process. If the handler is bound to a URL of */tthandler*, for example, calling it with a URL of */tthandler/help/index.html* would result in a value of /help/index.html for PATH_INFO. In this case, we would then process the *help/index.html* template in the *$rootdir/templates/cgi* directory, having removed the leading / from the path:

```
my $template = $r->path_info( )
    || 'index.html';

$template =~ s[^/][ ]g;
```

The next block of code creates a Template object and assigns it to the $TT package variable. If $TT already contains an object, it is reused instead. This ensures that the same Template object is used from one request to the next and thus benefits from the caching of compiled templates.

```
$TT ||= do {
    my $rootdir = $r->dir_config('rootdir')
        || return error($r, "'rootdir' not defined");

    Template->new({
        INCLUDE_PATH => [
            "$rootdir/templates/cgi",
            "$rootdir/templates/lib",
        ],
        PRE_PROCESS => 'config',
        WRAPPER     => 'wrapper',
        ERROR       => 'error.html',
    });
};
```

The root directory, $rootdir, from which the INCLUDE_PATH directories are built, is defined in the Apache configuration file that we will be looking at shortly. To fetch this value, the dir_config( ) method is called against the request object.

The content type is declared and the HTTP headers are sent to the client's browser:

```
$r->content_type('text/html');

$r->send_http_header( );
```

Then the page template, `$template`, is processed, passing the current request parameters as template variables. The request object, `$r`, is passed to the `process( )` method as the third argument. Rather than printing the generated HTML page to standard out, the *process( )* method will pass it to the request object by calling its `print( )` method:

```
$TT->process($template, \%params, $r)

    || return error($r, $TT->error( ));



return OK;

}
```

Example 12-31 shows the relevant directive for an Apache configuration file to use this handler.

Example 12-31. etc/tthandler.conf

```
<perl>

  use lib qw( /home/dent/guide/lib )

</perl>



PerlModule   TTBook::Apache::Handler

PerlSetVar   rootdir /home/dent/guide



<Location /myhandler>

    SetHandler       perl-script

    PerlHandler     TTBook::Apache::Handler

</Location>
```

The `<perl>` ... `</perl>` block allows Perl code to be embedded in the configuration. In this example, we are using it to add the location of our custom handler module to Perl's search path. The module is then loaded with the `PerlModule` directive. The `PerlSetVar` directive is used to set a value for the `rootdir` variable. Finally, a `<Location>` ... `</Location>` block is used to bind the handler to the URL */myhandler*.

12.4 A Complete Web Application

We are now going to build a complete mod_perl- and Template Toolkit-enabled, database-driven web application, based on our earlier examples. Although this is a relatively simple example as web applications go, we will nevertheless concentrate on making a clear separation between the different functional concerns.

Presentation will of course be handled by the Template Toolkit. The application-specific processing will be implemented in one module, using another separate module to manage the storage layer (i.e., the database). A third module will then provide the interface between Apache and the application.

12.4.1 Storage

To best understand how the complete application is built, it is perhaps easiest to start from the inside and work out. Or at the bottom and work up. Well, whatever direction it is, we're going to start with the storage module.

This provides a wrapper around a database to hide as much of the nitty-gritty detail as possible. This allows our different applications to use the same storage module, or for an application to use different storage modules as requirements change. In this example, we're using a MySQL database through the DBI module, but next week we might decide to use XML files instead.

In other words, it provides an abstraction that allows applications to work independently of any particular storage mechanism.

12.4.1.1 TTBook::H2G2::Database

This module begins in the usual way for any Perl module by declaring its package and then loading some external Perl modules:

```
package TTBook::H2G2::Database;



use strict;

use DBI;

use Class::Base;

use base qw( Class::Base );
```

The DBI module is of course required to access the MySQL database. We're also using Class::Base and defining it to be the base class of the TTBook::H2G2::Database module.

The three SQL queries that we will be using are defined in the $SQL package variable. They use ? placeholder characters to indicate positions where parameters to the query will be inserted.

```
our $SQL     = {

    get_entry_id   => 'SELECT id, name, author, about, date

                       FROM entry WHERE id=?',

    get_entry_name => 'SELECT id, name, author, about, date

                       FROM entry WHERE name=?',

    entry_search   => 'SELECT id, name, author, about, date

                       FROM entry WHERE name LIKE ?',

};
```

The Class::Base module defines a default new( ) constructor method. This calls the init( ) method to initialize the object using any configuration parameters passed.

```
sub init {

    my ($self, $config) = @_;
```

```
    @$self{ keys %$config } = values %$config;
    $self->{ sql } = $SQL;
    $self->connect( ) || return;
    return $self;
}
```

The contents of the $config hash array are copied into $self and the sql item is set to reference the $SQL package hash. The connect( ) method is then called to make a connection to the database.

Here is the connect( ) method. Notice how the database handle is cached internally in the object as the dbh item.

```
sub connect {
    my $self = shift;

    return $self->{ dbh } ||= do {
        my $dsn = $self->dsn( )
            || return $self->error("No DSN available");

        DBI->connect($dsn, $self->{ user }, $self->{ pass },
                     { RaiseError => 0, PrintError => 0 })
            || $self->error($DBI::errstr);
    };
}
```

The dsn( ) method returns a connection string (in Data Source Notation, hence DSN) for the connect( ) method. If a dsn is already defined, either by a configuration option or a previous call to dsn( ), it is returned as is. Otherwise, it is generated using some or all of the values for name, host, port, and driver, which should be provided as configuration options to the new( ) constructor.

```
sub dsn {
    my $self = shift;
    return $self->{ dsn } ||= do {
        my ($name, $host, $port) = @$self{ qw( name host port ) };
        $host .= ":$port" if $host && $port;
        $name .= "@$host" if $host;
        join(':', 'DBI', $self->{ driver }, $name);
    };
}
```

The prepare( ) method is used to fetch a named SQL query from the sql hash (e.g., get_entry_name, get_entry_id, etc.) and prepare it for execution. The prepared query is cached in the internal sql_query hash table for subsequent use.

```
sub prepare {
    my $self = shift;
    my $sql  = shift
        || return $self->error("no SQL");
    my $dbh  = $self->{ dbh }
        || return $self->error("DBI not connected");
```

```
    my $query;

    if ($query = $self->{ sql }->{ $sql }) {
        my $cache = $self->{ sql_query } ||= { };

        return $cache->{ $sql } ||= $dbh->prepare($query)
            || $self->error("DBI prepare failed: $DBI::errstr");
    }
    else {
        return $dbh->prepare($sql)
            || $self->error("DBI prepare failed: $DBI::errstr");
    }
}
```

The query( ) method calls prepare( ) to prepare a query, and then executes it:

```
sub query {
    my $self = shift;
    my $sql  = shift
        || return $self->error("no SQL");
    my $dbh  = $self->{ dbh }
        || return $self->error("DBI not connected");
    my $sth = $self->prepare($sql)
        || return;
    $sth->execute(@_)
        || return $self->error($sth->errstr( ));
    return $sth;
}
```

The item( ) method first calls query( ) to execute a query. It then calls fetchrow_hashref( ) on the returned DBI statement handle to fetch the first (or only) record returned.

```
sub item {
    my $self = shift;
    my $sth  = $self->query(@_) || return;
    return $sth->fetchrow_hashref( )
        || $self->error($DBI::errstr || "not found");
}
```

The list( ) method is similar, but calls fetchall_arrayref( ) to return a list of all records returned by the query:

```
sub list {

    my $self = shift;

    my $sth  = $self->query(@_) || return;

    return $sth->fetchall_arrayref({  })

        || $self->error($DBI::errstr || "not found");

}
```

The one other method that is worth mentioning is DESTROY. This calls the disconnect( ) method to ensure that the database connection is closed when the object is destroyed.

```
sub DESTROY {

    my $self = shift;

    $self->disconnect('object destroyed') if $self->{ dbh };

}
```

We haven't shown you disconnect( ) yet, but you can probably guess what it does. It is included in the complete listing of the TTBook::H2G2::Database module that follows in Example 12-32.

Example 12-32. lib/TTBook/H2G2/Database.pm

```
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

 = = == = = = = = = = = = = = = = = = = = = = = = = = = = = = =

  = = =

#

# TTBook::H2G2::Database

#

# DESCRIPTION

#   Backend database module for the H2G2 web application.

#

# AUTHOR

#   Andy Wardley <abw@wardley.org>

#

# COPYRIGHT

#   Copyright (C) 2003 Andy Wardley.  All Rights Reserved.

#

#   This module is free software; you can redistribute it and/or

#   modify it under the same terms as Perl itself.

#

# REVISION

#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

```
=

package TTBook::H2G2::Database;

use strict;
use DBI;
use Class::Base;
use base qw( Class::Base );

our $VERSION = sprintf("%d.%02d", q$Revision: 1.6 $ =~ /(\d+)\.(\d+)/);
our $ERROR   = '';
our $SQL     = {
    get_entry_id   => 'SELECT id, name, author, about, date
                      FROM entry WHERE id=?',
    get_entry_name => 'SELECT id, name, author, about, date
                      FROM entry WHERE name=?',
    entry_search   => 'SELECT id, name, author, about, date
                      FROM entry WHERE name LIKE ?',
};

#------------------------------------------------------------------------
# init(\%config)
#
# Initialization method called by Class::Base new( ) constructor.
#------------------------------------------------------------------------

sub init {
    my ($self, $config) = @_;
    @$self{ keys %$config } = values %$config;
    $self->{ sql } = $SQL;
    $self->connect( ) || return;
    return $self;
}

#------------------------------------------------------------------------
# dsn( )
#
# Generate a DSN string from the database
```

```
# connection parameters.
#------------------------------------------------------------------------

sub dsn {
    my $self = shift;
    return $self->{ dsn } ||= do {
        my ($name, $host, $port) = @$self{ qw( name host port ) };
        $host .= ":$port" if $host && $port;
        $name .= "@$host" if $host;
        join(':', 'DBI', $self->{ driver }, $name);
    };
}

#------------------------------------------------------------------------
# connect( )
#
# Connect to the backend database.
#------------------------------------------------------------------------

sub connect {
    my $self = shift;

    return $self->{ dbh } ||= do {
        my $dsn = $self->dsn( )
            || return $self->error("No DSN available");
        DBI->connect($dsn, $self->{ user }, $self->{ pass },
                     { RaiseError => 0, PrintError => 0 })
            || $self->error($DBI::errstr);
    };
}

#------------------------------------------------------------------------
# disconnect( )
#
# Disconnect the database.
#------------------------------------------------------------------------

sub disconnect {
```

```
    my $self = shift;
    my $msg = shift || '';
    $msg = " ($msg)" if length $msg;

    delete $self->{ sql_query };

    $self->{ dbh }->disconnect( )
        if $self->{ dbh };
    delete $self->{ dbh };

    return 1;
}

#------------------------------------------------------------------------
# prepare($sql)
#
# Prepare a query and store the live statement handle internally for
# subsequent execute( ) calls.
#------------------------------------------------------------------------

sub prepare {
    my $self = shift;
    my $sql  = shift
        || return $self->error("no SQL");
    my $dbh  = $self->{ dbh }
        || return $self->error("DBI not connected");
    my $query;

    if ($query = $self->{ sql }->{ $sql }) {
        my $cache = $self->{ sql_query } ||= { };

        return $cache->{ $sql } ||= $dbh->prepare($query)
            || $self->error("DBI prepare failed: $DBI::errstr");
    }
    else {
        return $dbh->prepare($sql)
            || $self->error("DBI prepare failed: $DBI::errstr");
```

```
    }
}

#------------------------------------------------------------------------
# query($sql, @params)
#
# Prepares and executes an SQL query.
#------------------------------------------------------------------------

sub query {
    my $self = shift;
    my $sql  = shift
        || return $self->error("no SQL");
    my $dbh  = $self->{ dbh }
        || return $self->error("DBI not connected");
    my $sth = $self->prepare($sql)
        || return;
    $sth->execute(@_)
        || return $self->error($sth->errstr( ));
    return $sth;
}

#------------------------------------------------------------------------
# item($sql, @args)
#
# Executes the $sql query, passing @args and calls fetchrow_hashref( ) on
# the returned statement handle to fetch a single row as a hash.
#------------------------------------------------------------------------

sub item {
    my $self = shift;
    my $sth  = $self->query(@_) || return;
    return $sth->fetchrow_hashref( )
        || $self->error($DBI::errstr || "not found");
```

```
}

#------------------------------------------------------------------------
# list($sql, @args)
#
# Executes the $sql query, passing @args and calls fetchall_arrayref( ) on
# the returned statement handle to fetch all rows as a list of hashes.
#------------------------------------------------------------------------

sub list {
    my $self = shift;
    my $sth  = $self->query(@_) || return;
    return $sth->fetchall_arrayref({  })
        || $self->error($DBI::errstr || "not found");
}

#------------------------------------------------------------------------
# insert_id( )
#
# Returns the identity of the record most recently inserted into the
# database.
#------------------------------------------------------------------------

sub insert_id {
    my $self = shift;
    return $self->{ dbh }->{ mysql_insertid };
}

#------------------------------------------------------------------------
# quote($value [, $data_type ])
#
# Returns a quoted string (correct for the connected database) from the
# value passed in.
#------------------------------------------------------------------------

sub quote {
    my $self = shift;
    my $dbh  = $self->{ dbh } || return $self->error("DBI not connected");
```

```
    return $dbh->quote(@_);
}

#------------------------------------------------------------------------
# dbh( )
#
# Internal method that retrieves the database handle belonging to the
# instance or attempts to create a new one using connect.
#------------------------------------------------------------------------

sub dbh {
    my $self = shift;
    return $self->{ dbh } || $self->connect( );
}

#------------------------------------------------------------------------
# DESTROY( )
#
# Destructor method called automatically when the object goes out of
# scope.  Disconnects any active database.
#------------------------------------------------------------------------

sub DESTROY {
    my $self = shift;
    $self->disconnect('object destroyed') if $self->{ dbh };
}

1;
```

12.4.2 Configuration

The database storage module expects to be provided with various configuration options to define the parameters for connecting to the database. Rather than littering this information around in several different places (something that makes it hard to find and change), we will create a single configuration module, as shown in Example 12-33.

Example 12-33. lib/TTBook/H2G2/Config.pm

```
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = =
```

```
#
# TTBook::H2G2::Config
#
# DESCRIPTION
#   Configuration module for the Hitch-Hiker's Guide to the Galaxy web
#   application.
#
# AUTHOR
#   Andy Wardley <abw@wardley.org>
#
# COPYRIGHT
#   Copyright (C) 2003 Andy Wardley.  All Rights Reserved.
#
#   This module is free software; you can redistribute it and/or
#   modify it under the same terms as Perl itself.
#
# REVISION
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = =

package TTBook::H2G2::Config;

use strict;
use warnings;

our $VERSION  = 1.00;
our $ROOTDIR  = '/home/dent/web/guide';
our $ROOTURL  = '/H2G2';
our $ROOTCGI  = '/H2G2/guide';
our $DATABASE = {
    driver  => 'mysql',
    name    => 'guide',
    user    => 'dent',
    pass    => 'ruhtra',
    host    => '',
    port    => '',
};
our $TEMPLATE = {
```

```
    INCLUDE_PATH => [
      "$ROOTDIR/templates/cgi",
      "$ROOTDIR/templates/lib",
    ],
    PRE_PROCESS  => 'config',
    WRAPPER      => 'wrapper',
    VARIABLES    => {
        rooturl  => $ROOTURL,
        rootcgi  => $ROOTCGI,
    }
};
our $TEMPLATES = {
    index   => 'index.html',
    entry   => 'entry.html',
    entries => 'entries.html',
    error   => 'error.html',
};

1;
```

It defines $ROOTDIR, $ROOTURL, and $ROOTCGI to indicate the root directory, the root URL for documents, and the URL to access the application handler, respectively. The $DATABASE hash array defines the connection parameters for the TTBook::H2G2::Database module. The $TEMPLATE hash provides the familiar set of options for the Template module. Finally, the $TEMPLATES hash (note the plural) maps application actions (e.g., fetch entry, fetch list of entries, etc.) to presentation templates for displaying the outcome of the operation.

12.4.3 Application

Now that we have a storage module and the means to configure it, we can start to build our main application module:

```
package TTBook::H2G2;


use strict;
use Template;
use TTBook::H2G2::Config;
use TTBook::H2G2::Database;
use Class::Base;
use base qw( Class::Base );
```

The TTBook::H2G2 module is also a subclass of Class::Base and uses the configuration and database modules that we have already defined. We will be making several references to the $ROOTURL and $TEMPLATES items in the TTBook::H2G2::Config module, so we create local package variables to alias them, to save us from typing them repeatedly, if nothing else:

```
our $ROOTURL   = $TTBook::H2G2::Config::ROOTURL;
our $TEMPLATES = $TTBook::H2G2::Config::TEMPLATES;
```

The init( ) method, called by the new( ) constructor method in Class::Base, looks for three different configuration options. The first, database, can be used to provide a reference to a storage object other than the default. The second, template, allows the default template processing engine to be replaced. We'll not be using either of these in this example, but they illustrate how easy it is to use different modules to handle storage or presentation issues. The third option, templates, allows a different set of template mapping to be provided. These are merged with the default set, $TEMPLATES.

```
sub init {
    my ($self, $config) = @_;

    # user can provide custom database object
    $self->{ database } = $config->{ database };

    # same for template object
    $self->{ template } = $config->{ template };

    # merge user-supplied templates with defaults
    my $templates = $config->{ templates } || { };
    $self->{ templates } = {
        map { defined $templates->{ $_ }
                    ? ($_ => $templates->{ $_ })
                    : ($_ => $TEMPLATES->{ $_ })
        } keys %$TEMPLATES
    };

    return $self;
}
```

The database( ) method creates a TTBook::H2G2::Database object using the $DATABASE connection parameters defined in TTBook::H2G2::Config and caches it internally as the database item. If an object is already defined for database, either by being passed to new( ) as a configuration option or by being created by a previous call to the database( ) method, it is instead returned.

```
sub database {
    my $self = shift;

    return $self->{ database } ||= do {
        my $params = @_ && UNIVERSAL::isa($_[0], 'HASH') ? shift : { @_ };
        my $config = $TTBook::H2G2::Config::DATABASE;
        $config = {
            %$config,
            %$params,
        };
        TTBook::H2G2::Database->new($config)
            || $self->error(TTBook::H2G2::Database->error( ));
    };
}
```

The template( ) method is a factory method similar to database( ). In this case, it creates a Template object for processing templates for the application.

```
sub template {
    my $self = shift;

    return $self->{ template } ||= do {
        my $params = @_ && UNIVERSAL::isa($_[0], 'HASH') ? shift : { @_ };
        my $config = $TTBook::H2G2::Config::TEMPLATE;
        $config = {
            %$config,
            %$params,
        };
        Template->new($config)
            || return $self->error(Template->error( ));
    };
}
```

Now we can define some application-processing methods. The first is entry( ). It expects either a name or id parameter and then makes a call to the database item method to fetch the entry in question.

```
sub entry {
    my $self = shift;
    my $args = @_ && ref $_[0] eq 'HASH' ? shift : { @_ };
    my $database = $self->database( ) || return;
    my $entry;

    if (defined $args->{ id }) {
        return $database->item( get_entry_id => $args->{ id } )
            || $self->error($database->error( ));
    }
    elsif (defined $args->{ name }) {
        return $database->item( get_entry_name => $args->{ name } )
            || $self->error($database->error( ));
    }
    else {
        return $self->error("entry( ) expects 'name' or 'id' parameter");
    }
}
```

The search( ) method expects a search term as an argument. It calls the database list method to fetch a list of items returned by the entry_search query, forwarding the search term (modified as before) as an argument.

```
sub search {
    my ($self, $search) = @_;
```

```
    my $database = $self->database( ) || return;


    # change '*' to '%'

    $search =~ s/*/\%/g;


    return $database->list( entry_search => $search )

        || $self->error($database->error( ));

}
```

The run( ) method ties it all together. It is passed a reference to a hash array of request parameters. It inspects the parameters and dispatches the appropriate method to handle it: entry( ) or search( ). The entry or entries returned are added to the $params hash as template variables. The $template variable is also set to indicate the correct page template for the action.

```
sub run {

    my ($self, $params) = @_;

    my $templates = $self->{ templates };

    my ($tt, $template, $output);


    if (defined $params->{ name } || defined $params->{ id }) {

        # fetch entry if 'name' or 'id' specified

        my $entry = $self->entry($params);

        if ($entry) {

            $params->{ entry } = $entry;

            $template = $templates->{ entry };

        }

        else {

            $params->{ error } = $self->error( );

            $template = $templates->{ error };

        }

    }

    elsif (defined $params->{ search }) {

        # search for entries if 'search' specified

        my $entries = $self->search($params->{ search });

        if ($entries) {

            $params->{ entries } = $entries;

            $template = $templates->{ entries };

        }

        else {

            $params->{ error } = $self->error( );

            $template = $templates->{ error };

        }
```

```
    }
    else {
        return [ redirect => "$ROOTURL/index.html" ];
    }
```

If none of the parameters is set, a reference to a list is returned, indicating that the application should redirect to the *index.html* page relative to the $ROOTURL. We will be looking at the meaning of these return values shortly.

The final section of the run( ) method uses the Template object returned by the template( ) method ($tt) to process the page template named in the $template variable. The $params hash defines the template variables and the output is saved to the $output variable.

```
$tt = $self->template( )
    || return [ error => $self->error( ) ];


$tt->process($template, $params, \$output)
    || return [ error => $tt->error( ) ];
```

Whatever happens the method returns a reference to a list. The first item in the list is a string indicating the required action to be undertaken. A value of redirect should trigger a redirect to the URL specified as the second item in the list. A value of error denotes an error, with the second item in the list being an appropriate error message.

A value of output indicates that the page was successfully processed and that it has generated output that should be sent back to the client's browser. In this case, the second item in the list is a *reference* to the variable containing the output.

```
return [ output => \$output ];
```

The complete TTBook::H2G2 module is shown in Example 12-34.

Example 12-34. lib/TTBook/H2G2.pm

```
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = =
#
# TTBook::H2G2
#
# DESCRIPTION
#   A web application for a guide such as the Hitch Hiker's Guide to the
#   Galaxy.
#
# AUTHOR
#   Andy Wardley <abw@wardley.org>
#
# COPYRIGHT
#   Copyright (C) 2003 Andy Wardley.  All Rights Reserved.
#
#   This module is free software; you can redistribute it and/or
#   modify it under the same terms as Perl itself.
```

```
#
# REVISION
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = =

package TTBook::H2G2;

use strict;
use Template;
use TTBook::H2G2::Config;
use TTBook::H2G2::Database;
use Class::Base;
use base qw( Class::Base );

our $VERSION = sprintf("%d.%02d", q$Revision: 1.6 $ =~ /(\d+)\.(\d+)/);
our $DEBUG   = 0 unless defined $DEBUG;
our $ERROR   = '';
our $ROOTURL   = $TTBook::H2G2::Config::ROOTURL;
our $TEMPLATES = $TTBook::H2G2::Config::TEMPLATES;

#------------------------------------------------------------------------
# init(\%config)
#
# Initializer method called by Class::Base new( ) method.
#------------------------------------------------------------------------

sub init {
    my ($self, $config) = @_;

    # user can provide custom database object
    $self->{ database } = $config->{ database };

    # same for template object
    $self->{ template } = $config->{ template };

    # merge user-supplied templates with defaults
    my $templates = $config->{ templates } || { };
    $self->{ templates } = {
```

```
        map { defined $templates->{ $_ }
                  ? ($_ => $templates->{ $_ })
                  : ($_ => $TEMPLATES->{ $_ })
        } keys %$TEMPLATES
    };

    return $self;
}

#------------------------------------------------------------------------
# database( )
#
# Create or reuse existing database object.
#------------------------------------------------------------------------

sub database {
    my $self = shift;

    return $self->{ database } ||= do {
        my $params = @_ && UNIVERSAL::isa($_[0], 'HASH') ? shift : { @_ };
        my $config = $TTBook::H2G2::Config::DATABASE;
        $config = {
            %$config,
            %$params,
        };
        TTBook::H2G2::Database->new($config)
            || $self->error(TTBook::H2G2::Database->error( ));
    };
}

#------------------------------------------------------------------------
# template( )
#
# Create or reuse existing template processing object.
#------------------------------------------------------------------------

sub template {
    my $self = shift;
```

```
    return $self->{ template } ||= do {
        my $params = @_ && UNIVERSAL::isa($_[0], 'HASH') ? shift : { @_ };
        my $config = $TTBook::H2G2::Config::TEMPLATE;
        $config = {
            %$config,
            %$params,
        };
        Template->new($config)
            || return $self->error(Template->error( ));
    };
}

#------------------------------------------------------------------------
# entry( id => 12345 )
# entry( name => 'Earth' )
#
# Fetch an entry from the database.
#------------------------------------------------------------------------

sub entry {
    my $self = shift;
    my $args = @_ && ref $_[0] eq 'HASH' ? shift : { @_ };
    my $database = $self->database( ) || return;
    my $entry;

    if (defined $args->{ id }) {
        return $database->item( get_entry_id => $args->{ id } )
            || $self->error($database->error( ));
    }
    elsif (defined $args->{ name }) {
        return $database->item( get_entry_name => $args->{ name } )
            || $self->error($database->error( ));
    }
    else {
        return $self->error("entry( ) expects 'name' or 'id' parameter");
```

```
    }
}

#------------------------------------------------------------------------
# search($term)
#
# Search for items in the database based on a search term.
#------------------------------------------------------------------------

sub search {
    my ($self, $search) = @_;
    my $database = $self->database( ) || return;

    # change '*' to '%'
    $search =~ s/*/\%/g;

    return $database->list( entry_search => $search )
        || $self->error($database->error( ));
}

#------------------------------------------------------------------------
# run(\%params)
#
# Run web application.
#------------------------------------------------------------------------

sub run {
    my ($self, $params) = @_;
    my $templates = $self->{ templates };
    my ($tt, $template, $output);

    if (defined $params->{ name } || defined $params->{ id }) {
        # fetch entry if 'name' or 'id' specified
        my $entry = $self->entry($params);
        if ($entry) {
            $params->{ entry } = $entry;
            $template = $templates->{ entry };
        }
        else {
```

```
            $params->{ error } = $self->error( );
            $template = $templates->{ error };
        }
    }
    elsif (defined $params->{ search }) {
        # search for entries if 'search' specified
        my $entries = $self->search($params->{ search });
        if ($entries) {
            $params->{ entries } = $entries;
            $template = $templates->{ entries };
        }
        else {
            $params->{ error } = $self->error( );
            $template = $templates->{ error };
        }
    }
    else {
        return [ redirect => "$ROOTURL/index.html" ];
    }

    # process template and return output or error
    $tt = $self->template( )
        || return [ error => $self->error( ) ];

    $tt->process($template, $params, \$output)
        || return [ error => $tt->error( ) ];

    return [ output => \$output ];
}

1;
```

12.4.4 Apache mod_perl Interface Module

Finally we can add a module to provide the Apache-specific interface to the web application. This is shown in Example 12-35.

Example 12-35. lib/TTBook/H2G2/Apache.pm

```
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

```
 = =
#
# TTBook::H2G2::Apache
#
# DESCRIPTION
#   Apache/mod_perl handler for the H2G2 web application.
#
# AUTHOR
#   Andy Wardley <abw@wardley.org>
#
# COPYRIGHT
#   Copyright (C) 2003 Andy Wardley.  All Rights Reserved.
#
#   This module is free software; you can redistribute it and/or
#   modify it under the same terms as Perl itself.
#
# REVISION
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = =

package TTBook::H2G2::Apache;

use strict;
use Apache;
use Apache::Constants qw(OK SERVER_ERROR);
use TTBook::H2G2;

our $VERSION = 1.00;
our $H2G2APP;

sub handler {
    my $r = shift;
    my %params   = $r->method( ) eq 'POST'
                 ? $r->content( ) : $r->args( );

    # create or reuse existing application object
    $H2G2APP ||= TTBook::H2G2->new( )
        ||  return error($r, "Can't create webapp instance: ",
```

```
                     TTBook::H2G2->error( ));

    # run the application
    my $result = $H2G2APP->run(\%params)
        || return error($r, "Can't run webapp",
                        $H2G2APP->error( ));

    # handle the result
    my $action = shift @$result;

    if ($action eq 'output') {
        my $content = shift @$result;
        $r->content_type('text/html');
        $r->headers_out->add('Content-Length', length($$content));
        $r->send_http_header( );
        $r->print($$content);
        return OK;
    }
    elsif ($action eq 'redirect') {
        my $url = shift @$result;
        $r->internal_redirect($url);
    }
    elsif ($action eq 'error') {
        return error($r, @$result);
    }
    else {
        return error($r, "cannot handle action: $action");
    }
}

sub error {
    my $r = shift;
    $r->log_error(@_);
    return SERVER_ERROR;
}

1;
```

The $H2G2APP package variable is used to store a persistent reference to a TTBook::H2G2 application object. Inside the handler( ) method, we call the application run( ) method, passing the current set of request parameters as arguments. The result returned in stored in the $result variables.

```
my $result = $H2G2APP->run(\%params)
    || return error($r, "Can't run webapp",
                    $H2G2APP->error( ));
```

Then all that is left to do is to examine the first item in the $result list reference and perform the appropriate action: return content to the client, perform a redirect, or log an error.

```
my $action = shift @$result;

if ($action eq 'output') {
    my $content = shift @$result;
    $r->content_type('text/html');
    $r->headers_out->add('Content-Length', length($$content));
    $r->send_http_header( );
    $r->print($$content);
    return OK;
}
elsif ($action eq 'redirect') {
    my $url = shift @$result;
    $r->internal_redirect($url);
}
elsif ($action eq 'error') {
    return error($r, @$result);
}
else {
    return error($r, "cannot handle action: $action");
}
```

12.4.5 Apache Configuration

All that remains to deploy our web application under mod_perl is to write an Apache configuration file and restart the web server. Example 12-36 shows a typical configuration that should be copied into the main *httpd.conf* file or loaded through an Include directive.

Example 12-36. etc/ttguide.conf

```
Alias /H2G2/images/   /home/dent/guide/images/
Alias /H2G2/          /home/dent/guide/html/

<perl>
  use lib qw( /home/dent/guide/lib )
```

```
</perl>

PerlModule   TTBook::H2G2::Apache

<Location /H2G2/guide>
    SetHandler      perl-script
    PerlHandler     TTBook::H2G2::Apache
</Location>
```

Appendix A. Appendix: Configuration Options

The Template Toolkit is extremely configurable, and mastery of the many options takes time and practice, and requires that you read a lot of documentation. This appendix will help with the third requirement, as it contains a complete list of the Template Toolkit configuration options.

A.1 Template Toolkit Configuration Options

The options listed here can be used from a Perl program as part of the configuration hash that is passed to the Template->new( ) method. In many cases, an equivalent option is available for *ttree* users. In those cases, the *ttree* version is mentioned in the description. Finally, each option identifies the Template Toolkit module that is the primary consumer of that option.

A.1.1 ABSOLUTE

The ABSOLUTE flag is used to indicate whether templates specified with absolute filenames (e.g., */foo/bar*) should be processed. It is disabled by default, and any attempt to load a template by such a name will cause a file exception to be raised.

```
my $tt = Template->new({

   ABSOLUTE => 1,

});



# this is why it's disabled by default

[% INSERT /etc/passwd %]
```

On Win32 systems, the regular expression for matching absolute pathnames is tweaked slightly to also detect filenames that start with a drive letter and colon, such as:

```
C:/Foo/Bar
```

The *ttree* equivalent of this option is --absolute.

ABSOLUTE is used by Template::Provider.

A.1.2 ANYCASE

By default, directive keywords should be expressed in uppercase. The ANYCASE option can be set to allow directive keywords to be specified in any case.

```
# ANYCASE => 0 (default)

[% INCLUDE foobar %]       # OK

[% include foobar %]       # ERROR

[% include = 10   %]       # OK, 'include' is a variable



# ANYCASE => 1

[% INCLUDE foobar %]       # OK

[% include foobar %]       # OK

[% include = 10   %]       # ERROR, 'include' is reserved word
```

One side effect of enabling ANYCASE is that you cannot use a variable of the same name as a reserved word, regardless of case. The reserved words are currently as follows:

```
GET CALL SET DEFAULT INSERT INCLUDE PROCESS WRAPPER

IF UNLESS ELSE ELSIF FOR FOREACH WHILE SWITCH CASE

USE PLUGIN FILTER MACRO PERL RAWPERL BLOCK META

TRY THROW CATCH FINAL NEXT LAST BREAK RETURN STOP

CLEAR TO STEP AND OR NOT MOD DIV END
```

The only lowercase reserved words that cannot be used for variables, regardless of the ANYCASE option, are these operators:

```
and or not mod div
```

The *ttree* equivalent of this option is --anycase.

ANYCASE is used by Template::Parser.

A.1.3 AUTO_RESET

The AUTO_RESET option is set by default and causes the local BLOCKS cache for the Template::Context object to be reset on each call to the Template process( ) method. This ensures that any BLOCKs defined within a template will persist only until that template is finished processing. This prevents BLOCKs defined in one processing request from interfering with other independent requests subsequently processed by the same context object.

The BLOCKS item may be used to specify a default set of block definitions for the Template::Context object. Subsequent BLOCK definitions in templates will override these but they will be reinstated on each reset if AUTO_RESET is enabled (default), or if the Template::Context reset( ) method is called.

AUTO_RESET is used by Template::Service.

A.1.4 BLOCKS

The BLOCKS option can be used to predefine a default set of template blocks. These should be specified as a reference to a hash array mapping template names to template text, subroutines, or Template::Document objects.

```
my $tt = Template->new({
    BLOCKS => {
        header  => 'The Header.  [% title %]',
        footer  => sub { return $some_output_text },
        another => Template::Document->new({ ... }),
    },
});
```

BLOCKS is used by Template::Context.

A.1.5 CACHE_SIZE

The Template::Provider module caches compiled templates to avoid the need to re-parse template files or blocks each time they are used. The CACHE_SIZE option is used to limit the number of compiled templates that the module should cache.

By default, the CACHE_SIZE option is undefined and all compiled templates are cached. When set to any positive value, the cache will be limited to storing no more than that number of compiled templates. When a new template is loaded and compiled and the cache is full (i.e., the number of entries = = CACHE_SIZE), the least recently used compiled template is discarded to make room for the new one.

CACHE_SIZE can be set to 0 to disable caching altogether:

```
my $tt = Template->new({
    CACHE_SIZE => 64,   # only cache 64 compiled templates
});


my $tt = Template->new({
    CACHE_SIZE => 0,   # don't cache any compiled templates
});
```

CACHE_SIZE is used by Template::Provider.

A.1.6 COMPILE_EXT

From Version 2 onward, the Template Toolkit has the ability to compile templates to Perl code and save them to disk for subsequent use (i.e., cache persistence). The COMPILE_EXT option may be provided to specify a filename extension for compiled template files. It is undefined by default and no attempt will be made to read or write any compiled template files.

```
my $tt = Template->new({
    COMPILE_EXT => '.ttc',
});
```

If COMPILE_EXT is defined (and COMPILE_DIR, covered next, isn't) compiled template files with the *COMPILE_EXT* extension will be written to the same directory from which the source template files were loaded.

Compiling and subsequent reuse of templates happens automatically whenever the COMPILE_EXT or COMPILE_DIR options are set. The Template Toolkit will automatically reload and reuse compiled files when it finds them on disk. If the corresponding source file has been modified since the compiled version was written, it will load and recompile the source and write a new compiled version to disk.

This form of cache persistence offers significant benefits in terms of time and resources required to reload templates. Compiled templates can be reloaded by a simple call to Perl's `require( )`, leaving Perl to handle all the parsing and compilation. This is a Good Thing.

The *ttree* equivalent of this option is `--compile_ext`.

A.1.7 COMPILE_DIR

The COMPILE_DIR option is used to specify an alternate directory root under which compiled template files should be saved:

```
my $tt = Template->new({
    COMPILE_DIR => '/tmp/ttc',
});
```

The COMPILE_EXT option may also be specified to have a consistent file extension added to these files:

```
my $tt1 = Template->new({
    COMPILE_DIR => '/tmp/ttc',
    COMPILE_EXT => '.ttc1',
});


my $tt2 = Template->new({
    COMPILE_DIR => '/tmp/ttc',
    COMPILE_EXT => '.ttc2',
});
```

When COMPILE_EXT is undefined, the compiled template files have the same name as the original template files, but reside in a different directory tree.

Each directory in INCLUDE_PATH is replicated in full beneath the COMPILE_DIR directory. This example:

```
my $tt = Template->new({
    COMPILE_DIR  => '/tmp/ttc',
    INCLUDE_PATH => '/home/abw/templates:/usr/share/templates',
});
```

would create the following directory structure:

```
/tmp/ttc/home/abw/templates/

/tmp/ttc/usr/share/templates/
```

Files loaded from different INCLUDE_PATH directories will have their compiled forms saved in the relevant COMPILE_DIR directory.

On Win32 platforms, a filename may by prefixed by a drive letter and colon. For example:

```
C:/My Templates/header
```

The colon will be silently stripped from the filename when it is added to the COMPILE_DIR value(s) to prevent illegal filenames being generated. Any colon in COMPILE_DIR elements will be left intact. For example:

```
# Win32 only

my $tt = Template->new({

    DELIMITER    => ';',

    COMPILE_DIR  => 'C:/TT2/Cache',

    INCLUDE_PATH => 'C:/TT2/Templates;D:/My Templates',

});
```

This would create the following cache directories:

```
C:/TT2/Cache/C/TT2/Templates

C:/TT2/Cache/D/My Templates
```

The *ttree* equivalent of this option is --compile_ext=STRING.

COMPILE_EXT and COMPILE_DIR are used by Template::Provider.

A.1.8 CONSTANTS

The CONSTANTS option can be used to specify a hash array of template variables that are compile-time constants. These variables are resolved once when the template is compiled, and thus don't require further resolution at runtime. This results in significantly faster processing of the compiled templates, and can be used for variables that don't change from one request to the next.

```
my $tt = Template->new({

    CONSTANTS => {

        title   => 'A Demo Page',

        author  => 'Joe Random Hacker',

        version => 3.14,

    },

};
```

CONSTANTS is used by Template.

A.1.9 CONSTANT_NAMESPACE

Constant variables are accessed via the constants namespace by default:

```
[% constants.title %]
```

The CONSTANTS_NAMESPACE option can be set to specify an alternate namespace:

```
my $tt = Template->new({

    CONSTANTS => {

        title   => 'A Demo Page',

        # ...etc...
```

```
    },
    CONSTANTS_NAMESPACE => 'const',
};
```

In this case, the constants would then be accessed as:

```
[% const.title %]
```

CONSTANTS_NAMESPACE is used by Template.

A.1.10 NAMESPACE

The constant-folding mechanism just described is an example of a namespace handler. Namespace handlers can be defined to provide alternate parsing mechanisms for variables in different namespaces.

Under the hood, the Template module converts a constructor configuration such as:

```
my $tt = Template->new({
    CONSTANTS => {
        title   => 'A Demo Page',
        # ...etc...
    },
    CONSTANTS_NAMESPACE => 'const',
};
```

into one like:

```
my $tt = Template->new({
    NAMESPACE => {
        const => Template:::Namespace::Constants->new({
            title   => 'A Demo Page',
            # ...etc...
        }),
    },
};
```

You can use this mechanism to define multiple constant namespaces, or to install custom handlers of your own.

```
my $tt = Template->new({
    NAMESPACE => {
        site => Template:::Namespace::Constants->new({
            title   => "Wardley's Widgets",
            version => 2.718,
        }),
        author => Template:::Namespace::Constants->new({
            name  => 'Andy Wardley',
            email => 'abw@andywardley.com',
        }),
        voodoo => My::Namespace::Handler->new( ... ),
```

```
    },
};
```

Now you have two constant namespaces, for example:

```
[% site.title %]
[% author.name %]
```

You also have your own custom namespace handler installed for the voodoo namespace.

```
[% voodoo.magic %]
```

NAMESPACE is used by Template::Directive and Template::Parser.

A.1.11 CONTEXT

A reference to a Template::Context object is used to define a specific environment in which templates are processed. A Template::Context object is passed as the only parameter to the Perl subroutines that represent "compiled" template documents. Template subroutines make callbacks into the context object to access Template Toolkit functionality—for example, to INCLUDE or PROCESS another template (include( ) and process( ) methods, respectively), to USE a plugin (plugin( )) or instantiate a filter (filter( )) or to access the stash (stash( )) that manages variable definitions via the get( ) and set( ) methods.

```
my $tt = Template->new({
    CONTEXT => MyOrg::Template::Context->new({ ... }),
});
```

CONTEXT is used by Template::Service.

A.1.12 DEBUG

The DEBUG option can be used to enable debugging within the various different modules that comprise the Template Toolkit. The TemplateConstants module defines a set of DEBUG_XXXX constants that can be combined using the logical OR operator (|).

```
use Template::Constants qw( :debug );


my $tt = Template->new({
    DEBUG => DEBUG_PARSER | DEBUG_PROVIDER,
});
```

For convenience, you can also provide a string containing a list of lowercase debug options, separated by any nonword characters:

```
my $tt = Template->new({
    DEBUG => 'parser, provider',
});
```

The following DEBUG_XXXX flags can be used:

DEBUG_SERVICE

Enables general debugging messages for the TemplateService module.

DEBUG_CONTEXT

Enables general debugging messages for the TemplateContext module.

DEBUG_PROVIDER

Enables general debugging messages for the TemplateProvider module.

DEBUG_PLUGINS

Enables general debugging messages for the TemplatePlugins module.

DEBUG_FILTERS

Enables general debugging messages for the TemplateFilters module.

DEBUG_PARSER

Causes the TemplateParser to generate debugging messages that show the Perl code generated by parsing and compiling each template.

DEBUG_UNDEF

Causes the Template Toolkit to throw an undef error whenever it encounters an undefined variable value.

DEBUG_DIRS

Causes the Template Toolkit to generate comments indicating the source file, line, and original text of each directive in the template. These comments are embedded in the template output using the format defined in the DEBUG_FORMAT configuration item, or a simple default format if unspecified.

For example, the following template fragment:

```
Hello World
```

would generate this output:

```
## input text line 1 :  ##

Hello

## input text line 2 : World ##

World
```

DEBUG_ALL

Enables all debugging messages.

DEBUG_CALLER

Causes all debug messages that aren't newline-terminated to have the filename and line number of the caller appended to them.

A.1.13 DEBUG_FORMAT

The DEBUG_FORMAT option can be used to specify a format string for the debugging messages generated via the DEBUG_DIRS option described earlier. Any occurrences of $file, $line, or $text will be replaced with the current filename, line, or directive text, respectively. Notice how the format is single-quoted to prevent Perl from interpolating those tokens as variables:

```
my $tt = Template->new({

    DEBUG => 'dirs',

    DEBUG_FORMAT => '<!-- $file line $line : [% $text %] -->',

});
```

The following template fragment:

```
[% foo = 'World' %]

Hello [% foo %]
```

would then generate this output:

```
<!-- input text line 2 : [% foo = 'World' %] -->

Hello <!-- input text line 3 : [% foo %] -->World
```

The DEBUG directive can also be used to set a debug format within a template:

```
[% DEBUG format '<!-- $file line $line : [% $text %] -->' %]
```

The *ttree* equivalent of this option is --debug (or -dbg).

DEBUG_FORMAT is used by Template::Context.

A.1.14 DEFAULT

The DEFAULT option can be used to specify a default template that should be used whenever a specified template can't be found in INCLUDE_PATH:

```
my $tt = Template->new({

    DEFAULT => 'notfound.html',

});
```

If a nonexistent template is requested through the Template process( ) method or by an INCLUDE, PROCESS, or WRAPPER directive, the DEFAULT template will instead be processed, if defined. Note that the DEFAULT template is not used when templates are specified with absolute or relative filenames, or as a reference to an input filehandle or text string.

The ttree equivalent of this option is --default=TEMPLATE.

DEFAULT is used by Template::Provider.

A.1.15 DELIMITER

This is used to provide an alternative delimiter character sequence for separating paths specified in INCLUDE_PATH. The default value for DELIMITER is :.

```
my $tt = Template->new({

    DELIMITER    => '; ',

    INCLUDE_PATH => 'C:/HERE/NOW; D:/THERE/THEN',

});
```

On Win32 systems, the default delimiter is a little more intelligent, splitting paths only on : characters that aren't followed by a /. This means that the following should work as planned, splitting INCLUDE_PATH into two separate directories, *C:/foo* and *C:/bar*:

```
# on Win32 only

my $tt = Template->new({

    INCLUDE_PATH => 'C:/Foo:C:/Bar'

});
```

However, if you're using Win32, it's recommended that you explicitly set the DELIMITER character to something else (e.g., ;) rather than rely on this subtle magic.

DELIMITER is used by Template::Service and Template::Provider.

A.1.16 ERROR

The ERROR (or ERRORS if you prefer) configuration item can be used to name a single template or specify a hash array mapping exception types to templates that should be used for error handling. If an uncaught exception is raised from within a template, the appropriate error template will instead be processed.

If specified as a single value, that template will be processed for all uncaught exceptions:

```
my $tt = Template->new({
    ERROR => 'error.html'
});
```

If the ERROR item is a hash reference, the keys are assumed to be exception types and the relevant template for a given exception will be selected. A "default" template may be provided for the general case. Note that ERROR can be pluralized to ERRORS if you find it more appropriate in this case.

```
my $tt = Template->new({
    ERRORS => {
        user     => 'user/index.html',
        dbi      => 'error/database',
        default  => 'error/default',
    },
});
```

In this example, any user exceptions thrown will cause the *user/index.html* template to be processed. dbi errors are handled by *error/database* and all others by the *error/default* template. Any *PRE_PROCESS* and/or *POST_PROCESS* templates will also be applied to these error templates.

Note that exception types are hierarchical, and a foo handler will catch all foo.* errors (e.g., foo.bar, foo.bar.baz) if a more specific handler isn't defined. Be sure to quote any exception types that contain periods to prevent Perl from concatenating them into a single string (i.e., user.passwd is parsed as 'user'.'passwd').

```
my $tt = Template->new({
    ERROR => {
        'user.login'  => 'user/login.html',
        'user.passwd' => 'user/badpasswd.html',
        'user'        => 'user/index.html',
        'default'     => 'error/default',
    },
});
```

In this example, any template processed by the $tt object, other templates, or code called from within can raise a user.login exception and have the service redirect to the *user/login.html* template. Similarly, a user.passwd exception has a specific handling template, *user/badpasswd.html*, while all other user or user.* exceptions cause a redirection to the *user/index.html* page. All other exception types are handled by *error/default*.

Exceptions can be raised in a template using the THROW directive:

```
[% THROW user.login 'no user id: please login' %]
```

or by calling the throw( ) method on the current Template::Context object:

```
$context->throw('user.passwd', 'Incorrect Password');
$context->throw('Incorrect Password');    # type 'undef'
```

or from Perl code by calling die( ) with a Template::Exception object:

```
die (Template::Exception->new('user.denied', 'Invalid User ID'));
```

or by simply calling die( ) with an error string. This is automatically caught and converted to an exception of undef type, which can then be handled in the usual way:

```
die "I'm sorry Dave, I can't do that";
```

The *ttree* equivalent for this option is --error=TEMPLATE.

ERROR is used by Template::Service.

A.1.17 EVAL_PERL

This flag is used to indicate whether PERL and/or RAWPERL blocks should be evaluated. By default, it is disabled, and any PERL or RAWPERL blocks encountered will raise exceptions of type perl with the message EVAL_PERL not set. Note, however, that any RAWPERL blocks should always contain valid Perl code, regardless of the EVAL_PERL flag. The parser will fail to compile templates that contain invalid Perl code in RAWPERL blocks, and will throw a file exception.

If EVAL_PERL is set when a template is compiled, all PERL and RAWPERL blocks will be included in the compiled template. If EVAL_PERL isn't set, Perl code will be generated, which *always* throws a perl exception with the message EVAL_PERL not set *whenever* the compiled template code is run.

Thus, you must have EVAL_PERL set if you want your compiled templates to include PERL and RAWPERL blocks.

At some point in the future, using a different invocation of the Template Toolkit, you may come to process such a precompiled template. Assuming the EVAL_PERL option was set at the time the template was compiled, the output of any RAWPERL blocks will be included in the compiled template and will get executed when the template is processed. This will happen regardless of the runtime EVAL_PERL status.

Regular PERL blocks are a little more cautious, however. If the EVAL_PERL flag isn't set for the *current* context—that is, the one that is trying to process it—it will throw the familiar perl exception with the message EVAL_PERL not set.

Thus you can compile templates to include PERL blocks, but optionally disable them when you process them later. Note, however, that it is possible for a PERL block to contain a Perl BEGIN { # some code } block that is always get run regardless of the runtime EVAL_PERL status. Thus, if you set EVAL_PERL when compiling templates, it is assumed that you trust the templates to Do The Right Thing. Otherwise, you must accept the fact that there's no bulletproof way to prevent any included code from trampling around in the living room of the runtime environment, making a real nuisance of itself if it really wants to. If you don't like the idea of such uninvited guests causing a bother, you can accept the default and keep EVAL_PERL disabled.

The *ttree* equivalent of this option is --eval_perl.

EVAL_PERL is used by Template::Directive, Template::Context, and Template::Filters.

A.1.18 FACTORY

FACTORY defines the class used by Template::Parser to generate Perl code for elements of the grammar, which defaults to Template::Directive.

FACTORY is used by Template::Parser.

A.1.19 FILTERS

The FILTERS option can be used to specify custom filters that can then be used with the FILTER directive like any other. These are added to the standard filters, which are available by default. Filters specified via this option will mask any standard filters of the same name.

The FILTERS option should be specified as a reference to a hash array in which each key represents the name of a filter. The corresponding value should contain a reference to an array containing a subroutine reference and a flag that indicates whether the filter is static (0) or dynamic (1). A filter may also be specified as a solitary subroutine reference and is assumed to be static.

```
$tt = Template->new({
    FILTERS => {
        'sfilt1' =>   \&static_filter,      # static
        'sfilt2' => [ \&static_filter, 0 ], # same as above
        'dfilt1' => [ \&dynamic_filter_factory, 1 ],
    },
});
```

Additional filters can be specified at any time by calling the define_filter( ) method on the current Template::Context object. The method accepts a filter name, a reference to a filter subroutine, and an optional flag to indicate whether the filter is dynamic.

```
my $context = $template->context( );
$context->define_filter('new_html', \&new_html);
$context->define_filter('new_repeat', \&new_repeat, 1);
```

In static filters, a single subroutine reference is used for all invocations of a particular filter. Filters that don't accept any configuration parameters (e.g., html) can be implemented statically. The subroutine reference is simply returned when that particular filter is requested. The subroutine is called to filter the output of a template block that is passed as the only argument. The subroutine should return the modified text.

```
sub static_filter {
    my $text = shift;
    # do something to modify $text...
    return $text;
}
```

The following template fragment:

```
[% FILTER sfilt1 %]
Blah blah blah.
[% END %]
```

is approximately equivalent to:

```
&static_filter("\nBlah blah blah.\n");
```

Filters that can accept parameters (e.g., truncate) should be implemented dynamically. In this case, the subroutine is taken to be a filter factory that is called to create a unique filter subroutine each time one is requested. A reference to the current Template::Context object is passed as the first parameter, followed by any additional parameters specified. The subroutine should return another subroutine reference (usually a closure) that implements the filter.

```
sub dynamic_filter_factory {
    my ($context, @args) = @_;

    return sub {
        my $text = shift;
        # do something to modify $text...
        return $text;
    }
}
```

The following template fragment:

```
[% FILTER dfilt1(123, 456) %]
Blah blah blah
[% END %]
```

is approximately equivalent to:

```
my $filter = &dynamic_filter_factory($context, 123, 456);
&$filter("\nBlah blah blah.\n");
```

FILTERS is used by Template::Context.

A.1.20 GRAMMAR

The GRAMMAR configuration item can be used to specify an alternate grammar for the parser. This allows a modified or entirely new template language to be constructed and used by the Template Toolkit.

Source templates are compiled to Perl code by the Template::Parser using the Template::Grammar (by default) to define the language structure and semantics. Compiled templates are thus inherently "compatible" with each other, and there is nothing to prevent any number of different template languages from being compiled and used within the same Template Toolkit processing environment (other than the usual time and memory constraints).

The Template::Grammar file is constructed from a YACC-like grammar (using Parse::YAPP) and a skeleton module template. These files are provided, along with a small script to rebuild the grammar, in the *parser* subdirectory of the distribution. You don't have to know or worry about these unless you want to hack on the template language or define your own variant. A README file in the same directory provides some small guidance, but it is assumed that you know what you're doing if you venture herein. If you grok LALR parsers, then you should find it comfortably familiar.

By default, an instance of the default Template::Grammar will be created and used automatically if a GRAMMAR item isn't specified:

```
use MyOrg::Template::Grammar;


my $tt = Template->new({
    GRAMMAR = MyOrg::Template::Grammar->new( );
});
```

GRAMMAR is used by Template::Parser.

A.1.21 INCLUDE_PATH

INCLUDE_PATH is used to specify one or more directories in which template files are located. When a template is requested that isn't defined locally as a BLOCK, each INCLUDE_PATH directory is searched in turn to locate the template file. Multiple directories can be specified as a reference to a list or as a single string where each directory is delimited by :.

```
my $tt = Template->new({
    INCLUDE_PATH => '/usr/local/templates',
});


my $tt = Template->new({
    INCLUDE_PATH => '/usr/local/templates:/tmp/my/templates',
});


my $tt = Template->new({
    INCLUDE_PATH => [ '/usr/local/templates',
                      '/tmp/my/templates' ],
});
```

On Win32 systems, a little extra magic is invoked, ignoring delimiters that have : followed by a / or \. This avoids confusion when using directory names such as *C:\Blah Blah*.

When specified as a list, the INCLUDE_PATH path can contain elements that dynamically generate a list of INCLUDE_PATH directories. These generator elements can be specified as a reference to a subroutine or an object that implements a paths( ) method.

```
my $tt = Template->new({
    INCLUDE_PATH => [ '/usr/local/templates',
                      \&incpath_generator,
                      My::IncPath::Generator->new( ... ) ],
});
```

Each time a template is requested and the INCLUDE_PATH examined, the subroutine or object method will be called. A reference to a list of directories should be returned. Generator subroutines should report errors using die( ). A generator object should return undef and make an error available via its error( ) method.

For example:

```
sub incpath_generator {

    # ...some code...

    if ($all_is_well) {
        return \@list_of_directories;
    }
    else {
        die "cannot generate INCLUDE_PATH...\n";
    }
}
```

or:

```
package My::IncPath::Generator;

# Template::Base (or Class::Base) provides error( ) method
use Template::Base;
use base qw( Template::Base );

sub paths {
    my $self = shift;

    # ...some code...

    if ($all_is_well) {
        return \@list_of_directories;
    }
    else {
        return $self->error("cannot generate INCLUDE_PATH...\n");
    }
}

1;
```

The *ttree* equivalent of this option is --lib=DIR (or -l DIR).

INCLUDE_PATH is used by Template::Provider.

A.1.22 INTERPOLATE

The INTERPOLATE flag, when set to any true value, will cause variable references in plain text (i.e., not surrounded by START_TAG and END_TAG) to be recognized and interpolated accordingly:

```
my $tt = Template->new({

    INTERPOLATE => 1,

});
```

Variables should be prefixed by a $ to identify them. Curly braces can be used in the familiar Perl/shell style to explicitly scope the variable name where required.

```
# INTERPOLATE => 0

<a href="http://[% server %]/[% help %]">

<img src="[% images %]/help.gif"></a>

[% myorg.name %]


# INTERPOLATE => 1

<a href="http://$server/$help">

<img src="$images/help.gif"></a>

$myorg.name


# explicit scoping with {  }

<img src="$images/${icon.next}.gif">
```

Note that a limitation in Perl's regex engine restricts the maximum length of an interpolated template to around 32 kilobytes or possibly less. Files that exceed this limit in size will typically cause Perl to dump core with a segmentation fault. If you routinely process templates of this size, you should disable INTERPOLATE or split the templates in several smaller files or blocks that can then be joined backed together via PROCESS or INCLUDE.

The *ttree* equivalent for this option is --interpolate.

INTERPOLATE is used by Template::Parser.

A.1.23 LOAD_FILTERS

The LOAD_FILTERS option can be used to specify a list of provider objects (i.e., they implement the fetch( ) method) that are responsible for returning and/or creating filter subroutines. The Template::Context filter( ) method queries each provider in turn in a "Chain of Responsibility" as per the template( ) and plugin( ) methods.

```
my $tt = Template->new({

    LOAD_FILTERS => [

        MyTemplate::Filters->new( ),

        Template::Filters->new( ),

    ],

});
```

By default, a single Template::Filters object is created for the LOAD_FILTERS list.

LOAD_FILTERS is used by Template::Context.

A.1.24 LOAD_PERL

If a plugin cannot be loaded using the PLUGINS or PLUGIN_BASE approaches, the provider can make a final attempt to load the module without prepending any prefix to the module path. This allows regular Perl modules (i.e., those that don't reside in Template::Plugin or some other such namespace) to be loaded and used as plugins.

By default, the LOAD_PERL option is set to 0 and no attempt will be made to load any Perl modules that aren't named explicitly in the PLUGINS hash or that don't reside in a package as named by one of the PLUGIN_BASE components.

Plugins loaded using the PLUGINS or PLUGIN_BASE receive a reference to the current context object as the first argument to the new( ) constructor. Modules loaded using LOAD_PERL are assumed to not conform to the plugin interface. They must provide a new( ) class method for instantiating objects, which will not receive a reference to the context as the first argument. Plugin modules should provide a load( ) class method (or inherit the default one from the Template::Plugin base class) that is called the first time the plugin is loaded. Regular Perl modules need not provide a load( ) method. In all other respects, regular Perl objects and Template Toolkit plugins are identical.

If a particular Perl module does not conform to the common, but not unilateral, new( ) constructor convention, a simple plugin wrapper can be written to interface to it.

The *ttree* equivalent of this option is --load_perl.

LOAD_PERL is used by Template::Plugins.

A.1.25 LOAD_PLUGINS

The LOAD_PLUGINS options can be used to specify a list of provider objects (i.e., they implement the fetch( ) method) that are responsible for loading and instantiating template plugin objects. The Template::Content plugin( ) method queries each provider in turn in a "Chain of Responsibility" as per the template( ) and filter( ) methods.

```
my $tt = Template->new({

    LOAD_PLUGINS => [

            MyOrg::Template::Plugins->new({ ... }),

            Template::Plugins->new({ ... }),

    ],

});
```

By default, a single Template::Plugins object is created using the current configuration hash. Configuration items destined for the Template::Plugins constructor may be added to the Template constructor.

```
my $tt = Template->new({

    PLUGIN_BASE => 'MyOrg::Template::Plugins',

    LOAD_PERL   => 1,

});
```

LOAD_PLUGINS is used by Template::Context.

A.1.26 LOAD_TEMPLATES

The LOAD_TEMPLATE option can be used to provide a reference to a list of Template::Provider objects or subclasses thereof that will take responsibility for loading and compiling templates.

```
my $tt = Template->new({

    LOAD_TEMPLATES => [

        MyOrg::Template::Provider->new({ ... }),

        Template::Provider->new({ ... }),

    ],

});
```

When a PROCESS, INCLUDE, or WRAPPER directive is encountered, the named template may refer to a locally defined BLOCK or a file relative to the INCLUDE_PATH (or an absolute or relative path if the appropriate ABSOLUTE or RELATIVE options are set). If a BLOCK definition can't be found (see Example 7-4 in the Section 7.3.5 for a discussion of BLOCK locality), each LOAD_TEMPLATES provider object is queried in turn via the fetch( ) method to see whether it can supply the required template. Each provider can return a compiled template or an error, or can decline to service the request, in which case the responsibility is passed to the next provider. If none of the providers can service the request, a not found error is returned. The same basic provider mechanism is also used for the INSERT directive, but it bypasses any BLOCK definitions and doesn't attempt to parse or process the contents of the template file.

This is an implementation of the "Chain of Responsibility" design pattern as described in *Design Patterns*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addision-Wesley).

If LOAD_TEMPLATES is undefined, a single default provider will be instantiated using the current configuration parameters. For example, the Template::Provider INCLUDE_PATH option can be specified in the Template configuration and will be correctly passed to the provider's constructor method:

```
my $tt = Template->new({

    INCLUDE_PATH => '/here:/there',

});
```

LOAD_TEMPLATES is used by Template::Context.

A.1.27 OUTPUT_PATH

OUTPUT_PATH allows a directory to be specified into which output files should be written. An output file can be specified by the OUTPUT option, or passed by name as the third parameter to the Template process( ) method.

```
my $template = Template->new({

    INCLUDE_PATH => "/tmp/src",

    OUTPUT_PATH  => "/tmp/dest",

});


my $vars = {

    ...

};


foreach my $file ('foo.html', 'bar.html') {

    $template->process($file, $vars, $file)

        || die $template->error( );

}
```

This example will read the input files */tmp/src/foo.html* and */tmp/src/bar.html*, and write the processed output to */tmp/dest/foo.html* and */tmp/dest/bar.htm*l, respectively.

The *ttree* equivalent of this option is --dest=DIR (or -d DIR).

OUTPUT_PATH is used by Template and Template::Filters.

A.1.28 OUTPUT

This is the default output location or handler. This may be specified as a filename (relative to OUTPUT_PATH, if defined, or the current working directory if not specified absolutely); a filehandle (e.g., GLOB or IO::Handle) opened for writing; a reference to a text string to that the output is appended (the string isn't cleared); a reference to a subroutine that is called, passing the output text as an argument; a reference to an array onto which the content will be push( )ed; or a reference to any object that supports the print( ) method. This latter option includes the Apache::Request object which is passed as the argument to Apache/mod_perl handlers (see Example A-1 through Example A-6).

Example A-1. Filename

```
my $tt = Template->new({

    OUTPUT => "/tmp/foo",

});
```

Example A-2. Text string

```
my $output = '';

my $tt = Template->new({
    OUTPUT => \$output,
});
```

Example A-3. Filehandle

```
open (TOUT, "> $file") || die "$file: $!\n";

my $tt = Template->new({
    OUTPUT => *TOUT,
});
```

Example A-4. Subroutine

```
sub output { my $out = shift; print "OUTPUT: $out" }

my $tt = Template->new({
    OUTPUT => \&output,
});
```

Example A-5. Array reference

```
my $tt = Template->new({
    OUTPUT => \@output,
})
```

Example A-6. Apache/mod_perl handler

```
sub handler {
    my $r = shift;

    my $tt = Template->new({
        OUTPUT => $r,
    });
    ...
}
```

The default OUTPUT location can be overridden by passing a third parameter to the Template process( ) method. This can be specified as any of the following argument types:

```
$tt->process($file, $vars, "/tmp/foo");

$tt->process($file, $vars, "bar");

$tt->process($file, $vars, *MYGLOB);

$tt->process($file, $vars, \@output);

$tt->process($file, $vars, $r);  # Apache::Request

...
```

OUTPUT is used by Template.

A.1.29 PARSER

The Template::Parser module implements a parser object for compiling templates into Perl code, which can then be executed. A default object of this class is created automatically and then used by Template::Provider whenever a template is loaded and requires compilation. The PARSER option can be used to provide a reference to an alternate parser object.

```
my $tt = Template->new({

    PARSER => MyOrg::Template::Parser->new({ ... }),

});
```

PARSER is used by Template::Provider.

A.1.30 PLUGIN_BASE

If a plugin is not defined in the PLUGINS hash, PLUGIN_BASE is used to attempt to construct a correct Perl module name that can be successfully loaded.

PLUGIN_BASE can be specified as a single value or as a reference to an array of multiple values. The default PLUGIN_BASE value, Template::Plugin, is always added to the end of the PLUGIN_BASE list (a single value is first converted to a list). Each value should contain a Perl package name to which the requested plugin name is appended. For example:

```
my $tt = Template->new({

    PLUGIN_BASE => 'MyOrg::Template::Plugin',

});


[% USE Foo %]    # => MyOrg::Template::Plugin::Foo

                  or        Template::Plugin::Foo
```

or:

```
my $tt = Template->new({

    PLUGIN_BASE => [   'MyOrg::Template::Plugin',

                    'YourOrg::Template::Plugin'  ],

});


[% USE Foo %]    # =>   MyOrg::Template::Plugin::Foo

                  or YourOrg::Template::Plugin::Foo

                  or         Template::Plugin::Foo
```

The *ttree* equivalent for this option is --plugin_base=PACKAGE.

PLUGIN_BASE is used by Template::Plugins.

A.1.31 PLUGINS

The PLUGINS option can be used to provide a reference to a hash array that maps plugin names to Perl module names. A number of standard plugins are defined (e.g., table, cgi, dbi, etc.) that map to their corresponding Template::Plugin::* counterparts. These can be redefined by values in the PLUGINS hash:

```
my $tt = Template->new({
    PLUGINS => {
        cgi => 'MyOrg::Template::Plugin::CGI',
        foo => 'MyOrg::Template::Plugin::Foo',
        bar => 'MyOrg::Template::Plugin::Bar',
    },
});
```

The USE directive is used to create plugin objects and does so by calling the plugin( ) method on the current Template::Context object. If the plugin name is defined in the PLUGINS hash, the corresponding Perl module is loaded via require( ). The context then calls the load( ) class method, which should return the class name (default and general case) or a prototype object against which the new( ) method can be called to instantiate individual plugin objects.

If the plugin name is not defined in the PLUGINS hash, the PLUGIN_BASE and/or LOAD_PERL options come into effect.

PLUGINS is used by Template::Plugins.

A.1.32 PRE_CHOMP, POST_CHOMP

Anything outside a directive tag is considered plain text and is generally passed through unaltered (but see the INTERPOLATE option for text that's altered as it is passed through). This includes all whitespace and newline characters surrounding directive tags. Directives that don't generate any output will leave gaps in the output document.

For example, this:

```
Foo
[% a = 10 %]
Bar
```

will output this:

```
Foo


Bar
```

The PRE_CHOMP and POST_CHOMP options can help to clean up some of this extraneous whitespace. Both are disabled by default.

```
my $tt = Template->new({
    PRE_CHOMP  => 1,
    POST_CHOMP => 1,
});
```

With PRE_CHOMP set to 1, the newline and whitespace preceding a directive at the start of a line will be deleted. This has the effect of concatenating a line that starts with a directive onto the end of the previous line.

```
        Foo <----------.
                       |
    ,---(PRE_CHOMP)----'
    |
    `-- [% a = 10 %] --.
                       |
```

```
  ,---(POST_CHOMP)---'

  |

  `-> Bar
```

With POST_CHOMP set to 1, any whitespace after a directive up to and including the newline will be deleted. This has the effect of joining a line that ends with a directive onto the start of the next line.

If PRE_CHOMP or POST_CHOMP is set to 2, instead of removing all the whitespace, the whitespace will be collapsed to a single space. This is useful for HTML, where (usually) a contiguous block of whitespace is rendered the same as a single space.

You may use the CHOMP_NONE, CHOMP_ALL, and CHOMP_COLLAPSE constants from the Template::Constants module to deactivate chomping, remove all whitespace, or collapse whitespace to a single space.

PRE_CHOMP and POST_CHOMP can be activated for individual directives by placing a dash (-) immediately at the start and/or end of the directive:

```
[% FOREACH user = userlist %]

  [%- user -%]

[% END %]
```

The - character activates both PRE_CHOMP and POST_CHOMP for the one directive [%- name -%]. Thus, the template will be processed as if written:

```
[% FOREACH user = userlist %][% user %][% END %]
```

Note that this is the same as if PRE_CHOMP and POST_CHOMP were set to CHOMP_ALL; the only way to get the CHOMP_COLLAPSE behavior is to set PRE_CHOMP or POST_CHOMP accordingly. If PRE_CHOMP or POST_CHOMP is already set to CHOMP_COLLAPSE, using - will give you CHOMP_COLLAPSE behavior, not CHOMP_ALL behavior.

Similarly, + characters can be used to disable PRE_CHOMP or POST_CHOMP (i.e., leave the whitespace/newline intact) options on a per-directive basis:

```
[% FOREACH user = userlist %]

User: [% user +%]

[% END %]
```

With POST_CHOMP enabled, the previous example would be parsed as if written:

```
[% FOREACH user = userlist %]User: [% user %]

[% END %]
```

The *ttree* equivalents of these options are --pre_chomp and --post_chomp.

PRE_CHOMP and POST_CHOMP are used by Template::Parser.

A.1.33 PRE_DEFINE, VARIABLES

The PRE_DEFINE option (or VARIABLES; they're equivalent) can be used to specify a hash array of template variables that should be used to preinitialize the stash when it is created. These items are ignored if the STASH item is defined:

```
my $tt = Template->new({

  VARIABLES => {

    title   => 'A Demo Page',

    author  => 'Joe Random Hacker',

    version => 3.14,

  },

};
```

or:

```
my $tt = Template->new({

    PRE_DEFINE => {

        title   => 'A Demo Page',

        author  => 'Joe Random Hacker',

        version => 3.14,

    },

};
```

The *ttree* equivalent of this option is --define var=value.

PRE_DEFINE is used by Template::Context.

A.1.34 PRE_PROCESS, POST_PROCESS

These values may be set to contain the name(s) of template files (relative to INCLUDE_PATH) that should be processed immediately before and/or after each template. These do not get added to templates processed into a document via directives such as INCLUDE, PROCESS, WRAPPER, etc.

```
my $tt = Template->new({

    PRE_PROCESS  => 'header',

    POST_PROCESS => 'footer',

};



$tt->process('mydoc.html')

    || die $tt->error( );
```

Multiple templates may be specified as a reference to a list. Each is processed in the order defined.

```
my $tt = Template->new({

    PRE_PROCESS  => [ 'config', 'header' ],

    POST_PROCESS => 'footer',

};
```

Alternately, multiple templates may be specified as a single string, delimited by the : character. This delimiter string can be changed via the DELIMITER option.

```
my $tt = Template->new({

    PRE_PROCESS  => 'config:header',

    POST_PROCESS => 'footer',

};
```

The PRE_PROCESS and POST_PROCESS templates are evaluated in the same variable context as the main document and may define or update variables for subsequent use.

The Template::Document object representing the main template being processed is available within PRE_PROCESS and POST_PROCESS templates as the template variable. Metadata items defined via the META directive may be accessed accordingly.

Example A-7 through Example A-10 show the *config*, *header*, *footer*, and *mydoc.html* files.

Example A-7. config

```
[% # set some site-wide variables
   bgcolor = '#ffffff'
   version = 2.718
%]
```

Example A-8. header

```
[% DEFAULT title = 'My Funky Web Site' %]
<html>
<head>
<title>[% title %]</title>
</head>
<body bgcolor="[% bgcolor %]">
```

Example A-9. footer

```
<hr />
Version [% version %]
</body>
</html>
```

Example A-10. mydoc.html

```
[% META title = 'My Document Title' %]
blah blah blah
...
```

The *ttree* equivalents for these options are --pre_process=TEMPLATE and --post_process=TEMPLATE.

PRE_PROCESS and POST_PROCESS are used by Template::Service.

A.1.35 PROCESS

The PROCESS option may be set to contain the name(s) of template files (relative to INCLUDE_PATH) that should be processed instead of the main template passed to the Template process( ) method. This can be used to apply consistent wrappers around all templates, similar to the use of PRE_PROCESS and POST_PROCESS templates.

```
my $tt = Template->new({
    PROCESS  => 'content',
};

# processes 'content' instead of 'foo.html'
$tt->process('foo.html');
```

A reference to the original template is available in the template variable. Metadata items can be inspected and the template can be processed by specifying it as a variable reference (i.e., prefixed by $) to an INCLUDE, PROCESS, or WRAPPER directive.

Example A-11, Example A-12, and Example A-13 show the *content*, *foo.html*, and *output* files.

Example A-11. content

```
<html>
<head>
<title>[% template.title %]</title>
</head>

<body>
[% PROCESS $template %]
<hr />
&copy; Copyright [% template.copyright %]
</body>
</html>
```

Example A-12. foo.html

```
[% META
   title     = 'The Foo Page'
   author    = 'Fred Foo'
   copyright = '2000 Fred Foo'
%]
<h1>[% template.title %]</h1>
Welcome to the Foo Page, blah blah blah
```

Example A-13. output

```
<html>
<head>
<title>The Foo Page</title>
</head>

<body>
<h1>The Foo Page</h1>
Welcome to the Foo Page, blah blah blah
<hr />
&copy; Copyright 2000 Fred Foo
</body>
</html>
```

The *ttree* equivalent of this option is --process=TEMPLATE.

PROCESS is used by Template::Service.

A.1.36 RECURSION

The template processor will raise a file exception if it detects direct or indirect recursion into a template. Setting this option to any true value will allow templates to include each other recursively.

The *ttree* equivalent of this option is --recursion.

RECURSION is used by Template::Context and Template::Document.

A.1.37 RELATIVE

The RELATIVE flag is used to indicate whether templates specified with filenames relative to the current directory (e.g., *./foo/bar* or *../../some/where/else*) should be loaded. It is also disabled by default, and will raise a file error if such template names are encountered.

```
my $tt = Template->new({
    RELATIVE => 1,
});


[% INCLUDE ../logs/error.log %]
```

The *ttree* equivalent of this option is --relative.

RELATIVE is used by Template::Provider.

A.1.38 SERVICE

This provides a reference to a Template::Service object, or subclass thereof, to which the Template module should delegate. If unspecified, a Template::Service object is automatically created using the current configuration hash.

```
my $tt = Template->new({
    SERVICE => MyOrg::Template::Service->new({ ... }),
});
```

SERVICE is used by Template.

A.1.39 STASH

This provides a reference to a Template::Stash object or subclass that will take responsibility for managing template variables.

```
my $stash = MyOrg::Template::Stash->new({ ... });

my $tt = Template->new({
    STASH => $stash,
});
```

If unspecified, a default stash object is created using the VARIABLES configuration item to initialize the stash variables. These may also be specified as the PRE_DEFINE option for backward compatibility with Version 1.

```
my $tt = Template->new({
    VARIABLES => {
        id    => 'abw',
        name  => 'Andy Wardley',
    },
};
```

STASH is used by Template::Context.

A.1.40 START_TAG, END_TAG

The START_TAG and END_TAG options are used to specify character sequences or regular expressions that mark the start and end of a template directive. The default values for START_TAG and END_TAG are [% and %], respectively, giving us the familiar directive style:

```
[% example %]
```

Any Perl regex characters can be used and therefore should be escaped (or use the Perl quotemeta function) if they are intended to represent literal characters:

```
my $tt = Template->new({
    START_TAG => quotemeta('<+'),
    END_TAG   => quotemeta('+>'),
});
```

For example:

```
<+ INCLUDE foobar +>
```

The TAGS directive can also be used to set the START_TAG and END_TAG values on a per-template file basis:

```
[% TAGS <+ +> %]
```

The *ttree* equivalents for these options are --start_tag=STRING and --end_tag=STRING.

START_TAG and END_TAG are used by Template::Parser.

A.1.41 TAG_STYLE

The TAG_STYLE option can be used to set both START_TAG and END_TAG according to predefined tag styles.

```
my $tt = Template->new({
    TAG_STYLE => 'star',
});
```

Available styles are as follows:

```
template    [% ... %]               (default)
template1   [% ... %] or %% ... %%  (TT version 1)
metatext    %% ... %%               (Text::MetaText)
star        [* ... *]               (TT alternate)
php         <? ... ?>               (PHP)
asp         <% ... %>               (ASP)
mason       <% ...  >               (HTML::Mason)
html        <!-- ... -->            (HTML comments)
```

Any values specified for START_TAG and/or END_TAG will override those defined by a TAG_STYLE.

The TAGS directive may also be used to set a TAG_STYLE:

```
[% TAGS html %]
<!-- INCLUDE header -->
```

The *ttree* equivalent for this option is --tag_style=STRING.

TAG_STYLE is used by Template::Parser.

A.1.42 TOLERANT

The TOLERANT flag is used by the various Template Toolkit provider modules (Template::Provider, Template::Plugins, Template::Filters) to control their behavior when errors are encountered. By default, any errors are reported as such, with the request for the particular resource (template, plugin, filter) being denied and an exception raised. When the TOLERANT flag is set to any true values, errors will be silently ignored and the provider will instead return STATUS_DECLINED. This allows a subsequent provider to take responsibility for providing the resource, rather than failing the request outright. If all providers decline to service the request, either through tolerated failure or a genuine disinclination to comply, a <resource> not found exception is raised.

TOLERANT is used by Template::Provider, Template::Plugins, and Template::Filters.

A.1.43 TRIM

The TRIM option can be set to have any leading and trailing whitespace automatically removed from the output of all template files and BLOCKs. The possible values, CHOMP_ALL, CHOMP_COLLAPSE, and CHOMP_NONE, are available from Template::Constants:

```
use Template::Constants qw( :chomp );

my $tt = Template->new(TRIM => CHOMP_ALL);
```

The TRIM option is disabled (CHOMP_NONE) by default.

The *ttree* equivalent for this option is --trim.

TRIM is used by Template::Context.

A.1.44 VARIABLES, PRE_DEFINE

VARIABLES is a synonym for PRE_DEFINE.

A.1.45 V1DOLLAR

In Version 1 of the Template Toolkit, an optional leading $ could be placed on any template variable and would be silently ignored:

```
# VERSION 1

[% $foo %]       = = =  [% foo %]

[% $hash.$key %] = = =  [% hash.key %]
```

To interpolate a variable value, the ${' ... `} construct was used. Typically, one would do this to index into a hash array when the key value was stored in a variable.

For example:

```
my $vars = {

    users => {

        aba => { name => 'Alan Aardvark', ... },

        abw => { name => 'Andy Wardley', ... },

        ...

    },

    uid => 'aba',

    ...

};


$template->process('user/home.html', $vars)

    || die $template->error( ), "\n";
```

This is what goes in *user/home.html*:

```
[% user = users.${uid} %]     # users.aba

Name: [% user.name %]         # Alan Aardvark
```

This was inconsistent with double-quoted strings and also the INTERPOLATE mode, where a leading $ in text was enough to indicate a variable for interpolation, and the additional curly braces were used to delimit variable names where necessary. Note that this use is consistent with Unix and Perl conventions, among others.

```
# double quoted string interpolation

[% name = "$title ${user.name}" %]



# INTERPOLATE = 1

<img src="$images/help.gif"></a>

<img src="$images/${icon.next}.gif">
```

For Version 2, these inconsistencies have been removed and the syntax clarified. A leading $ on a variable is now used exclusively to indicate that the variable name should be interpolated (e.g., subsituted for its value) before being used. The earlier example from Version 1:

```
# VERSION 1

[% user = users.${uid} %]

Name: [% user.name %]
```

can now be simplified in Version 2 as:

```
# VERSION 2

[% user = users.$uid %]

Name: [% user.name %]
```

The leading $ is no longer ignored and has the same effect of interpolation as ${' ... '} in Version 1. The curly braces may still be used to explicitly scope the interpolated variable name where necessary. For example:

```
[% user = users.${me.id} %]

Name: [% user.name %]
```

The rule applies for all variables, both within directives and in plain text if processed with the INTERPOLATE option. This means that you should no longer (if you ever did) add a leading $ to a variable inside a directive, unless you explicitly want it to be interpolated.

One obvious side-effect is that any Version 1 templates with variables using a leading $ will no longer be processed as expected. Given the following variable definitions:

```
[% foo = 'bar'

   bar = 'baz'

%]
```

Version 1 would interpret them as:

```
# VERSION 1

[% $foo %] => [% GET foo %] => bar
```

whereas Version 2 interprets it as:

```
# VERSION 2

[% $foo %] => [% GET $foo %] => [% GET bar %] => baz
```

In Version 1, the $ is ignored and the value for the variable foo is retrieved and printed. In Version 2, the variable $foo is first interpolated to give the variable name bar, whose value is then retrieved and printed.

The use of the optional $ has never been strongly recommended, but to assist in backward compatibility with any Version 1 templates that may rely on this "feature," the V1DOLLAR option can be set to 1 (default: 0) to revert the behavior and have leading $ characters ignored.

```
my $tt = Template->new({

    V1DOLLAR => 1,

});
```

V1DOLLAR is used by Template::Parser.

A.2 Apache::Template Configuration Options

Most of the Apache::Template configuration directives relate directly to their Template Toolkit counterparts, differing only in having a TT2 prefix, mixed capitalization, and lack of underscores to space individual words. This is to make sure Apache::Template configuration directives keep with the preferred Apache/mod_perl style. For example:

```
Apache::Template  =>  Template Toolkit

--------------------------------------

TT2Trim             TRIM

TT2IncludePath      INCLUDE_PATH

TT2PostProcess      POST_PROCESS

...etc...
```

In some cases, the configuration directives are named or behave slightly differently to optimize for the Apache/mod_perl environment or domain-specific features. For example, the TT2Tags configuration directive can be used to set TAG_STYLE and/or START_TAG and END_TAG, and as such is more akin to the Template Toolkit TAGS directive. For example:

```
TT2Tags       html

TT2Tags       <!--  -->
```

See Section 12.3.1 in Chapter 12 for more details about configuring Apache::Template.

A.2.1 TT2Tags

This is used to set the tags used to indicate Template Toolkit directives within source templates. A single value can be specified to indicate a TAG_STYLE:

```
TT2Tags       html
```

A pair of values can be used to indicate a START_TAG and END_TAG:

```
TT2Tags       <!--    -->
```

Note that, unlike the Template Toolkit START_TAG and END_TAG configuration options, these values are automatically escaped to remove any special meaning within regular expressions:

```
TT2Tags       [*  *]      # no need to escape [ or *
```

By default, the start and end tags are set to [% and %], respectively. Thus, directives are embedded in the form [% INCLUDE my/file %].

A.2.2 TT2PreChomp

This is equivalent to the PRE_CHOMP configuration item. This flag can be set to remove any whitespace preceding a directive, up to and including the preceding newline. Default is Off.

```
TT2PreChomp    On
```

A.2.3 TT2PostChomp

This is equivalent to the POST_CHOMP configuration item. This flag can be set to automatically remove any whitespace after a directive, up to and including the following newline. Default is Off.

```
TT2PostChomp    On
```

A.2.4 TT2Trim

TT2Trim is equivalent to the TRIM configuration item. This flag can be set to have all surrounding whitespace stripped from template output. Default is Off.

```
TT2Trim        On
```

A.2.5 TT2AnyCase

This is equivalent to the ANY_CASE configuration item. This flag can be set to allow directive keywords to be specified in any case. By default, this setting is Off, and all directives (e.g., INCLUDE, FOREACH, etc.) should be specified in uppercase only.

```
TT2AnyCase      On
```

A.2.6 TT2Interpolate

TT2Interpolate is equivalent to the INTERPOLATE configuration item. This flag can be set to allow simple variables of the form $var to be embedded within templates, outside of regular directives. By default, this setting is Off, and variables must appear in the form [% var %], or more explicitly, [% GET var %].

```
TT2Interpolate  On
```

A.2.7 TT2IncludePath

This is equivalent to the INCLUDE_PATH configuration item, and can be used to specify one or more directories in which templates are located. Multiple directories may appear on each TT2IncludePath directive line, and the directive may be repeated. Directories are searched in the order defined.

```
TT2IncludePath  /usr/local/tt2/templates

TT2InludePath   /home/abw/tt2   /tmp/tt2
```

Note that this affects only templates that are processed via directives such as INCLUDE, PROCESS, INSERT, WRAPPER, etc. The full path of the main template processed by the Apache/mod_perl handler is generated (by Apache) by appending the request URI to the DocumentRoot, as per usual. For example, consider the following configuration extract:

```
DocumentRoot    /usr/local/web/ttdocs

[...]

TT2IncludePath  /usr/local/tt2/templates



<Files *.tt2>

    SetHandler      perl-script

    PerlHandler     Apache::Template

</Files>
```

A request with a URI of */foo/bar.tt2* will cause the handler to process the file */usr/local/web/ttdocs/foo/bar.tt2* (i.e., DocumentRoot + URI). If that file should include a directive such as [% INCLUDE foo/bar.tt2 %], that template should exist as the file */usr/local/tt2/templates/foo/bar.tt2* (i.e., TT2IncludePath + template name).

A.2.8 TT2Absolute

TT2Absolute is equivalent to the ABSOLUTE configuration item. This flag can be enabled to allow templates to be processed (via INCLUDE, PROCESS, etc.) that are specified with absolute filenames.

```
TT2Absolute     On
```

With the flag enabled, a template directive of the form:

```
[% INSERT /var/log/maillog %]
```

will be honored. The default setting is Off, and any attempt to load a template by absolute filename will result in a file exception being thrown with a message indicating that the ABSOLUTE option is not set. See the Template(1) manpage for further discussion on exception handling.

A.2.9 TT2Relative

This is equivalent to the RELATIVE configuration item, and is similar to the TT2Absolute option, but relates to files specified with a relative filename—that is, starting with *./* or *../*.

```
TT2Relative On
```

Enabling the option permits templates to be specifed as per this example:

```
[% INCLUDE ../../../etc/passwd %]
```

As with TT2Absolute, this option is set Off, causing a file exception to be thrown if used in this way.

A.2.10 TT2Delimiter

TT2Delimiter is equivalent to the DELIMTER configuration item, and can be set to define an alternate delimiter for separating multiple TT2IncludePath options. By default, it is set to :, and thus multiple directories can be specified as:

```
TT2IncludePath  /here:/there
```

Note that Apache implicitly supports space-delimited options, so the following is also valid and defines three directories, */here*, */there*, and */anywhere*:

```
TT2IncludePath  /here:/there /anywhere
```

If you're unfortunate enough to be running Apache on a Win32 system and you need to specify a : in a pathname, set the TT2Delimiter to an alternate value to avoid confusing the Template Toolkit into thinking you're specifying more than one directory:

```
TT2Delimiter    ,

TT2IncludePath  C:/HERE D:/THERE E:/ANYWHERE
```

A.2.11 TT2PreProcess

This is equivalent to PRE_PROCESS. This option allows one or more templates to be named that should be processed before the main template. This can be used to process a global configuration file, add canned headers, etc. These templates should be located in one of the TT2IncludePath directories, or specified absolutely if the TT2Absolute option is set.

```
TT2PreProcess   config header
```

A.2.12 TT2PostProcess

This is equivalent to POST_PROCESS. This option allows one or more templates to be named that should be processed after the main template—e.g., to add standard footers. As per TTPreProcess, these should be located in one of the TT2IncludePath directories, or specified absolutely if the TT2Absolute option is set.

```
TT2PostProcess  copyright footer
```

A.2.13 TT2Process

This is equivalent to the PROCESS configuration item. It can be used to specify one or more templates to be processed instead of the main template. This can be used to apply a standard "wrapper" around all template files processed by the handler.

```
TT2Process      mainpage
```

The original template (i.e., whose path is formed from the DocumentRoot + URI, as explained in the TT2IncludePath item earlier) is preloaded and available as the template variable. A typical TT2Process template might look like this:

```
[% PROCESS header %]

[% PROCESS $template %]

[% PROCESS footer %]
```

Note the use of the leading $ on template to defeat the auto-quoting mechanism that is applied to directives such as INCLUDE, PROCESS, etc. The directive would otherwise by interpreted as:

```
[% PROCESS "template" %]
```

A.2.14 TT2Default

TT2Default is equivalent to the DEFAULT configuration item. This can be used to name a template to be used in place of a missing template specified in a directive such as INCLUDE, PROCESS, INSERT, etc. Note that if the main template is not found (i.e., that which is mapped from the URI), the handler will decline the request, resulting in a 404 - Not Found. The template specified should exist in one of the directories named by TT2IncludePath.

```
TT2Default      nonsuch
```

A.2.15 TT2Error

This is equivalent to the ERROR configuration item. It can be used to name a template to be used to report errors that are otherwise uncaught. The template specified should exist in one of the directories named by TT2IncludePath. When the error template is processed, the error variable will be set to contain the relevant error details.

```
TT2Error        error
```

A.2.16 TT2EvalPerl

This is equivalent to the EVAL_PERL configuration item. It can be enabled to allow embedded [% PERL %] ... [% END %] sections within templates. It is disabled by default, and any PERL sections encountered will raise Perl exceptions with the message EVAL_PERL not set.

```
TT2EvalPerl     On
```

A.2.17 TT2LoadPerl

This is equivalent to the LOAD_PERL configuration item, which allows regular Perl modules to be loaded as Template Toolkit plugins via the USE directive. It is set Off by default.

```
TT2LoadPerl     On
```

A.2.18 TT2Recursion

This is equivalent to the RECURSION option, which allows templates to recurse into themselves either directly or indirectly. It is set Off by default.

```
TT2Recursion    On
```

A.2.19 TT2PluginBase

This is equivalent to the PLUGIN_BASE option. It allows multiple Perl packages to be specified that effectively form a search path for loading Template Toolkit plugins. The default value is Template::Plugin.

```
TT2PluginBase   My::Plugins  Your::Plugins
```

A.2.20 TT2AutoReset

TT2AutoReset is equivalent to the AUTO_RESET option and is enabled by default. It causes any template BLOCK definitions to be cleared before each main template is processed.

```
TT2AutoReset    Off
```

A.2.21 TT2CacheSize

This is equivalent to the CACHE_SIZE option. It can be used to limit the number of compiled templates that are cached in memory. The default value is undefined and all compiled templates will be cached in memory. It can be set to a specified numerical value to define the maximum number of templates, or set to 0 to disable caching altogether.

```
TT2CacheSize    64
```

A.2.22 TT2CompileExt

This is equivalent to the COMPILE_EXT option. It can be used to specify a filename extension that the Template Toolkit will use for writing compiled templates back to disk, thus providing cache persistence.

```
TT2CompileExt   .ttc
```

A.2.23 TT2CompileDir

TT2CompileDir is equivalent to the COMPILE_DIR option. It can be used to specify a root directory under which compiled templates should be written back to disk for cache persistence. Any *TT2IncludePath* directories will be replicated in full under this root directory.

```
TT2CompileDir   /var/tt2/cache
```

A.2.24 TT2Debug

This is equivalent to the DEBUG option, which enables Template Toolkit debugging. The main effect is to raise additional warnings when undefined variables are used, but it is likely to be expanded in a future release to provide more extensive debugging capabilities.

```
TT2Debug        On
```

A.2.25 TT2Headers

This allows you to specify which HTTP headers you want added to the response. Current permitted values are: modified (Last-Modified), length (Content-Length), etag (E-Tag) or all (all of the above).

```
TT2Headers      all
```

A.2.26 TT2Params

TT2Params allows you to specify which parameters you want defined as template variables. Current permitted values are uri, env (hash of environment variables), params (hash of CGI parameters), pnotes (the request pnotes hash), cookies (hash of cookies), uploads (a list of Apache::Upload instances), or all (all of the above).

```
TT2Params       uri env params uploads
```

When set, these values can then be accessed from within any template processed:

```
The URI is [% uri %]


Server name is [% env.SERVER_NAME %]


CGI params are:
<table>
[% FOREACH key = params.keys %]
   <tr>
     <td>[% key %]</td>  <td>[% params.$key %]</td>
   </tr>
[% END %]
</table>
```

A.2.27 TT2ServiceModule

The modules have been designed in such a way as to make it easy to subclass the Template::Service::Apache module to create your own custom services.

For example, the regular service module does a simple 1:1 mapping of URI to template using the requested filename provided by Apache, but you might want to implement an alternative scheme. You might prefer, for example, to map multiple URIs to the same template file, but to set some different template variables along the way.

To do this, you can subclass Template::Service::Apache and redefine the appropriate methods. The template( ) method performs the task of mapping URIs to templates, and the params( ) method sets up the template variable parameters. Or if you need to modify the HTTP headers, headers( ) is the one for you.

The TT2ServiceModule option can be set to indicate the name of your custom service module. The following trivial example shows how you might subclass Template::Service::Apache to add an additional parameter, in this case as the template variable message:

```
<perl>

package My::Service::Module;

use base qw( Template::Service::Apache );



sub params {

    my $self = shift;

    my $params = $self->SUPER::params(@_);

    $params->{ message } = 'Hello World';

    return $params;

}

</perl>


PerlModule          Apache::Template

TT2ServiceModule    My::Service::Module
```

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *Perl Template Toolkit* is a badger. The badger (*Meles meles*) is the largest member of the weasel family, and is found extensively throughout the northern hemisphere. Badgers are the best diggers of all carnivores, and can be found in the sandy or clay soils of dry open fields, parklands, and pastures where there are few large tree roots in their way as they digs.

Badgers are completely covered in gray or black fur except for on the head, where white stripes (or badges) run from the nose to the shoulders. Adult males can weigh as much as 26 pounds in autumn as, to prepare for winter, badgers tend to consume large amounts of food. Although they do not hibernate, badgers sleep in their burrows during winter and live off of their body fat.

The bones and muscles are large for an animal of the badger's size. The forefeet are armed with long, wide claws for digging. The claws on the hind legs are short and shovel-like for scooping away dirt. The flattened body easily slips into small burrows. A badger can dig itself into a hole in a few minutes.

Badgers are nocturnal, foraging for food at night. They eat everything from earthworms, insects, fruits, and berries to squirrels, mice, rabbits, and snakes. If attacked by a person or coyote--its main enemies--the badger acts quickly. The badger digs itself into a hole, throwing dirt and dust into its attacker's face. The badger turns with its powerful claws and terrible bite to face its enemy. The badger then starts to fill the hole in front of it with loose dirt to hide itself. Coyotes usually leave to find less dangerous prey. Few other animals will attack a badger.

Often hunted for their pelts, many countries now have laws protecting badgers. Badgers have been known to live for up to 14 years in the wild, but are likely to die or be killed before they reach this age.

Darren Kelly was the production editor, Audrey Doyle was the copyeditor, and Mary Brady was the proofreader for *Perl Template Toolkit*. Mary Anne Weeks Mayo and Colleen Gorman provided quality control. Tom Dinse wrote the index. Jamie Peppard, Matt Hutchinson, and Mary Agner provided production assistance.

Emma Colby designed the cover of this book, based on a series design by Edie Freedman. The cover image is an original engraving from the 19th century. Emma produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

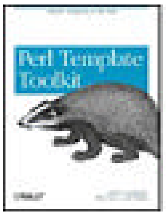
David Futato designed the interior layout. This book was converted by Joe Wizda to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Darren Kelly.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Ellie Cutler) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, Ellie Cutler, and Jeff Liggett.

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [**N**] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]



Perl Template Toolkit

By Darren Chamberlain, David Cross, Andy Wardley

Publisher: O'Reilly

Pub Date: December 2003

ISBN: 0-596-00476-1

Pages: 576

Slots: 1.0

Written by core members of the technology's development team, *Perl Template Toolkit* guides you through the entire process of installing, configuring, using, and extending the Template Toolkit. It begins with a fast-paced but thorough tutorial on building web content with the Template Toolkit, and then walks you through generating and using data files, particularly with XML. It also provides detailed information on the Template Toolkit's modules, libraries, and tools in addition to a complete reference manual.

A.1 Template Toolkit Configuration Options

The options listed here can be used from a Perl program as part of the configuration hash that is passed to the Template->new( ) method. In many cases, an equivalent option is available for *ttree* users. In those cases, the *ttree* version is mentioned in the description. Finally, each option identifies the Template Toolkit module that is the primary consumer of that option.

A.1.1 ABSOLUTE

The ABSOLUTE flag is used to indicate whether templates specified with absolute filenames (e.g., */foo/bar*) should be processed. It is disabled by default, and any attempt to load a template by such a name will cause a file exception to be raised.

```
my $tt = Template->new({

   ABSOLUTE => 1,

});



# this is why it's disabled by default

[% INSERT /etc/passwd %]
```

On Win32 systems, the regular expression for matching absolute pathnames is tweaked slightly to also detect filenames that start with a drive letter and colon, such as:

```
C:/Foo/Bar
```

The *ttree* equivalent of this option is --absolute.

ABSOLUTE is used by Template::Provider.

A.1.2 ANYCASE

By default, directive keywords should be expressed in uppercase. The ANYCASE option can be set to allow directive keywords to be specified in any case.

```
# ANYCASE => 0 (default)

[% INCLUDE foobar %]       # OK

[% include foobar %]       # ERROR

[% include = 10   %]       # OK, 'include' is a variable



# ANYCASE => 1

[% INCLUDE foobar %]       # OK

[% include foobar %]       # OK

[% include = 10   %]       # ERROR, 'include' is reserved word
```

One side effect of enabling ANYCASE is that you cannot use a variable of the same name as a reserved word, regardless of case. The reserved words are currently as follows:

```
GET CALL SET DEFAULT INSERT INCLUDE PROCESS WRAPPER

IF UNLESS ELSE ELSIF FOR FOREACH WHILE SWITCH CASE

USE PLUGIN FILTER MACRO PERL RAWPERL BLOCK META

TRY THROW CATCH FINAL NEXT LAST BREAK RETURN STOP

CLEAR TO STEP AND OR NOT MOD DIV END
```

The only lowercase reserved words that cannot be used for variables, regardless of the ANYCASE option, are these operators:

```
and or not mod div
```

The *ttree* equivalent of this option is --anycase.

ANYCASE is used by Template::Parser.

A.1.3 AUTO_RESET

The AUTO_RESET option is set by default and causes the local BLOCKS cache for the Template::Context object to be reset on each call to the Template process( ) method. This ensures that any BLOCKs defined within a template will persist only until that template is finished processing. This prevents BLOCKs defined in one processing request from interfering with other independent requests subsequently processed by the same context object.

The BLOCKS item may be used to specify a default set of block definitions for the Template::Context object. Subsequent BLOCK definitions in templates will override these but they will be reinstated on each reset if AUTO_RESET is enabled (default), or if the Template::Context reset( ) method is called.

AUTO_RESET is used by Template::Service.

A.1.4 BLOCKS

The BLOCKS option can be used to predefine a default set of template blocks. These should be specified as a reference to a hash array mapping template names to template text, subroutines, or Template::Document objects.

```
my $tt = Template->new({

   BLOCKS => {

      header  => 'The Header.  [% title %]',

      footer  => sub { return $some_output_text },

      another => Template::Document->new({ ... }),

   },

});
```

BLOCKS is used by Template::Context.

A.1.5 CACHE_SIZE

The Template::Provider module caches compiled templates to avoid the need to re-parse template files or blocks each time they are used. The CACHE_SIZE option is used to limit the number of compiled templates that the module should cache.

By default, the CACHE_SIZE option is undefined and all compiled templates are cached. When set to any positive value, the cache will be limited to storing no more than that number of compiled templates. When a new template is loaded and compiled and the cache is full (i.e., the number of entries = = CACHE_SIZE), the least recently used compiled template is discarded to make room for the new one.

CACHE_SIZE can be set to 0 to disable caching altogether:

```
my $tt = Template->new({

   CACHE_SIZE => 64,   # only cache 64 compiled templates

});



my $tt = Template->new({

   CACHE_SIZE => 0,   # don't cache any compiled templates

});
```

CACHE_SIZE is used by Template::Provider.

A.1.6 COMPILE_EXT

From Version 2 onward, the Template Toolkit has the ability to compile templates to Perl code and save them to disk for subsequent use (i.e., cache persistence). The COMPILE_EXT option may be provided to specify a filename extension for compiled template files. It is undefined by default and no attempt will be made to read or write any compiled template files.

```
my $tt = Template->new({
    COMPILE_EXT => '.ttc',
});
```

If COMPILE_EXT is defined (and COMPILE_DIR, covered next, isn't) compiled template files with the *COMPILE_EXT* extension will be written to the same directory from which the source template files were loaded.

Compiling and subsequent reuse of templates happens automatically whenever the COMPILE_EXT or COMPILE_DIR options are set. The Template Toolkit will automatically reload and reuse compiled files when it finds them on disk. If the corresponding source file has been modified since the compiled version was written, it will load and recompile the source and write a new compiled version to disk.

This form of cache persistence offers significant benefits in terms of time and resources required to reload templates. Compiled templates can be reloaded by a simple call to Perl's `require( )`, leaving Perl to handle all the parsing and compilation. This is a Good Thing.

The *ttree* equivalent of this option is `--compile_ext`.

A.1.7 COMPILE_DIR

The COMPILE_DIR option is used to specify an alternate directory root under which compiled template files should be saved:

```
my $tt = Template->new({
    COMPILE_DIR => '/tmp/ttc',
});
```

The COMPILE_EXT option may also be specified to have a consistent file extension added to these files:

```
my $tt1 = Template->new({
    COMPILE_DIR => '/tmp/ttc',
    COMPILE_EXT => '.ttc1',
});


my $tt2 = Template->new({
    COMPILE_DIR => '/tmp/ttc',
    COMPILE_EXT => '.ttc2',
});
```

When COMPILE_EXT is undefined, the compiled template files have the same name as the original template files, but reside in a different directory tree.

Each directory in INCLUDE_PATH is replicated in full beneath the COMPILE_DIR directory. This example:

```
my $tt = Template->new({
    COMPILE_DIR  => '/tmp/ttc',
    INCLUDE_PATH => '/home/abw/templates:/usr/share/templates',
});
```

would create the following directory structure:

```
/tmp/ttc/home/abw/templates/
/tmp/ttc/usr/share/templates/
```

Files loaded from different INCLUDE_PATH directories will have their compiled forms saved in the relevant COMPILE_DIR directory.

On Win32 platforms, a filename may by prefixed by a drive letter and colon. For example:

```
C:/My Templates/header
```

The colon will be silently stripped from the filename when it is added to the COMPILE_DIR value(s) to prevent illegal filenames being generated. Any colon in COMPILE_DIR elements will be left intact. For example:

```
# Win32 only

my $tt = Template->new({

    DELIMITER    => ';',

    COMPILE_DIR  => 'C:/TT2/Cache',

    INCLUDE_PATH => 'C:/TT2/Templates;D:/My Templates',

});
```

This would create the following cache directories:

```
C:/TT2/Cache/C/TT2/Templates

C:/TT2/Cache/D/My Templates
```

The *ttree* equivalent of this option is --compile_ext=STRING.

COMPILE_EXT and COMPILE_DIR are used by Template::Provider.

A.1.8 CONSTANTS

The CONSTANTS option can be used to specify a hash array of template variables that are compile-time constants. These variables are resolved once when the template is compiled, and thus don't require further resolution at runtime. This results in significantly faster processing of the compiled templates, and can be used for variables that don't change from one request to the next.

```
my $tt = Template->new({

    CONSTANTS => {

        title   => 'A Demo Page',

        author  => 'Joe Random Hacker',

        version => 3.14,

    },

};
```

CONSTANTS is used by Template.

A.1.9 CONSTANT_NAMESPACE

Constant variables are accessed via the constants namespace by default:

```
[% constants.title %]
```

The CONSTANTS_NAMESPACE option can be set to specify an alternate namespace:

```
my $tt = Template->new({

    CONSTANTS => {

        title   => 'A Demo Page',

        # ...etc...
```

```
    },
    CONSTANTS_NAMESPACE => 'const',
};
```

In this case, the constants would then be accessed as:

```
[% const.title %]
```

CONSTANTS_NAMESPACE is used by Template.

A.1.10 NAMESPACE

The constant-folding mechanism just described is an example of a namespace handler. Namespace handlers can be defined to provide alternate parsing mechanisms for variables in different namespaces.

Under the hood, the Template module converts a constructor configuration such as:

```
my $tt = Template->new({
    CONSTANTS => {
        title   => 'A Demo Page',
        # ...etc...
    },
    CONSTANTS_NAMESPACE => 'const',
};
```

into one like:

```
my $tt = Template->new({
    NAMESPACE => {
        const => Template:::Namespace::Constants->new({
            title   => 'A Demo Page',
            # ...etc...
        }),
    },
};
```

You can use this mechanism to define multiple constant namespaces, or to install custom handlers of your own.

```
my $tt = Template->new({
    NAMESPACE => {
        site => Template:::Namespace::Constants->new({
            title   => "Wardley's Widgets",
            version => 2.718,
        }),
        author => Template:::Namespace::Constants->new({
            name  => 'Andy Wardley',
            email => 'abw@andywardley.com',
        }),
        voodoo => My::Namespace::Handler->new( ... ),
```

```
    },
};
```

Now you have two constant namespaces, for example:

```
[% site.title %]
[% author.name %]
```

You also have your own custom namespace handler installed for the voodoo namespace.

```
[% voodoo.magic %]
```

NAMESPACE is used by Template::Directive and Template::Parser.

A.1.11 CONTEXT

A reference to a Template::Context object is used to define a specific environment in which templates are processed. A Template::Context object is passed as the only parameter to the Perl subroutines that represent "compiled" template documents. Template subroutines make callbacks into the context object to access Template Toolkit functionality—for example, to INCLUDE or PROCESS another template (include( ) and process( ) methods, respectively), to USE a plugin (plugin( )) or instantiate a filter (filter( )) or to access the stash (stash( )) that manages variable definitions via the get( ) and set( ) methods.

```
my $tt = Template->new({
    CONTEXT => MyOrg::Template::Context->new({ ... }),
});
```

CONTEXT is used by Template::Service.

A.1.12 DEBUG

The DEBUG option can be used to enable debugging within the various different modules that comprise the Template Toolkit. The TemplateConstants module defines a set of DEBUG_XXXX constants that can be combined using the logical OR operator (|).

```
use Template::Constants qw( :debug );


my $tt = Template->new({
    DEBUG => DEBUG_PARSER | DEBUG_PROVIDER,
});
```

For convenience, you can also provide a string containing a list of lowercase debug options, separated by any nonword characters:

```
my $tt = Template->new({
    DEBUG => 'parser, provider',
});
```

The following DEBUG_XXXX flags can be used:

DEBUG_SERVICE

Enables general debugging messages for the TemplateService module.

DEBUG_CONTEXT

Enables general debugging messages for the TemplateContext module.

DEBUG_PROVIDER

Enables general debugging messages for the TemplateProvider module.

DEBUG_PLUGINS

Enables general debugging messages for the TemplatePlugins module.

DEBUG_FILTERS

Enables general debugging messages for the TemplateFilters module.

DEBUG_PARSER

Causes the TemplateParser to generate debugging messages that show the Perl code generated by parsing and compiling each template.

DEBUG_UNDEF

Causes the Template Toolkit to throw an undef error whenever it encounters an undefined variable value.

DEBUG_DIRS

Causes the Template Toolkit to generate comments indicating the source file, line, and original text of each directive in the template. These comments are embedded in the template output using the format defined in the DEBUG_FORMAT configuration item, or a simple default format if unspecified.

For example, the following template fragment:

```
Hello World
```

would generate this output:

```
## input text line 1 :  ##

Hello

## input text line 2 : World ##

World
```

DEBUG_ALL

Enables all debugging messages.

DEBUG_CALLER

Causes all debug messages that aren't newline-terminated to have the filename and line number of the caller appended to them.

A.1.13 DEBUG_FORMAT

The DEBUG_FORMAT option can be used to specify a format string for the debugging messages generated via the DEBUG_DIRS option described earlier. Any occurrences of $file, $line, or $text will be replaced with the current filename, line, or directive text, respectively. Notice how the format is single-quoted to prevent Perl from interpolating those tokens as variables:

```
my $tt = Template->new({

   DEBUG => 'dirs',

   DEBUG_FORMAT => '<!-- $file line $line : [% $text %] -->',

});
```

The following template fragment:

```
[% foo = 'World' %]

Hello [% foo %]
```

would then generate this output:

```
<!-- input text line 2 : [% foo = 'World' %] -->

Hello <!-- input text line 3 : [% foo %] -->World
```

The DEBUG directive can also be used to set a debug format within a template:

```
[% DEBUG format '<!-- $file line $line : [% $text %] -->' %]
```

The *ttree* equivalent of this option is --debug (or -dbg).

DEBUG_FORMAT is used by Template::Context.

A.1.14 DEFAULT

The DEFAULT option can be used to specify a default template that should be used whenever a specified template can't be found in INCLUDE_PATH:

```
my $tt = Template->new({

   DEFAULT => 'notfound.html',

});
```

If a nonexistent template is requested through the Template process( ) method or by an INCLUDE, PROCESS, or WRAPPER directive, the DEFAULT template will instead be processed, if defined. Note that the DEFAULT template is not used when templates are specified with absolute or relative filenames, or as a reference to an input filehandle or text string.

The ttree equivalent of this option is --default=TEMPLATE.

DEFAULT is used by Template::Provider.

A.1.15 DELIMITER

This is used to provide an alternative delimiter character sequence for separating paths specified in INCLUDE_PATH. The default value for DELIMITER is :.

```
my $tt = Template->new({

   DELIMITER    => '; ',

   INCLUDE_PATH => 'C:/HERE/NOW; D:/THERE/THEN',

});
```

On Win32 systems, the default delimiter is a little more intelligent, splitting paths only on : characters that aren't followed by a /. This means that the following should work as planned, splitting INCLUDE_PATH into two separate directories, *C:/foo* and *C:/bar*:

```
# on Win32 only

my $tt = Template->new({

   INCLUDE_PATH => 'C:/Foo:C:/Bar'

});
```

However, if you're using Win32, it's recommended that you explicitly set the DELIMITER character to something else (e.g., ;) rather than rely on this subtle magic.

DELIMITER is used by Template::Service and Template::Provider.

A.1.16 ERROR

The ERROR (or ERRORS if you prefer) configuration item can be used to name a single template or specify a hash array mapping exception types to templates that should be used for error handling. If an uncaught exception is raised from within a template, the appropriate error template will instead be processed.

If specified as a single value, that template will be processed for all uncaught exceptions:

```
my $tt = Template->new({
    ERROR => 'error.html'
});
```

If the ERROR item is a hash reference, the keys are assumed to be exception types and the relevant template for a given exception will be selected. A "default" template may be provided for the general case. Note that ERROR can be pluralized to ERRORS if you find it more appropriate in this case.

```
my $tt = Template->new({
    ERRORS => {
        user     => 'user/index.html',
        dbi      => 'error/database',
        default  => 'error/default',
    },
});
```

In this example, any user exceptions thrown will cause the *user/index.html* template to be processed. dbi errors are handled by *error/database* and all others by the *error/default* template. Any *PRE_PROCESS* and/or *POST_PROCESS* templates will also be applied to these error templates.

Note that exception types are hierarchical, and a foo handler will catch all foo.* errors (e.g., foo.bar, foo.bar.baz) if a more specific handler isn't defined. Be sure to quote any exception types that contain periods to prevent Perl from concatenating them into a single string (i.e., user.passwd is parsed as 'user'.'passwd').

```
my $tt = Template->new({
    ERROR => {
        'user.login'  => 'user/login.html',
        'user.passwd' => 'user/badpasswd.html',
        'user'        => 'user/index.html',
        'default'     => 'error/default',
    },
});
```

In this example, any template processed by the $tt object, other templates, or code called from within can raise a user.login exception and have the service redirect to the *user/login.html* template. Similarly, a user.passwd exception has a specific handling template, *user/badpasswd.html*, while all other user or user.* exceptions cause a redirection to the *user/index.html* page. All other exception types are handled by *error/default*.

Exceptions can be raised in a template using the THROW directive:

```
[% THROW user.login 'no user id: please login' %]
```

or by calling the throw( ) method on the current Template::Context object:

```
$context->throw('user.passwd', 'Incorrect Password');
$context->throw('Incorrect Password');    # type 'undef'
```

or from Perl code by calling die( ) with a Template::Exception object:

```
die (Template::Exception->new('user.denied', 'Invalid User ID'));
```

or by simply calling die( ) with an error string. This is automatically caught and converted to an exception of undef type, which can then be handled in the usual way:

```
die "I'm sorry Dave, I can't do that";
```

The *ttree* equivalent for this option is --error=TEMPLATE.

ERROR is used by Template::Service.

A.1.17 EVAL_PERL

This flag is used to indicate whether PERL and/or RAWPERL blocks should be evaluated. By default, it is disabled, and any PERL or RAWPERL blocks encountered will raise exceptions of type perl with the message EVAL_PERL not set. Note, however, that any RAWPERL blocks should always contain valid Perl code, regardless of the EVAL_PERL flag. The parser will fail to compile templates that contain invalid Perl code in RAWPERL blocks, and will throw a file exception.

If EVAL_PERL is set when a template is compiled, all PERL and RAWPERL blocks will be included in the compiled template. If EVAL_PERL isn't set, Perl code will be generated, which *always* throws a perl exception with the message EVAL_PERL not set *whenever* the compiled template code is run.

Thus, you must have EVAL_PERL set if you want your compiled templates to include PERL and RAWPERL blocks.

At some point in the future, using a different invocation of the Template Toolkit, you may come to process such a precompiled template. Assuming the EVAL_PERL option was set at the time the template was compiled, the output of any RAWPERL blocks will be included in the compiled template and will get executed when the template is processed. This will happen regardless of the runtime EVAL_PERL status.

Regular PERL blocks are a little more cautious, however. If the EVAL_PERL flag isn't set for the *current* context—that is, the one that is trying to process it—it will throw the familiar perl exception with the message EVAL_PERL not set.

Thus you can compile templates to include PERL blocks, but optionally disable them when you process them later. Note, however, that it is possible for a PERL block to contain a Perl BEGIN { # some code } block that is always get run regardless of the runtime EVAL_PERL status. Thus, if you set EVAL_PERL when compiling templates, it is assumed that you trust the templates to Do The Right Thing. Otherwise, you must accept the fact that there's no bulletproof way to prevent any included code from trampling around in the living room of the runtime environment, making a real nuisance of itself if it really wants to. If you don't like the idea of such uninvited guests causing a bother, you can accept the default and keep EVAL_PERL disabled.

The *ttree* equivalent of this option is --eval_perl.

EVAL_PERL is used by Template::Directive, Template::Context, and Template::Filters.

A.1.18 FACTORY

FACTORY defines the class used by Template::Parser to generate Perl code for elements of the grammar, which defaults to Template::Directive.

FACTORY is used by Template::Parser.

A.1.19 FILTERS

The FILTERS option can be used to specify custom filters that can then be used with the FILTER directive like any other. These are added to the standard filters, which are available by default. Filters specified via this option will mask any standard filters of the same name.

The FILTERS option should be specified as a reference to a hash array in which each key represents the name of a filter. The corresponding value should contain a reference to an array containing a subroutine reference and a flag that indicates whether the filter is static (0) or dynamic (1). A filter may also be specified as a solitary subroutine reference and is assumed to be static.

```
$tt = Template->new({
    FILTERS => {
        'sfilt1' =>   \&static_filter,      # static
        'sfilt2' => [ \&static_filter, 0 ], # same as above
        'dfilt1' => [ \&dynamic_filter_factory, 1 ],
    },
});
```

Additional filters can be specified at any time by calling the define_filter( ) method on the current Template::Context object. The method accepts a filter name, a reference to a filter subroutine, and an optional flag to indicate whether the filter is dynamic.

```
my $context = $template->context( );
$context->define_filter('new_html', \&new_html);
$context->define_filter('new_repeat', \&new_repeat, 1);
```

In static filters, a single subroutine reference is used for all invocations of a particular filter. Filters that don't accept any configuration parameters (e.g., html) can be implemented statically. The subroutine reference is simply returned when that particular filter is requested. The subroutine is called to filter the output of a template block that is passed as the only argument. The subroutine should return the modified text.

```
sub static_filter {

    my $text = shift;

    # do something to modify $text...

    return $text;

}
```

The following template fragment:

```
[% FILTER sfilt1 %]

Blah blah blah.

[% END %]
```

is approximately equivalent to:

```
&static_filter("\nBlah blah blah.\n");
```

Filters that can accept parameters (e.g., truncate) should be implemented dynamically. In this case, the subroutine is taken to be a filter factory that is called to create a unique filter subroutine each time one is requested. A reference to the current Template::Context object is passed as the first parameter, followed by any additional parameters specified. The subroutine should return another subroutine reference (usually a closure) that implements the filter.

```
sub dynamic_filter_factory {

    my ($context, @args) = @_;


    return sub {

        my $text = shift;

        # do something to modify $text...

        return $text;

    }

}
```

The following template fragment:

```
[% FILTER dfilt1(123, 456) %]

Blah blah blah

[% END %]
```

is approximately equivalent to:

```
my $filter = &dynamic_filter_factory($context, 123, 456);

&$filter("\nBlah blah blah.\n");
```

FILTERS is used by Template::Context.

A.1.20 GRAMMAR

The GRAMMAR configuration item can be used to specify an alternate grammar for the parser. This allows a modified or entirely new template language to be constructed and used by the Template Toolkit.

Source templates are compiled to Perl code by the Template::Parser using the Template::Grammar (by default) to define the language structure and semantics. Compiled templates are thus inherently "compatible" with each other, and there is nothing to prevent any number of different template languages from being compiled and used within the same Template Toolkit processing environment (other than the usual time and memory constraints).

The Template::Grammar file is constructed from a YACC-like grammar (using Parse::YAPP) and a skeleton module template. These files are provided, along with a small script to rebuild the grammar, in the *parser* subdirectory of the distribution. You don't have to know or worry about these unless you want to hack on the template language or define your own variant. A README file in the same directory provides some small guidance, but it is assumed that you know what you're doing if you venture herein. If you grok LALR parsers, then you should find it comfortably familiar.

By default, an instance of the default Template::Grammar will be created and used automatically if a GRAMMAR item isn't specified:

```
use MyOrg::Template::Grammar;


my $tt = Template->new({
    GRAMMAR = MyOrg::Template::Grammar->new( );
});
```

GRAMMAR is used by Template::Parser.

A.1.21 INCLUDE_PATH

INCLUDE_PATH is used to specify one or more directories in which template files are located. When a template is requested that isn't defined locally as a BLOCK, each INCLUDE_PATH directory is searched in turn to locate the template file. Multiple directories can be specified as a reference to a list or as a single string where each directory is delimited by :.

```
my $tt = Template->new({
    INCLUDE_PATH => '/usr/local/templates',
});


my $tt = Template->new({
    INCLUDE_PATH => '/usr/local/templates:/tmp/my/templates',
});


my $tt = Template->new({
    INCLUDE_PATH => [ '/usr/local/templates',
                      '/tmp/my/templates' ],
});
```

On Win32 systems, a little extra magic is invoked, ignoring delimiters that have : followed by a / or \. This avoids confusion when using directory names such as *C:\Blah Blah*.

When specified as a list, the INCLUDE_PATH path can contain elements that dynamically generate a list of INCLUDE_PATH directories. These generator elements can be specified as a reference to a subroutine or an object that implements a paths( ) method.

```
my $tt = Template->new({
    INCLUDE_PATH => [ '/usr/local/templates',
                      \&incpath_generator,
                      My::IncPath::Generator->new( ... ) ],
});
```

Each time a template is requested and the INCLUDE_PATH examined, the subroutine or object method will be called. A reference to a list of directories should be returned. Generator subroutines should report errors using die( ). A generator object should return undef and make an error available via its error( ) method.

For example:

```
sub incpath_generator {

    # ...some code...

    if ($all_is_well) {
        return \@list_of_directories;
    }
    else {
        die "cannot generate INCLUDE_PATH...\n";
    }
}
```

or:

```
package My::IncPath::Generator;

# Template::Base (or Class::Base) provides error( ) method
use Template::Base;
use base qw( Template::Base );

sub paths {
    my $self = shift;

    # ...some code...

    if ($all_is_well) {
        return \@list_of_directories;
    }
    else {
        return $self->error("cannot generate INCLUDE_PATH...\n");
    }
}

1;
```

The *ttree* equivalent of this option is --lib=DIR (or -l DIR).

INCLUDE_PATH is used by Template::Provider.

A.1.22 INTERPOLATE

The INTERPOLATE flag, when set to any true value, will cause variable references in plain text (i.e., not surrounded by START_TAG and END_TAG) to be recognized and interpolated accordingly:

```
my $tt = Template->new({
    INTERPOLATE => 1,
});
```

Variables should be prefixed by a $ to identify them. Curly braces can be used in the familiar Perl/shell style to explicitly scope the variable name where required.

```
# INTERPOLATE => 0
<a href="http://[% server %]/[% help %]">
<img src="[% images %]/help.gif"></a>
[% myorg.name %]


# INTERPOLATE => 1
<a href="http://$server/$help">
<img src="$images/help.gif"></a>
$myorg.name


# explicit scoping with {  }
<img src="$images/${icon.next}.gif">
```

Note that a limitation in Perl's regex engine restricts the maximum length of an interpolated template to around 32 kilobytes or possibly less. Files that exceed this limit in size will typically cause Perl to dump core with a segmentation fault. If you routinely process templates of this size, you should disable INTERPOLATE or split the templates in several smaller files or blocks that can then be joined backed together via PROCESS or INCLUDE.

The *ttree* equivalent for this option is --interpolate.

INTERPOLATE is used by Template::Parser.

A.1.23 LOAD_FILTERS

The LOAD_FILTERS option can be used to specify a list of provider objects (i.e., they implement the fetch( ) method) that are responsible for returning and/or creating filter subroutines. The Template::Context filter( ) method queries each provider in turn in a "Chain of Responsibility" as per the template( ) and plugin( ) methods.

```
my $tt = Template->new({
    LOAD_FILTERS => [
        MyTemplate::Filters->new( ),
        Template::Filters->new( ),
    ],
});
```

By default, a single Template::Filters object is created for the LOAD_FILTERS list.

LOAD_FILTERS is used by Template::Context.

A.1.24 LOAD_PERL

If a plugin cannot be loaded using the PLUGINS or PLUGIN_BASE approaches, the provider can make a final attempt to load the module without prepending any prefix to the module path. This allows regular Perl modules (i.e., those that don't reside in Template::Plugin or some other such namespace) to be loaded and used as plugins.

By default, the LOAD_PERL option is set to 0 and no attempt will be made to load any Perl modules that aren't named explicitly in the PLUGINS hash or that don't reside in a package as named by one of the PLUGIN_BASE components.

Plugins loaded using the PLUGINS or PLUGIN_BASE receive a reference to the current context object as the first argument to the new( ) constructor. Modules loaded using LOAD_PERL are assumed to not conform to the plugin interface. They must provide a new( ) class method for instantiating objects, which will not receive a reference to the context as the first argument. Plugin modules should provide a load( ) class method (or inherit the default one from the Template::Plugin base class) that is called the first time the plugin is loaded. Regular Perl modules need not provide a load( ) method. In all other respects, regular Perl objects and Template Toolkit plugins are identical.

If a particular Perl module does not conform to the common, but not unilateral, new( ) constructor convention, a simple plugin wrapper can be written to interface to it.

The *ttree* equivalent of this option is --load_perl.

LOAD_PERL is used by Template::Plugins.

A.1.25 LOAD_PLUGINS

The LOAD_PLUGINS options can be used to specify a list of provider objects (i.e., they implement the fetch( ) method) that are responsible for loading and instantiating template plugin objects. The Template::Content plugin( ) method queries each provider in turn in a "Chain of Responsibility" as per the template( ) and filter( ) methods.

```
my $tt = Template->new({
   LOAD_PLUGINS => [
         MyOrg::Template::Plugins->new({ ... }),
         Template::Plugins->new({ ... }),
   ],
});
```

By default, a single Template::Plugins object is created using the current configuration hash. Configuration items destined for the Template::Plugins constructor may be added to the Template constructor.

```
my $tt = Template->new({
   PLUGIN_BASE => 'MyOrg::Template::Plugins',
   LOAD_PERL   => 1,
});
```

LOAD_PLUGINS is used by Template::Context.

A.1.26 LOAD_TEMPLATES

The LOAD_TEMPLATE option can be used to provide a reference to a list of Template::Provider objects or subclasses thereof that will take responsibility for loading and compiling templates.

```
my $tt = Template->new({
   LOAD_TEMPLATES => [
      MyOrg::Template::Provider->new({ ... }),
      Template::Provider->new({ ... }),
   ],
});
```

When a PROCESS, INCLUDE, or WRAPPER directive is encountered, the named template may refer to a locally defined BLOCK or a file relative to the INCLUDE_PATH (or an absolute or relative path if the appropriate ABSOLUTE or RELATIVE options are set). If a BLOCK definition can't be found (see Example 7-4 in the Section 7.3.5 for a discussion of BLOCK locality), each LOAD_TEMPLATES provider object is queried in turn via the fetch( ) method to see whether it can supply the required template. Each provider can return a compiled template or an error, or can decline to service the request, in which case the responsibility is passed to the next provider. If none of the providers can service the request, a not found error is returned. The same basic provider mechanism is also used for the INSERT directive, but it bypasses any BLOCK definitions and doesn't attempt to parse or process the contents of the template file.

This is an implementation of the "Chain of Responsibility" design pattern as described in *Design Patterns*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addision-Wesley).

If LOAD_TEMPLATES is undefined, a single default provider will be instantiated using the current configuration parameters. For example, the Template::Provider INCLUDE_PATH option can be specified in the Template configuration and will be correctly passed to the provider's constructor method:

```
my $tt = Template->new({
    INCLUDE_PATH => '/here:/there',
});
```

LOAD_TEMPLATES is used by Template::Context.

A.1.27 OUTPUT_PATH

OUTPUT_PATH allows a directory to be specified into which output files should be written. An output file can be specified by the OUTPUT option, or passed by name as the third parameter to the Template process( ) method.

```
my $template = Template->new({
    INCLUDE_PATH => "/tmp/src",
    OUTPUT_PATH  => "/tmp/dest",
});

my $vars = {
    ...
};

foreach my $file ('foo.html', 'bar.html') {
    $template->process($file, $vars, $file)
        || die $template->error( );
}
```

This example will read the input files */tmp/src/foo.html* and */tmp/src/bar.html*, and write the processed output to */tmp/dest/foo.html* and */tmp/dest/bar.htm*l, respectively.

The *ttree* equivalent of this option is --dest=DIR (or -d DIR).

OUTPUT_PATH is used by Template and Template::Filters.

A.1.28 OUTPUT

This is the default output location or handler. This may be specified as a filename (relative to OUTPUT_PATH, if defined, or the current working directory if not specified absolutely); a filehandle (e.g., GLOB or IO::Handle) opened for writing; a reference to a text string to that the output is appended (the string isn't cleared); a reference to a subroutine that is called, passing the output text as an argument; a reference to an array onto which the content will be push( )ed; or a reference to any object that supports the print( ) method. This latter option includes the Apache::Request object which is passed as the argument to Apache/mod_perl handlers (see Example A-1 through Example A-6).

Example A-1. Filename

```
my $tt = Template->new({
    OUTPUT => "/tmp/foo",
});
```

Example A-2. Text string

```
my $output = '';

my $tt = Template->new({
    OUTPUT => \$output,
});
```

Example A-3. Filehandle

```
open (TOUT, "> $file") || die "$file: $!\n";

my $tt = Template->new({
    OUTPUT => *TOUT,
});
```

Example A-4. Subroutine

```
sub output { my $out = shift; print "OUTPUT: $out" }

my $tt = Template->new({
    OUTPUT => \&output,
});
```

Example A-5. Array reference

```
my $tt = Template->new({
    OUTPUT => \@output,
})
```

Example A-6. Apache/mod_perl handler

```
sub handler {
    my $r = shift;

    my $tt = Template->new({
        OUTPUT => $r,
    });
    ...
}
```

The default OUTPUT location can be overridden by passing a third parameter to the Template process( ) method. This can be specified as any of the following argument types:

```
$tt->process($file, $vars, "/tmp/foo");
$tt->process($file, $vars, "bar");
$tt->process($file, $vars, *MYGLOB);
$tt->process($file, $vars, \@output);
$tt->process($file, $vars, $r);  # Apache::Request
...
```

OUTPUT is used by Template.

A.1.29 PARSER

The Template::Parser module implements a parser object for compiling templates into Perl code, which can then be executed. A default object of this class is created automatically and then used by Template::Provider whenever a template is loaded and requires compilation. The PARSER option can be used to provide a reference to an alternate parser object.

```
my $tt = Template->new({
    PARSER => MyOrg::Template::Parser->new({ ... }),
});
```

PARSER is used by Template::Provider.

A.1.30 PLUGIN_BASE

If a plugin is not defined in the PLUGINS hash, PLUGIN_BASE is used to attempt to construct a correct Perl module name that can be successfully loaded.

PLUGIN_BASE can be specified as a single value or as a reference to an array of multiple values. The default PLUGIN_BASE value, Template::Plugin, is always added to the end of the PLUGIN_BASE list (a single value is first converted to a list). Each value should contain a Perl package name to which the requested plugin name is appended. For example:

```
my $tt = Template->new({
    PLUGIN_BASE => 'MyOrg::Template::Plugin',
});


[% USE Foo %]    # => MyOrg::Template::Plugin::Foo
                    or       Template::Plugin::Foo
```

or:

```
my $tt = Template->new({
    PLUGIN_BASE => [   'MyOrg::Template::Plugin',
                       'YourOrg::Template::Plugin'  ],
});


[% USE Foo %]    # =>   MyOrg::Template::Plugin::Foo
                    or YourOrg::Template::Plugin::Foo
                    or        Template::Plugin::Foo
```

The *ttree* equivalent for this option is --plugin_base=PACKAGE.

PLUGIN_BASE is used by Template::Plugins.

A.1.31 PLUGINS

The PLUGINS option can be used to provide a reference to a hash array that maps plugin names to Perl module names. A number of standard plugins are defined (e.g., table, cgi, dbi, etc.) that map to their corresponding Template::Plugin::* counterparts. These can be redefined by values in the PLUGINS hash:

```
my $tt = Template->new({
    PLUGINS => {
        cgi => 'MyOrg::Template::Plugin::CGI',
        foo => 'MyOrg::Template::Plugin::Foo',
        bar => 'MyOrg::Template::Plugin::Bar',
    },
});
```

The USE directive is used to create plugin objects and does so by calling the plugin( ) method on the current Template::Context object. If the plugin name is defined in the PLUGINS hash, the corresponding Perl module is loaded via require( ). The context then calls the load( ) class method, which should return the class name (default and general case) or a prototype object against which the new( ) method can be called to instantiate individual plugin objects.

If the plugin name is not defined in the PLUGINS hash, the PLUGIN_BASE and/or LOAD_PERL options come into effect.

PLUGINS is used by Template::Plugins.

A.1.32 PRE_CHOMP, POST_CHOMP

Anything outside a directive tag is considered plain text and is generally passed through unaltered (but see the INTERPOLATE option for text that's altered as it is passed through). This includes all whitespace and newline characters surrounding directive tags. Directives that don't generate any output will leave gaps in the output document.

For example, this:

```
Foo
[% a = 10 %]
Bar
```

will output this:

```
Foo


Bar
```

The PRE_CHOMP and POST_CHOMP options can help to clean up some of this extraneous whitespace. Both are disabled by default.

```
my $tt = Template->new({
    PRE_CHOMP  => 1,
    POST_CHOMP => 1,
});
```

With PRE_CHOMP set to 1, the newline and whitespace preceding a directive at the start of a line will be deleted. This has the effect of concatenating a line that starts with a directive onto the end of the previous line.

```
        Foo <----------.
                       |
    ,---(PRE_CHOMP)----'
    |
    `-- [% a = 10 %] --.
                       |
```

```
  ,---(POST_CHOMP)---'

  |

  `-> Bar
```

With POST_CHOMP set to 1, any whitespace after a directive up to and including the newline will be deleted. This has the effect of joining a line that ends with a directive onto the start of the next line.

If PRE_CHOMP or POST_CHOMP is set to 2, instead of removing all the whitespace, the whitespace will be collapsed to a single space. This is useful for HTML, where (usually) a contiguous block of whitespace is rendered the same as a single space.

You may use the CHOMP_NONE, CHOMP_ALL, and CHOMP_COLLAPSE constants from the Template::Constants module to deactivate chomping, remove all whitespace, or collapse whitespace to a single space.

PRE_CHOMP and POST_CHOMP can be activated for individual directives by placing a dash (-) immediately at the start and/or end of the directive:

```
[% FOREACH user = userlist %]

  [%- user -%]

[% END %]
```

The - character activates both PRE_CHOMP and POST_CHOMP for the one directive [%- name -%]. Thus, the template will be processed as if written:

```
[% FOREACH user = userlist %][% user %][% END %]
```

Note that this is the same as if PRE_CHOMP and POST_CHOMP were set to CHOMP_ALL; the only way to get the CHOMP_COLLAPSE behavior is to set PRE_CHOMP or POST_CHOMP accordingly. If PRE_CHOMP or POST_CHOMP is already set to CHOMP_COLLAPSE, using - will give you CHOMP_COLLAPSE behavior, not CHOMP_ALL behavior.

Similarly, + characters can be used to disable PRE_CHOMP or POST_CHOMP (i.e., leave the whitespace/newline intact) options on a per-directive basis:

```
[% FOREACH user = userlist %]

User: [% user +%]

[% END %]
```

With POST_CHOMP enabled, the previous example would be parsed as if written:

```
[% FOREACH user = userlist %]User: [% user %]

[% END %]
```

The *ttree* equivalents of these options are --pre_chomp and --post_chomp.

PRE_CHOMP and POST_CHOMP are used by Template::Parser.

A.1.33 PRE_DEFINE, VARIABLES

The PRE_DEFINE option (or VARIABLES; they're equivalent) can be used to specify a hash array of template variables that should be used to preinitialize the stash when it is created. These items are ignored if the STASH item is defined:

```
my $tt = Template->new({

  VARIABLES => {

    title   => 'A Demo Page',

    author  => 'Joe Random Hacker',

    version => 3.14,

  },

};
```

or:

```
my $tt = Template->new({
    PRE_DEFINE => {
        title   => 'A Demo Page',
        author  => 'Joe Random Hacker',
        version => 3.14,
    },
};
```

The *ttree* equivalent of this option is --define var=value.

PRE_DEFINE is used by Template::Context.

A.1.34 PRE_PROCESS, POST_PROCESS

These values may be set to contain the name(s) of template files (relative to INCLUDE_PATH) that should be processed immediately before and/or after each template. These do not get added to templates processed into a document via directives such as INCLUDE, PROCESS, WRAPPER, etc.

```
my $tt = Template->new({
    PRE_PROCESS  => 'header',
    POST_PROCESS => 'footer',
};


$tt->process('mydoc.html')
    || die $tt->error( );
```

Multiple templates may be specified as a reference to a list. Each is processed in the order defined.

```
my $tt = Template->new({
    PRE_PROCESS  => [ 'config', 'header' ],
    POST_PROCESS => 'footer',
};
```

Alternately, multiple templates may be specified as a single string, delimited by the : character. This delimiter string can be changed via the DELIMITER option.

```
my $tt = Template->new({
    PRE_PROCESS  => 'config:header',
    POST_PROCESS => 'footer',
};
```

The PRE_PROCESS and POST_PROCESS templates are evaluated in the same variable context as the main document and may define or update variables for subsequent use.

The Template::Document object representing the main template being processed is available within PRE_PROCESS and POST_PROCESS templates as the template variable. Metadata items defined via the META directive may be accessed accordingly.

Example A-7 through Example A-10 show the *config*, *header*, *footer*, and *mydoc.html* files.

Example A-7. config

```
[% # set some site-wide variables
   bgcolor = '#ffffff'
   version = 2.718
%]
```

Example A-8. header

```
[% DEFAULT title = 'My Funky Web Site' %]
<html>
<head>
<title>[% title %]</title>
</head>
<body bgcolor="[% bgcolor %]">
```

Example A-9. footer

```
<hr />
Version [% version %]
</body>
</html>
```

Example A-10. mydoc.html

```
[% META title = 'My Document Title' %]
blah blah blah
...
```

The *ttree* equivalents for these options are --pre_process=TEMPLATE and --post_process=TEMPLATE.

PRE_PROCESS and POST_PROCESS are used by Template::Service.

A.1.35 PROCESS

The PROCESS option may be set to contain the name(s) of template files (relative to INCLUDE_PATH) that should be processed instead of the main template passed to the Template process( ) method. This can be used to apply consistent wrappers around all templates, similar to the use of PRE_PROCESS and POST_PROCESS templates.

```
my $tt = Template->new({
    PROCESS  => 'content',
};

# processes 'content' instead of 'foo.html'
$tt->process('foo.html');
```

A reference to the original template is available in the template variable. Metadata items can be inspected and the template can be processed by specifying it as a variable reference (i.e., prefixed by $) to an INCLUDE, PROCESS, or WRAPPER directive.

Example A-11, Example A-12, and Example A-13 show the *content*, *foo.html*, and *output* files.

Example A-11. content

```
<html>
<head>
<title>[% template.title %]</title>
</head>

<body>
[% PROCESS $template %]
<hr />
&copy; Copyright [% template.copyright %]
</body>
</html>
```

Example A-12. foo.html

```
[% META
   title     = 'The Foo Page'
   author    = 'Fred Foo'
   copyright = '2000 Fred Foo'
%]
<h1>[% template.title %]</h1>
Welcome to the Foo Page, blah blah blah
```

Example A-13. output

```
<html>
<head>
<title>The Foo Page</title>
</head>

<body>
<h1>The Foo Page</h1>
Welcome to the Foo Page, blah blah blah
<hr />
&copy; Copyright 2000 Fred Foo
</body>
</html>
```

The *ttree* equivalent of this option is --process=TEMPLATE.

PROCESS is used by Template::Service.

A.1.36 RECURSION

The template processor will raise a file exception if it detects direct or indirect recursion into a template. Setting this option to any true value will allow templates to include each other recursively.

The *ttree* equivalent of this option is --recursion.

RECURSION is used by Template::Context and Template::Document.

A.1.37 RELATIVE

The RELATIVE flag is used to indicate whether templates specified with filenames relative to the current directory (e.g., *./foo/bar* or *../../some/where/else*) should be loaded. It is also disabled by default, and will raise a file error if such template names are encountered.

```
my $tt = Template->new({

    RELATIVE => 1,

});



[% INCLUDE ../logs/error.log %]
```

The *ttree* equivalent of this option is --relative.

RELATIVE is used by Template::Provider.

A.1.38 SERVICE

This provides a reference to a Template::Service object, or subclass thereof, to which the Template module should delegate. If unspecified, a Template::Service object is automatically created using the current configuration hash.

```
my $tt = Template->new({

    SERVICE => MyOrg::Template::Service->new({ ... }),

});
```

SERVICE is used by Template.

A.1.39 STASH

This provides a reference to a Template::Stash object or subclass that will take responsibility for managing template variables.

```
my $stash = MyOrg::Template::Stash->new({ ... });

my $tt = Template->new({

    STASH => $stash,

});
```

If unspecified, a default stash object is created using the VARIABLES configuration item to initialize the stash variables. These may also be specified as the PRE_DEFINE option for backward compatibility with Version 1.

```
my $tt = Template->new({

    VARIABLES => {

        id    => 'abw',

        name  => 'Andy Wardley',

    },

};
```

STASH is used by Template::Context.

A.1.40 START_TAG, END_TAG

The START_TAG and END_TAG options are used to specify character sequences or regular expressions that mark the start and end of a template directive. The default values for START_TAG and END_TAG are [% and %], respectively, giving us the familiar directive style:

```
[% example %]
```

Any Perl regex characters can be used and therefore should be escaped (or use the Perl quotemeta function) if they are intended to represent literal characters:

```
my $tt = Template->new({
    START_TAG => quotemeta('<+'),
    END_TAG   => quotemeta('+>'),
});
```

For example:

```
<+ INCLUDE foobar +>
```

The TAGS directive can also be used to set the START_TAG and END_TAG values on a per-template file basis:

```
[% TAGS <+ +> %]
```

The *ttree* equivalents for these options are --start_tag=STRING and --end_tag=STRING.

START_TAG and END_TAG are used by Template::Parser.

A.1.41 TAG_STYLE

The TAG_STYLE option can be used to set both START_TAG and END_TAG according to predefined tag styles.

```
my $tt = Template->new({
    TAG_STYLE => 'star',
});
```

Available styles are as follows:

```
template    [% ... %]               (default)
template1   [% ... %] or %% ... %%  (TT version 1)
metatext    %% ... %%               (Text::MetaText)
star        [* ... *]               (TT alternate)
php         <? ... ?>               (PHP)
asp         <% ... %>               (ASP)
mason       <% ...  >               (HTML::Mason)
html        <!-- ... -->            (HTML comments)
```

Any values specified for START_TAG and/or END_TAG will override those defined by a TAG_STYLE.

The TAGS directive may also be used to set a TAG_STYLE:

```
[% TAGS html %]
<!-- INCLUDE header -->
```

The *ttree* equivalent for this option is --tag_style=STRING.

TAG_STYLE is used by Template::Parser.

A.1.42 TOLERANT

The TOLERANT flag is used by the various Template Toolkit provider modules (Template::Provider, Template::Plugins, Template::Filters) to control their behavior when errors are encountered. By default, any errors are reported as such, with the request for the particular resource (template, plugin, filter) being denied and an exception raised. When the TOLERANT flag is set to any true values, errors will be silently ignored and the provider will instead return STATUS_DECLINED. This allows a subsequent provider to take responsibility for providing the resource, rather than failing the request outright. If all providers decline to service the request, either through tolerated failure or a genuine disinclination to comply, a <resource> not found exception is raised.

TOLERANT is used by Template::Provider, Template::Plugins, and Template::Filters.

A.1.43 TRIM

The TRIM option can be set to have any leading and trailing whitespace automatically removed from the output of all template files and BLOCKs. The possible values, CHOMP_ALL, CHOMP_COLLAPSE, and CHOMP_NONE, are available from Template::Constants:

```
use Template::Constants qw( :chomp );

my $tt = Template->new(TRIM => CHOMP_ALL);
```

The TRIM option is disabled (CHOMP_NONE) by default.

The *ttree* equivalent for this option is --trim.

TRIM is used by Template::Context.

A.1.44 VARIABLES, PRE_DEFINE

VARIABLES is a synonym for PRE_DEFINE.

A.1.45 V1DOLLAR

In Version 1 of the Template Toolkit, an optional leading $ could be placed on any template variable and would be silently ignored:

```
# VERSION 1

[% $foo %]       = = =  [% foo %]

[% $hash.$key %] = = =  [% hash.key %]
```

To interpolate a variable value, the ${' ... `} construct was used. Typically, one would do this to index into a hash array when the key value was stored in a variable.

For example:

```
my $vars = {

    users => {

        aba => { name => 'Alan Aardvark', ... },

        abw => { name => 'Andy Wardley', ... },

        ...

    },

    uid => 'aba',

    ...

};


$template->process('user/home.html', $vars)

    || die $template->error( ), "\n";
```

This is what goes in *user/home.html*:

```
[% user = users.${uid} %]     # users.aba

Name: [% user.name %]         # Alan Aardvark
```

This was inconsistent with double-quoted strings and also the INTERPOLATE mode, where a leading $ in text was enough to indicate a variable for interpolation, and the additional curly braces were used to delimit variable names where necessary. Note that this use is consistent with Unix and Perl conventions, among others.

```
# double quoted string interpolation

[% name = "$title ${user.name}" %]



# INTERPOLATE = 1

<img src="$images/help.gif"></a>

<img src="$images/${icon.next}.gif">
```

For Version 2, these inconsistencies have been removed and the syntax clarified. A leading $ on a variable is now used exclusively to indicate that the variable name should be interpolated (e.g., subsituted for its value) before being used. The earlier example from Version 1:

```
# VERSION 1

[% user = users.${uid} %]

Name: [% user.name %]
```

can now be simplified in Version 2 as:

```
# VERSION 2

[% user = users.$uid %]

Name: [% user.name %]
```

The leading $ is no longer ignored and has the same effect of interpolation as ${' ... '} in Version 1. The curly braces may still be used to explicitly scope the interpolated variable name where necessary. For example:

```
[% user = users.${me.id} %]

Name: [% user.name %]
```

The rule applies for all variables, both within directives and in plain text if processed with the INTERPOLATE option. This means that you should no longer (if you ever did) add a leading $ to a variable inside a directive, unless you explicitly want it to be interpolated.

One obvious side-effect is that any Version 1 templates with variables using a leading $ will no longer be processed as expected. Given the following variable definitions:

```
[% foo = 'bar'

   bar = 'baz'

%]
```

Version 1 would interpret them as:

```
# VERSION 1

[% $foo %] => [% GET foo %] => bar
```

whereas Version 2 interprets it as:

```
# VERSION 2

[% $foo %] => [% GET $foo %] => [% GET bar %] => baz
```

In Version 1, the $ is ignored and the value for the variable foo is retrieved and printed. In Version 2, the variable $foo is first interpolated to give the variable name bar, whose value is then retrieved and printed.

The use of the optional $ has never been strongly recommended, but to assist in backward compatibility with any Version 1 templates that may rely on this "feature," the V1DOLLAR option can be set to 1 (default: 0) to revert the behavior and have leading $ characters ignored.

```
my $tt = Template->new({

    V1DOLLAR => 1,

});
```

V1DOLLAR is used by Template::Parser.

A.2 Apache::Template Configuration Options

Most of the Apache::Template configuration directives relate directly to their Template Toolkit counterparts, differing only in having a TT2 prefix, mixed capitalization, and lack of underscores to space individual words. This is to make sure Apache::Template configuration directives keep with the preferred Apache/mod_perl style. For example:

```
Apache::Template  =>  Template Toolkit

--------------------------------------

TT2Trim              TRIM

TT2IncludePath       INCLUDE_PATH

TT2PostProcess       POST_PROCESS

...etc...
```

In some cases, the configuration directives are named or behave slightly differently to optimize for the Apache/mod_perl environment or domain-specific features. For example, the TT2Tags configuration directive can be used to set TAG_STYLE and/or START_TAG and END_TAG, and as such is more akin to the Template Toolkit TAGS directive. For example:

```
TT2Tags      html

TT2Tags      <!--  -->
```

See Section 12.3.1 in Chapter 12 for more details about configuring Apache::Template.

A.2.1 TT2Tags

This is used to set the tags used to indicate Template Toolkit directives within source templates. A single value can be specified to indicate a TAG_STYLE:

```
TT2Tags      html
```

A pair of values can be used to indicate a START_TAG and END_TAG:

```
TT2Tags      <!--    -->
```

Note that, unlike the Template Toolkit START_TAG and END_TAG configuration options, these values are automatically escaped to remove any special meaning within regular expressions:

```
TT2Tags      [*  *]      # no need to escape [ or *
```

By default, the start and end tags are set to [% and %], respectively. Thus, directives are embedded in the form [% INCLUDE my/file %].

A.2.2 TT2PreChomp

This is equivalent to the PRE_CHOMP configuration item. This flag can be set to remove any whitespace preceding a directive, up to and including the preceding newline. Default is Off.

```
TT2PreChomp    On
```

A.2.3 TT2PostChomp

This is equivalent to the POST_CHOMP configuration item. This flag can be set to automatically remove any whitespace after a directive, up to and including the following newline. Default is Off.

```
TT2PostChomp    On
```

A.2.4 TT2Trim

TT2Trim is equivalent to the TRIM configuration item. This flag can be set to have all surrounding whitespace stripped from template output. Default is Off.

```
TT2Trim        On
```

A.2.5 TT2AnyCase

This is equivalent to the ANY_CASE configuration item. This flag can be set to allow directive keywords to be specified in any case. By default, this setting is Off, and all directives (e.g., INCLUDE, FOREACH, etc.) should be specified in uppercase only.

```
TT2AnyCase      On
```

A.2.6 TT2Interpolate

TT2Interpolate is equivalent to the INTERPOLATE configuration item. This flag can be set to allow simple variables of the form $var to be embedded within templates, outside of regular directives. By default, this setting is Off, and variables must appear in the form [% var %], or more explicitly, [% GET var %].

```
TT2Interpolate  On
```

A.2.7 TT2IncludePath

This is equivalent to the INCLUDE_PATH configuration item, and can be used to specify one or more directories in which templates are located. Multiple directories may appear on each TT2IncludePath directive line, and the directive may be repeated. Directories are searched in the order defined.

```
TT2IncludePath  /usr/local/tt2/templates

TT2InludePath   /home/abw/tt2   /tmp/tt2
```

Note that this affects only templates that are processed via directives such as INCLUDE, PROCESS, INSERT, WRAPPER, etc. The full path of the main template processed by the Apache/mod_perl handler is generated (by Apache) by appending the request URI to the DocumentRoot, as per usual. For example, consider the following configuration extract:

```
DocumentRoot    /usr/local/web/ttdocs

[...]

TT2IncludePath  /usr/local/tt2/templates



<Files *.tt2>

    SetHandler      perl-script

    PerlHandler     Apache::Template

</Files>
```

A request with a URI of */foo/bar.tt2* will cause the handler to process the file */usr/local/web/ttdocs/foo/bar.tt2* (i.e., DocumentRoot + URI). If that file should include a directive such as [% INCLUDE foo/bar.tt2 %], that template should exist as the file */usr/local/tt2/templates/foo/bar.tt2* (i.e., TT2IncludePath + template name).

A.2.8 TT2Absolute

TT2Absolute is equivalent to the ABSOLUTE configuration item. This flag can be enabled to allow templates to be processed (via INCLUDE, PROCESS, etc.) that are specified with absolute filenames.

```
TT2Absolute     On
```

With the flag enabled, a template directive of the form:

```
[% INSERT /var/log/maillog %]
```

will be honored. The default setting is Off, and any attempt to load a template by absolute filename will result in a file exception being thrown with a message indicating that the ABSOLUTE option is not set. See the Template(1) manpage for further discussion on exception handling.

A.2.9 TT2Relative

This is equivalent to the RELATIVE configuration item, and is similar to the TT2Absolute option, but relates to files specified with a relative filename—that is, starting with *./* or *../*.

```
TT2Relative On
```

Enabling the option permits templates to be specifed as per this example:

```
[% INCLUDE ../../../etc/passwd %]
```

As with TT2Absolute, this option is set Off, causing a file exception to be thrown if used in this way.

A.2.10 TT2Delimiter

TT2Delimiter is equivalent to the DELIMTER configuration item, and can be set to define an alternate delimiter for separating multiple TT2IncludePath options. By default, it is set to :, and thus multiple directories can be specified as:

```
TT2IncludePath  /here:/there
```

Note that Apache implicitly supports space-delimited options, so the following is also valid and defines three directories, */here*, */there*, and */anywhere*:

```
TT2IncludePath  /here:/there /anywhere
```

If you're unfortunate enough to be running Apache on a Win32 system and you need to specify a : in a pathname, set the TT2Delimiter to an alternate value to avoid confusing the Template Toolkit into thinking you're specifying more than one directory:

```
TT2Delimiter    ,

TT2IncludePath  C:/HERE D:/THERE E:/ANYWHERE
```

A.2.11 TT2PreProcess

This is equivalent to PRE_PROCESS. This option allows one or more templates to be named that should be processed before the main template. This can be used to process a global configuration file, add canned headers, etc. These templates should be located in one of the TT2IncludePath directories, or specified absolutely if the TT2Absolute option is set.

```
TT2PreProcess   config header
```

A.2.12 TT2PostProcess

This is equivalent to POST_PROCESS. This option allows one or more templates to be named that should be processed after the main template—e.g., to add standard footers. As per TTPreProcess, these should be located in one of the TT2IncludePath directories, or specified absolutely if the TT2Absolute option is set.

```
TT2PostProcess  copyright footer
```

A.2.13 TT2Process

This is equivalent to the PROCESS configuration item. It can be used to specify one or more templates to be processed instead of the main template. This can be used to apply a standard "wrapper" around all template files processed by the handler.

```
TT2Process      mainpage
```

The original template (i.e., whose path is formed from the DocumentRoot + URI, as explained in the TT2IncludePath item earlier) is preloaded and available as the template variable. A typical TT2Process template might look like this:

```
[% PROCESS header %]

[% PROCESS $template %]

[% PROCESS footer %]
```

Note the use of the leading $ on template to defeat the auto-quoting mechanism that is applied to directives such as INCLUDE, PROCESS, etc. The directive would otherwise by interpreted as:

```
[% PROCESS "template" %]
```

A.2.14 TT2Default

TT2Default is equivalent to the DEFAULT configuration item. This can be used to name a template to be used in place of a missing template specified in a directive such as INCLUDE, PROCESS, INSERT, etc. Note that if the main template is not found (i.e., that which is mapped from the URI), the handler will decline the request, resulting in a 404 - Not Found. The template specified should exist in one of the directories named by TT2IncludePath.

```
TT2Default      nonsuch
```

A.2.15 TT2Error

This is equivalent to the ERROR configuration item. It can be used to name a template to be used to report errors that are otherwise uncaught. The template specified should exist in one of the directories named by TT2IncludePath. When the error template is processed, the error variable will be set to contain the relevant error details.

```
TT2Error        error
```

A.2.16 TT2EvalPerl

This is equivalent to the EVAL_PERL configuration item. It can be enabled to allow embedded [% PERL %] ... [% END %] sections within templates. It is disabled by default, and any PERL sections encountered will raise Perl exceptions with the message EVAL_PERL not set.

```
TT2EvalPerl     On
```

A.2.17 TT2LoadPerl

This is equivalent to the LOAD_PERL configuration item, which allows regular Perl modules to be loaded as Template Toolkit plugins via the USE directive. It is set Off by default.

```
TT2LoadPerl     On
```

A.2.18 TT2Recursion

This is equivalent to the RECURSION option, which allows templates to recurse into themselves either directly or indirectly. It is set Off by default.

```
TT2Recursion    On
```

A.2.19 TT2PluginBase

This is equivalent to the PLUGIN_BASE option. It allows multiple Perl packages to be specified that effectively form a search path for loading Template Toolkit plugins. The default value is Template::Plugin.

```
TT2PluginBase   My::Plugins  Your::Plugins
```

A.2.20 TT2AutoReset

TT2AutoReset is equivalent to the AUTO_RESET option and is enabled by default. It causes any template BLOCK definitions to be cleared before each main template is processed.

```
TT2AutoReset    Off
```

A.2.21 TT2CacheSize

This is equivalent to the CACHE_SIZE option. It can be used to limit the number of compiled templates that are cached in memory. The default value is undefined and all compiled templates will be cached in memory. It can be set to a specified numerical value to define the maximum number of templates, or set to 0 to disable caching altogether.

```
TT2CacheSize    64
```

A.2.22 TT2CompileExt

This is equivalent to the COMPILE_EXT option. It can be used to specify a filename extension that the Template Toolkit will use for writing compiled templates back to disk, thus providing cache persistence.

```
TT2CompileExt   .ttc
```

A.2.23 TT2CompileDir

TT2CompileDir is equivalent to the COMPILE_DIR option. It can be used to specify a root directory under which compiled templates should be written back to disk for cache persistence. Any *TT2IncludePath* directories will be replicated in full under this root directory.

```
TT2CompileDir   /var/tt2/cache
```

A.2.24 TT2Debug

This is equivalent to the DEBUG option, which enables Template Toolkit debugging. The main effect is to raise additional warnings when undefined variables are used, but it is likely to be expanded in a future release to provide more extensive debugging capabilities.

```
TT2Debug        On
```

A.2.25 TT2Headers

This allows you to specify which HTTP headers you want added to the response. Current permitted values are: modified (Last-Modified), length (Content-Length), etag (E-Tag) or all (all of the above).

```
TT2Headers      all
```

A.2.26 TT2Params

TT2Params allows you to specify which parameters you want defined as template variables. Current permitted values are uri, env (hash of environment variables), params (hash of CGI parameters), pnotes (the request pnotes hash), cookies (hash of cookies), uploads (a list of Apache::Upload instances), or all (all of the above).

```
TT2Params       uri env params uploads
```

When set, these values can then be accessed from within any template processed:

```
The URI is [% uri %]


Server name is [% env.SERVER_NAME %]


CGI params are:

<table>

[% FOREACH key = params.keys %]

   <tr>

     <td>[% key %]</td>  <td>[% params.$key %]</td>

   </tr>

[% END %]

</table>
```

A.2.27 TT2ServiceModule

The modules have been designed in such a way as to make it easy to subclass the Template::Service::Apache module to create your own custom services.

For example, the regular service module does a simple 1:1 mapping of URI to template using the requested filename provided by Apache, but you might want to implement an alternative scheme. You might prefer, for example, to map multiple URIs to the same template file, but to set some different template variables along the way.

To do this, you can subclass Template::Service::Apache and redefine the appropriate methods. The template( ) method performs the task of mapping URIs to templates, and the params( ) method sets up the template variable parameters. Or if you need to modify the HTTP headers, headers( ) is the one for you.

The TT2ServiceModule option can be set to indicate the name of your custom service module. The following trivial example shows how you might subclass Template::Service::Apache to add an additional parameter, in this case as the template variable message:

```
<perl>

package My::Service::Module;

use base qw( Template::Service::Apache );



sub params {

    my $self = shift;

    my $params = $self->SUPER::params(@_);

    $params->{ message } = 'Hello World';

    return $params;

}

</perl>


PerlModule          Apache::Template

TT2ServiceModule    My::Service::Module
```

Appendix A. Appendix: Configuration Options

The Template Toolkit is extremely configurable, and mastery of the many options takes time and practice, and requires that you read a lot of documentation. This appendix will help with the third requirement, as it contains a complete list of the Template Toolkit configuration options.

1.1 What the Template Toolkit Does

The basic task of a template processor is to output some amount of changeable data surrounded by some unchanging data. A simple example of this is a form letter, where the same text is sent to many different people, with just the name, address, and other personal details being changed. The template contains the fixed ("boilerplate") text together with special markup tags indicating where the variable pieces of data are to be placed.

Example 1-1 shows a template for such a form letter. This template is marked up using the default style of the Template Toolkit, where the [% ... %] tags indicate variable values. Everything else is fixed text that passes through the processor untouched.

Example 1-1. A form letter template, destruction.tt

```
People of [% planet %], your attention please.


This is [% captain %] of the
Galactic Hyperspace Planning Council.


As you will no doubt be aware, the plans
for development of the outlying regions
of the Galaxy require the building of a
hyperspatial express route through your
star system, and regrettably your planet
is one of those scheduled for destruction.


The process will take slightly less than
[% time %].


Thank you.
```

A template processor takes the template, together with a list of the variable data to be included in the letter, and produces a finished letter. The Template Toolkit provides *tpage* for doing just that from the command line. Pass the name of the template file to *tpage* as a command-line option, along with any number of --define options to provide values for variables. If the preceding template is stored in the *destruction.tt* file in the current directory, the following command processes it:

```
$ tpage --define planet=Earth \
>       --define captain="Prostetnic Vogon Jeltz" \
>       --define time="two of your earth minutes" \
>       destruction.tt
```

The output this generates is shown in Example 1-2.

Example 1-2. Form letter generated by template in Example 1-1

```
People of Earth, your attention please.


This is Prostetnic Vogon Jeltz of the
```

```
Galactic Hyperspace Planning Council.

As you will no doubt be aware, the plans
for development of the outlying regions
of the Galaxy require the building of a
hyperspatial express route through your
star system, and regrettably your planet
is one of those scheduled for destruction.

The process will take slightly less than
two of your earth minutes.

Thank you.
```

Process the same template a few thousand times with different sets of data and you have the entire basis of the junk-mail industry. Or a Vogon Constructor Fleet.

This book is a good example of a more complex template. All O'Reilly books conform to one of a small number of formats. They all have similar sets of front matter (title page, publication information, table of contents, and preface), followed by the actual chapters, some (optional) appendices, an index, and finally the colophon. Templates that define the look of all of these parts are defined in the publication system, and the data for a particular book is formatted to conform to those rules. If someone decides to change the font used for the chapter titles in forthcoming books, he need only change the setting in the template definition.

Another way to look at a template processor is as a tool for separating *processing* from *presentation*. For example, a company sales report is probably created from data stored in a database. One way to create the report would be to extract the required data into a spreadsheet and then do calculations on the data to produce the information required. The spreadsheet could then be printed out or emailed to the required recipients.

Although you can use templates to generate any kind of text document, the most common use is to generate HTML pages for web content. The whole genre of template processing systems has matured rapidly in less than a decade, particularly within the Perl community, in response to the demands of people struggling to build and maintain ever more complex content and applications for their web sites.

Templates help in a number of ways. The most obvious benefit is that they can be used to apply a consistent look and feel to all the pages in a web site to achieve a common branding. You can use a template to add standard headers, footers, menus, and other user interface components as easily as the Hyperspace Planning Council ruthlessly adds a few lines of Vogon poetry to every planet destruction order, just to rub salt into the wounds.

This is just the tip of the iceberg. In addition to the use of variables, the Template Toolkit provides a number of other directives that instruct it to perform more complex processing actions, such as including another template, repeating a section of markup for different pieces of data, or choosing a section to process based on a particular condition. Example 1-3 illustrates some of these directives in action.

Example 1-3. Loops, conditions, and processing instructions in a template

```
[% FOREACH order IN council.demolition.orders %]
  [% PROCESS header %]

  [% IF order.destruction %]
    As you will no doubt be aware, the plans
    for development of the outlying regions
    of the Galaxy require the building of a
    hyperspatial express route through your
    star system, and regrettably your planet
```

```
    is one of those scheduled for destruction.
  [% ELSE %]
    Our representatives will be visiting your
    star system within the next few weeks,
    and would like to invite you to a reading of
    Vogon Poetry. Attendance is mandatory.
    Resistance is useless!
  [% END %]


  [% PROCESS footer %]


  [% PROCESS poetry/excerpt
       IF today.day = = 'Vogonsday'
  %]
[% END %]
```

We explain the purpose of these directives later in this chapter, and show examples of the different ways they can be used throughout the rest of the book. For now, you can probably work out what they do from their names.

The Template Toolkit is just one example of a template processor. Although it's written in Perl, you don't actually need to know any Perl to use it. The presentation language that it provides is intentionally simple, regular, and easy to understand and use. This makes it simple for web designers and other nonprogrammers to use it without first having to get to grips with Perl. The Template Toolkit provides language features and off-the-shelf plugin modules that allow you to perform many common tasks, including CGI programming, manipulating XML files, and accessing SQL databases.

If you do know Perl, however, you'll be able to get more out of the Template Toolkit by writing custom functions and extensions to handle the specifics of your particular application. The good news for Perl programmers is that the Template Toolkit allows you to separate Perl code clearly from HTML templates. This clear separation means that you don't have to wade through pages of HTML markup to find the part of your web application that needs attention. It allows you to concentrate on one thing at a time, be it the HTML presentation or the Perl application, without having the other aspects in your face and under your feet. It makes both your HTML templates and Perl code more portable and reusable, and easier to read, write, and maintain.

1.2 The Templating Ecosystem

At least half a dozen mature and respected templating systems are available for Perl. The best-known and best-supported template processors include the following:

Text::Template

Text::Template is a library for generating form letters, building HTML pages, or filling in templates generally. A template is a piece of text that has little Perl programs embedded in it here and there. When you fill in a template, you evaluate the little programs and replace them with their values. These programs are written in Perl: you embed Perl code in your template, with { at the beginning and } at the end. If you want a variable interpolated, you write it the way you would in Perl. If you need to make a loop, you can use any of the Perl loop constructions. All the Perl built-in functions are available.

Text::Template is available from http://www.plover.com/~mjd/perl/Template/ or from CPAN (http://search.cpan.org/dist/Text-Template/).

HTML::Template

HTML::Template attempts to make using HTML templates easy and natural. It extends standard HTML with a few HTML-like tags, and enforces the divide between design and programming by restricting what a template is capable of doing. By limiting the programmer to using just simple variables and loops in the HTML, the template remains accessible to designers and other non-Perl people. The use of HTML-like syntax goes further to make the format understandable to others.

HTML::Template is available from CPAN (http://search.cpan.org/dist/HTML-Template/).

HTML::Mason

HTML::Mason is a Perl-based web site development and delivery system. Mason allows web pages and sites to be constructed from shared, reusable building blocks called *components*. Components contain a mix of Perl and HTML, and can call each other and pass values back and forth like subroutines. Components increase modularity and eliminate repetitive work: common design elements (headers, footers, menus, logos) can be extracted into their own components where they need be changed only once to affect the whole site. Mason also includes powerful filtering and templating facilities and an HTML/data caching model.

HTML::Mason is available from http://www.masonhq.com/ and CPAN (http://search.cpan.org/dist/HTML-Mason/).

HTML::Embperl

Embperl gives you the power to embed Perl code in your HTML documents, and the ability to build your web site out of small reusable objects in an object-oriented style. You can also take advantage of all the usual Perl modules (including DBI for database access), use their functionality, and easily include their output in your web pages.

Embperl has several features that are especially useful for creating HTML, including dynamic tables, form field processing, URL escaping/unescaping, session handling, and more.

Embperl is a server-side tool, which means that it's browser-independent. It can run in various ways: under mod_perl, as a CGI script, or offline.

HTML::Embperl is available from http://www.ecos.de/ or CPAN (http://search.cpan.org/dist/HTML-Embperl/).

Apache::ASP

Apache::ASP provides an Active Server Pages port to the Apache web server with Perl scripting only, and enables development of dynamic web applications with session management and embedded Perl code. Apache::ASP also provides many powerful extensions, including XML taglibs, XSLT rendering, and new events not originally part of the ASP API.

Apache::ASP is available from CPAN (http://search.cpan.org/dist/Apache-ASP/).

The Template Toolkit attempts to offer the best features of these modules, including separation of Perl from templates and applicability beyond HTML.

1.2.1 The Template Toolkit Is for More Than HTML

The Template Toolkit is a generic template processing system that will process any kind of document for use in any environment or application. Many other template systems were designed specifically to create HTML pages for web content. In some cases, that is all the system can be used for. In others, it is possible (with varying degrees of difficulty) to use the system in a non-web environment.

The Template Toolkit was originally designed to help Andy create his web site, but he was careful to ensure that it was just as usable outside of that environment. As a result, there is nothing within the Template Toolkit that assumes it is being used to generate HTML. It is equally at home creating any other kind of data.

1.2.2 The Template Toolkit Lets You Choose Your Separation

Template Toolkit doesn't prescribe any particular methodology or framework that forces you to use it in a certain way. Some modules (for example, `HTML::Template`) enforce a very strict interpretation of template processing that intentionally limits what can be done in a template to accessing variables and using simple conditional or looping constructs. Others (such as `HTML::Mason` and `HTML::Embperl`) use embedded Perl code to allow any kind of application functionality to be incorporated directly into the templates.

The Template Toolkit gives you the best of both worlds. It has a powerful data engine (the *Stash*) that does all the hard work of mapping complex data structures from your Perl code, configuration files, SQL databases, XML files, and so on, into template variables that are accessed by a simple and uniform dotted notation (e.g., `person.surname`). You can use this to keep your templates simple without limiting the complexity or functionality of the systems that put data into the templates.

At the opposite end of the spectrum, the Template Toolkit also allows you to embed Perl code directly in your templates. We don't normally encourage this because it tends to defeat the purpose of having a template processing system in the first place. Because this is the exception rather than the norm, template processors must set the `EVAL_PERL` option to embed Perl code in the template (it is disabled by default). We look at how to set options later in this chapter.

Template Toolkit also lets you work between the two extremes. It provides a rich set of language features (*directives*) that allow you to add complex functionality to your templates without requiring you to embed Perl code. It also has a powerful *plugin* mechanism that allows you to load and use Perl modules to extend the functionality in any way you can imagine.

In short, the Template Toolkit allows you to take a modular approach to building your web site or other document system, but doesn't enforce it. Sometimes you want to build a complex and highly structured system to run a web site. Other times you just want to roll up a quick all-in-one template to generate a report from a database. The Template Toolkit encourages whatever approach is most appropriate to the task at hand.

1.2.3 Nonprogrammers Can Maintain Templates

Template Toolkit's template language is designed to be as simple as possible without being too simple. The dotted notation makes accessing variables far less daunting than in Perl. For example:

```
$person->{surname}   # Perl

person.surname       # Template Toolkit
```

This hides the underlying implementation details from the template designer. In the previous example, the Perl syntax implies that `$person` is a reference to a hash array containing a `surname` value. However, you might one day decide to implement `$person` as an object with a `surname( )` method:

```
$person->surname( )   # Perl

person.surname        # Template Toolkit
```

The Perl code requires a different syntax but the Template Toolkit code stays the same. This lets you change the underlying implementation at any time without having to change the templates. As long as the data is laid out in the same way (i.e., don't change `surname` to `last_name`), it doesn't really matter what data structures are used, or whether they are precomputed, fetched from a database, or generated on demand.

This uniform syntax also means that your template designers can remain blissfully ignorant of the difference between a hash array and an object. They don't have to worry about any confusing syntax and can concentrate on the task at hand of presenting the data nicely. This makes the template language as friendly as possible for people who aren't already Perl programmers.

The general rule is to use Perl for programming and the Template Toolkit for presentation. But again, it's not mandatory, so you're still free to bend (or break) the rules when you really need to.

1.2.4 The Template Toolkit Is Easy to Extend

The Template Toolkit is designed to be easy to extend. If it doesn't already do what you want, there's a good chance you can reimplement a small part of it to change it to do what you what. The object-oriented architecture of the Template Toolkit makes this process relatively straightforward, and there are programming hooks throughout the system to give you as much flexibility as possible.

A number of plugins exist for the Template Toolkit, and we cover them in Chapter 6. They are designed to give templates convenient control over things such as HTML tables, database connections, and CGI parameters.

1.3 Installing the Template Toolkit

At any one time you can download from the Web at least two possible versions of the Template Toolkit: a stable version and a developer version. The stable version has a version number such as 2.10, and has been widely tested before release. The developer versions have version numbers such as 2.10a, and typically have bug fixes and early implementations of new features. Generally, you should install the latest stable release.

1.3.1 Downloading

The Template Toolkit is available from the Comprehensive Perl Archive Network (CPAN). You can always download the most recent stable version of the Template Toolkit from http://search.cpan.org/dist/Template-Toolkit/ (which is where most people download it).

In addition, a web site is dedicated to the Template Toolkit. Located at http://www.template-toolkit.org, this site offers the latest stable version, as well as a number of other goodies such as native packages of the Template Toolkit for Debian GNU/Linux, Mac OS X (for installation using Fink), and Microsoft Windows (for installation using ActiveState's Perl Package Manager).

You can also get developer versions of the Template Toolkit from the web site. Normally, you need to download only the current stable version, but if you come across a bug that isn't fixed in the CPAN version, you may need to use a developer release.

If a developer release isn't cutting-edge enough for you, the web site contains information on how to get access to the CVS repository, which is where the very latest versions of the Template Toolkit source code are kept. If you want to add functionality to the Template Toolkit or have found a bug that you can fix, and you want your patch to be accepted by Template Toolkit developers, you should make your changes against the current CVS HEAD.

1.3.2 Installing

Installing the Template Toolkit is like installing any other Perl module (see *perlmodinstall(1)* for platform-specific details). The basic idea is as follows:

```
$ perl Makefile.PL

$ make

$ make test

$ make install
```

A few optional modules and pages of documentation come with the Template Toolkit, and how much of that gets installed is controlled by arguments to `perl Makefile.PL`. Run `perl Makefile.PL TT_HELP` to get the following full list of options:

```
The following options can be specified as command-line

arguments to 'perl Makefile.PL'.  e.g.,


  perl Makefile.PL TT_PREFIX=/my/tt2/dir TT_ACCEPT=y


  TT_PREFIX       installation prefix     (/usr/local/tt2)

  TT_IMAGES       images URL              (/tt2/images)

  TT_DOCS         build HTML docs         (y)

  TT_SPLASH       use Splash! for docs    (y)

  TT_THEME        Splash! theme           (default)

  TT_EXAMPLES     build HTML examples     (y)

  TT_EXTRAS       install optional extras (y)

  TT_XS_ENABLE    Enable XS Stash         (y)

  TT_XS_DEFAULT   Use XS Stash by default (y)
```

```
TT_DBI          run DBI tests           (y if DBI installed)


TT_LATEX        install LaTeX filter    (y if LaTeX found)
TT_LATEX_PATH       path to latex       (system dependant)
TT_PDFLATEX_PATH    path to pdflatex    (  "      "      " )
TT_DVIPS_PATH       path to dvips       (  "      "      " )


TT_QUIET        no messages             (n)
TT_ACCEPT       accept defaults         (n)


By default, the Makefile.PL runs in interactive mode,
prompting for confirmation of the various configuration
options.  Setting the TT_ACCEPT option causes the default
value (possibly modified by other command line options)
to be accepted.  The TT_QUIET option can also be set to
suppress the prompt messages.
```

The make test step is important, especially if you're using a developer release or version from CVS. Over 2,000 tests are provided with the Template Toolkit to ensure that everything works as expected, and to let you know about any problems that you might have. It takes no more than a minute or so to run the tests, and they can save you a great deal of debugging time in the unlikely event that something is wrong with your installation.

Test failures don't necessarily indicate that something is fatally wrong. A serious problem causes nearly all of the tests to fail, although we haven't heard of that happening to anyone for quite some time. More often than not, errors raised in the test suite come from plugin modules whose external Perl modules are not installed on your system or are the wrong version.

This kind of problem is rarely serious. At worst, it may mean that a particular plugin doesn't work as expected—or at all—but that won't stop the rest of the Template Toolkit from doing its job. You can usually solve the problem by installing the latest version of any dependent modules. If you are unsure about whether a particular test failure is significant, ask on the mailing list, or check the mailing list archives mentioned in Section 1.4.3, later in this chapter. Major problems tend to be reported by many people.

The *README* and *INSTALL* files in the Template Toolkit distribution directory provide further information about running the test suite and what to do if something goes wrong.

1.4 Documentation and Support

In this section, we take a look at the support that is available for the Template Toolkit.

1.4.1 Viewing the Documentation

The Template Toolkit comes with an incredible amount of documentation. The documentation is supplied in the standard Perl Plain Old Documentation (POD) format. Once you have installed the Template Toolkit, you can see any of the documentation pages using *perldoc* or *man*, just as you can with any other Perl module:

```
$ perldoc Template         # should always work

$ man Template             # does not work everywhere
```

During the Template Toolkit installation procedure you are offered the chance to install HTML versions of the documentation. The default location for the installation of these files is */usr/local/tt2* under Unix and *C:/Program Files/Template Toolkit 2* under Win32. The installation procedure prompts for alternate locations.

If you are running a web server on your local machine, you can configure it to know where these files are. For example, you might put the contents of Example 1-4 in the *httpd.conf* for an Apache web server.

Example 1-4. Apache configuration directives to view Template Toolkit documentation

```
# TT2

Alias /tt2/images/   /usr/local/tt2/images/

Alias /tt2/docs/     /usr/local/tt2/docs/html/

Alias /tt2/examples/ /usr/local/tt2/examples/html/


<Directory /usr/local/tt2/>

    Options Indexes

    AllowOverride None

    Order allow,deny

    Allow from all

</Directory>
```

You can now access the locally installed documentation by pointing your browser at *http://localhost/tt2/docs*. For more information on configuring your web server, see the *INSTALL* file that comes with the Template Toolkit.

The complete documentation set is also available online at the Template Toolkit web site. You can find it at http://www.template-toolkit.org/docs.html.

1.4.2 Overview of the Documentation

A large number of manual pages come with the Template Toolkit. Here is a list of some of the most useful ones:

Template

The manual page for the Template module, the main module for using the Template Toolkit from Perl.

Template::Manual

An introduction and table of contents for the rest of the manual pages.

Template::Manual::Intro

A brief introduction to using the Template Toolkit. Not unlike this chapter.

Template::Manual::Syntax

The syntax, structure, and semantics of the Template Toolkit directives and general presentation language. Chapter 3 covers this aspect.

Template::Manual::Variables

A description of the various ways that Perl data can be bound to variables for accessing from templates. Chapter 3 also has the details.

Template::Manual::Directives

A reference guide to all Template Toolkit directives, with examples of usage. See Chapter 4.

Template::Manual::VMethods

A guide to the virtual methods available to manipulate Template Toolkit variables. These are also covered in Chapter 3.

Template::Manual::Filters

A guide to the various standard filters that are supplied with the Template Toolkit. See Chapter 5.

Template::Manual::Plugins

A guide to the various standard plugins that are supplied with the Template Toolkit. See Chapter 6.

Template::Manual::Internals

An overview of the internal architecture of Template Toolkit. See Chapter 7.

Template::Manual::Config

Details of the configuration options that can be used to customize the behavior and extend the features of the Template Toolkit. This is covered in the Appendix.

Template::Manual::Views

A description of dynamic views—a powerful but experimental feature in the Template Toolkit. The use of views is covered briefly in Chapter 9.

Template::Tutorial

An introduction and table of contents to the tutorials that are distributed with Template Toolkit. Currently there are two. Template::Tutorial::Web is a quick start to using the Template Toolkit to create web pages, and Template::Tutorial::Datafile is a guide to creating datafiles in various formats (particularly XML). See Chapter 2 and Chapter 10 for more information about using the Template Toolkit to generate web pages and XML, respectively.

Template::Library::HTML and Template::Library::Splash

Two guides to using libraries of user interface components (widgets) for creating HTML with the Template Toolkit.

Template::Modules

A list of the various Perl modules that make up the Template Toolkit. Each module has its own manual page.

1.4.3 Accessing the Mailing List

If you can't find the answer to your questions in any of the documentation, you can always turn to the mailing list set up for discussion of the Template Toolkit. You can subscribe to the mailing list at: http://template-toolkit.org/mailman/listinfo/templates. All previous posts are archived at: http://template-toolkit.org/pipermail/templates.

Activity on the list is moderate (around 100 messages per month) and many of the Template Toolkit experts are on the list.

1.5 Using the Template Toolkit

The rest of this chapter provides a brief introduction to using the Template Toolkit. We look at the structure and syntax of templates, showing how variables and directives are embedded in plain text and expanded by the template processing engine. We talk about some of the different kinds of directives that the Template Toolkit provides, what they're used for, and how you go about using them.

We start by looking at the four main ways of using the Template Toolkit to process templates: from the command line using the *tpage* and *ttree* programs; from a Perl script using the Template module; and in a mod_perl-enabled Apache web server using the Apache::Template module.

1.5.1 tpage

The *tpage* program provides a quick and easy way to process a template file from the command line. The name of the template file is specified as a command-line argument. This is processed through the Template Toolkit processing engine, and the resultant output is printed to STDOUT:

```
$ tpage infile
```

You can use the > file redirect operator (if your operating system supports it, or something similar) to save the output into another file:

```
$ tpage infile > outfile
```

In this example, the input template, *infile*, is processed by *tpage* with the output saved in *outfile*. If something goes wrong and the template can't be processed (for example, if the input file specified doesn't exist or contains an invalid template directive or markup error), an error is printed to STDERR, and *tpage* exits without generating any standard output.

The following shows what happens if we try and coerce *tpage* into processing a file, *nosuchfile*, which doesn't exist on our system:

```
$ tpage nosuchfile

file error - nosuchfile: not found at /usr/bin/tpage line 60.
```

tpage offers just one command-line option, --define, which allows you to provide values for template variables embedded in the document. We saw this earlier in Example 1-1 where it processed the Vogon form letter:

```
$ tpage --define planet=Earth \

>       --define captain="Prostetnic Vogon Jeltz" \

>       --define time="two of your earth minutes" \

>       destruction.tt
```

The *tpage* program is ideal for simple template processing such as this, where nothing more is required than the ability to insert a few variable values. More complex tasks need the *ttree* program or custom programs using the Template module.

However, there is one last *tpage* trick we can show you. If you don't provide *tpage* with the name of a template file, it reads it from STDIN. This allows you to use it as Unix-style pipeline filter. For example, if the output of the *mktemplate* program is a Template Toolkit template, the following command can be used to pipe it into *tpage* to have it processed:

```
$ mktemplate | tpage
```

Invoking *tpage* by itself, with no arguments and no piped input, starts it in interactive mode. In this case, *tpage* sits and waits for you to type in a source template. This can be very useful for trying out small snippets of template syntax to see what they do.

Here's an example:

```
$ tpage

[% subject = 'cat'

  object  = 'mat'

%]

The [% subject %] sat on the [% object %].
```

```
^D

The cat sat on the mat.
```

The first line invokes *tpage* from the command line. The next three lines are the body of the template in which we type, followed by the end-of-file (EOF) character telling *tpage* that we're done. On Unix systems, this is Ctrl-D, shown in the example as ^D. On Microsoft Windows platforms, Ctrl-Z is the EOF character.

The rest of the example shows the output generated by *tpage* from processing the template. The cat is sitting on the mat, and everything is working as expected.

1.5.2 ttree

The *ttree* program offers many more features and options than *tpage* does. The first major difference is that *ttree* works with entire directories of templates rather than with single files. If you're using the Template Toolkit to build a web site, for example, you can point *ttree* at a directory of source templates to process them all, saving the generated HTML pages to corresponding files in an output directory.

The following example shows how you could invoke *ttree* to process all the templates in the *templates* directory (containing the files *cat* and *dog* for the purpose of this example), and save the generated output in files of the same name, which are located in the *output* directory:

```
$ ttree -s templates -d output -v
```

The -s option defines the source directory for templates, and -d defines the destination directory for output files. The -v (verbose) option causes *ttree* to print a summary of what it's doing to STDERR.

Here's an example of the kind of information generated by the -v option:

```
ttree 2.63 (Template Toolkit version 2.10)


      Source: templates

 Destination: output

Include Path: [  ]

      Ignore: [  ]

        Copy: [  ]

      Accept: [ * ]


  + dog

  + cat
```

This is a summary of the processing options, including the Source and Destination that we provided as the -s and -d command-line options. The *dog* and *cat* files are listed as the two files that *ttree* found in the *templates* directory. The + characters indicate that both files were successfully processed, creating *dog* and *cat* files in the *output* directory.

Now that these templates have been processed, *ttree* will not process them again until they are modified or the corresponding output file is deleted. By looking at the file modification times of the source template and destination file, *ttree* can decide which templates have changed and which have not. It saves time by processing only those that have changed.

If you run the same *ttree* command again, you see that the + characters to the left of the filenames have changed to - characters:

```
ttree 2.63 (Template Toolkit version 2.10)


      Source: templates

 Destination: output

Include Path: [  ]

      Ignore: [  ]

        Copy: [  ]
```

```
    Accept: [ * ]

  - dog                    (not modified)
  - cat                    (not modified)
```

These - characters indicate that the template files were not processed this time, with the reason given in parentheses to the right. This can save a great deal of time when building large document systems using templates (e.g., a typical web site) in which only a few pages change at any one time.

The -a option forces *ttree* to process all templates, regardless of their modification times:

```
$ ttree -a
```

A second benefit of *ttree* is that it offers numerous options for changing its behavior. Adding a standard header and footer to each page template, for example, is as easy as setting the relevant option:

```
$ ttree -s templates -d output -v \
>       --pre_process=header \
>       --post_process=footer
```

The number of options can be overwhelming at first, but in practice, only a few are used on a regular basis. To avoid having to always use the command line to specify options—something that can quickly become cumbersome and error prone, especially if you are using more than a few—*ttree* allows you to use configuration files to define all the options for a particular web site or other document system. You can then invoke *ttree*, passing the name of the configuration file using the -f option:

```
$ ttree -f /home/dent/web/ttree.cfg
```

Example 1-5 shows a sample *ttree* configuration file.

Example 1-5. A sample ttree configuration file, ttree.cfg

```
src  = /home/dent/web/templates
dest = /home/dent/web/html
lib  = /home/dent/web/lib

pre_process  = header
post_process = footer

verbose
```

In the configuration file, the -s and -d options are represented by the src and dest options. We also added a lib option (-l on the command line), which tells *ttree* about an additional library directory where our *header* and *footer* templates are found.

Setting up *ttree* is a little more involved than using *tpage*, but the effort quickly pays off in the time it saves you. We look at *ttree* in detail in Chapter 2, showing everything from first-time use through writing and managing configuration files.

1.5.3 The Template Module

Both *tpage* and *ttree* use the Template Perl module to do the dirty work of processing templates. As it happens, the Template module doesn't actually do much in the way of dirty work itself, but delegates it to other modules in the Template Toolkit with exotic names such as Template::Service, Template::Context, Template::Provider, and Template::Stash. The Template module provides a simple interface for using the Template Toolkit from Perl so that you don't have to worry about the complex underlying functionality that makes it work. Chapter 7 goes into greater detail about what lurks beneath the hood of the Template Toolkit, but for now we cover just the basics.

If you are already a Perl hacker experienced in using modules, the Template manpage gives you an executive summary to get you quickly up to speed. If you're not a Perl hacker but would like to be, *Learning Perl*, Third Edition, by Randal Schwartz and Tom Phoenix (O'Reilly) is a good place to start.

However, you don't need to know any Perl to use the Template Toolkit. Thanks to the *tpage* and *ttree* programs, you can build your entire web site or other template-based document system without ever having to write a line of Perl code. Nevertheless, it's useful to have a basic understanding of how the Template module is used in Perl programs (including *tpage* and *ttree*), even if you never plan on using the module. Also, certain features are accessible only through Perl (for example, the ability to define a subroutine to return the value for a variable), so there is a good chance that sooner or later you will want or need those Perl-specific features.

Example 1-6 shows a simple Perl program for processing the *destruction.tt* template from Example 1-1.

Example 1-6. A Perl program for processing the Vogon form letter template

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;

my $tt    = Template->new( );
my $input = 'destruction.tt';
my $vars  = {
    planet  => 'Earth',
    captain => 'Prostetnic Vogon Jeltz',
    time    => 'two of your earth minutes',
};

$tt->process($input, $vars)
    || die $tt->error( );
```

The first line defines the path to the Perl interpreter on your system. This is very much a Unix-specific convention. On a Windows machine, for example, this line is not relevant or required.

In the first block, we enable Perl's strict and warnings pragmata and then load the Template module:

```
use strict;
use warnings;
use Template;
```

> It is good Perl style to include use strict; and use warnings; at the top of every program, or to invoke Perl with the -w switch instead of use warnings; for versions of Perl earlier than 5.6.0. These two precautions will catch many common programming and typographical errors, and warn you about any questionable practices. Perl examples in this book may omit them for brevity, but you should always include them in any nontrivial chunk of code.

The next line creates a new Template object and assigns it to the $tt variable:

```
my $tt = Template->new( );
```

We store the name of the template to be processed in the $input variable and define some template variables in $vars:

```
my $input = 'destruction.tt2';

my $vars  = {

    planet  => 'Earth',

    captain => 'Prostetnic Vogon Jeltz',

    time    => 'two of your earth minutes',

};
```

Then we invoke the process( ) method against the $tt template object to process the source template:

```
$tt->process($input, $vars)

    || die $tt->error( );
```

The name of the source template file, here stored in the $input variable, is passed as the first argument, followed by a reference to a hash array of template variables, defined in $vars.

The process( ) method processes the template and returns a true value to indicate success. The output is printed to STDOUT by default so that you see it scrolling up your screen when you run the program.

If an error occurs, the process( ) method returns false. In this case, we call the error( ) method to find out what went wrong and report it as a fatal error using die. An error can be returned for a number of reasons, such as the file specified could not be found, had embedded directives containing illegal syntax that could not be parsed, or generated a runtime error while the template was being processed.

1.5.3.1 Template configuration options

We mentioned the --pre_process and --post_process options when using *ttree* earlier. Now we can see how these are used in the underlying Perl implementation.

Configuration options are passed to the new( ) constructor method as a reference to a hash, as shown in Example 1-7. The Template module expects options to be provided in uppercase, so the options for *ttree* translate to the PRE_PROCESS and POST_PROCESS options for the Template module. We also set the INCLUDE_PATH option to indicate the location of the source and library templates, which *ttree* provides from the src (or -s) and lib (or -l) options. These are provided as a reference to a list of the two directory paths.

Example 1-7. Specifying options when processing templates, ttperl3.pl

```
my $tt = Template->new({

    PRE_PROCESS  => 'header',

    POST_PROCESS => 'footer',

    INCLUDE_PATH => [

        '/home/dent/web/templates',     # src

        '/home/dent/web/lib',           # lib

    ],

});
```

Now when the process( ) method is invoked against the $tt object, the source template, *destruction.tt*, will be processed complete with the *header* and *footer* added before and after the main page content, respectively. For this example, we are assuming that the *destruction.tt* template is located in the */home/dent/web/templates* directory, and that *header* and *footer* can be found in the */home/dent/web/lib* directory.

The Template Toolkit provides numerous configuration options. These are described in detail in the Appendix. We describe the useful ones as we encounter them in later chapters.

1.5.4 Apache::Template Module

The Apache::Template module marries the Template Toolkit with the Apache web server. It is distributed separately from the rest of the Template Toolkit and can be downloaded at http://search.cpan.org/dist/Apache-Template/. It requires an Apache installation that includes Doug MacEachern's mod_perl extension module, details of which can be found at http://perl.apache.org/. For a full discussion of mod_perl, we recommend *Practical mod_perl*, by Stas Bekman and Eric

Cholet (O'Reilly), which contains an appendix dealing specifically with using the Template Toolkit under Apache and mod_perl.

Apache::Template can be configured via Apache's normal *httpd.conf* configuration file. Example 1-8 shows an extract of an *httpd.conf* file that sets the same options as Example 1-7.

Example 1-8. httpd.conf directives to set options with Apache::Template

```
PerlModule        Apache::Template


TT2IncludePath      /home/dent/web/templates

TT2IncludePath      /home/dent/web/lib

TT2PreProcess       header

TT2PostProcess      footer


TT2Params           uri env params cookies

TT2Headers          modified length


<Files *.tt2>

    SetHandler      perl-script

    PerlHandler     Apache::Template

</Files>
```

The first section loads the Apache::Template module:

```
PerlModule        Apache::Template
```

The next block sets some standard Template Toolkit options:

```
TT2IncludePath      /home/dent/web/templates

TT2IncludePath      /home/dent/web/lib

TT2PreProcess       header

TT2PostProcess      footer
```

Apache::Template adopts the Apache convention of using StudlyCaps for the names of configuration options and also adds a unique TT2 prefix. So the Apache::Template options TT2IncludePath and TT2PreProcess, for example, equate to the INCLUDE_PATH and PRE_PROCESS options for the Template module.

The two options that follow are specific to the Apache::Template handler:

```
TT2Params           uri env params cookies

TT2Headers          modified length
```

The first, TT2Params, provides a list of items that the handler should automatically extract from the Apache request and make available as template variables. Any template can use the uri, env, params, and cookies variables to access the request URI, environment variables, request parameters, and cookies, respectively. The second directive, TT2Headers, indicates that Last-Modified and Content-Length headers should be automatically added to the response sent to the client.

The final section uses the Apache Files directive to define the files that should be processed as templates:

```
<Files *.tt2>

    SetHandler      perl-script

    PerlHandler     Apache::Template

</Files>
```

The SetHandler and PerlHandler directives within the Files block are standard procedure in Apache for binding a mod_perl handler (Apache::Template in this case) to a set of files. With this configuration, the Apache server processes any files with a *.tt2* extension using the Apache::Template handler, but continues to deliver pages with any other extensions as static files, or using any other handlers defined for them.

This is a convenient way of mixing static HTML pages with dynamic page templates in any directory that is currently accessible by the Apache web server. If you want to create a static page, use a *.html* or other appropriate extension. If you want to create a dynamic page from a template, with the appropriate headers and footer added automatically, simply give it a *.tt2* extension and leave Apache::Template to take care of it.

If you would rather not open up your entire web server to the Apache::Template module, you can instead use the Location directive.

```
<Location /tt2/>
    SetHandler      perl-script
    PerlHandler     Apache::Template
</Location>
```

In this case, only those files located under the */tt2/* URI will be processed through the Apache::Template handler.

There are numerous other Apache configuration directives, all of which are described in the documentation provided with Apache. For a full discussion of the Apache::Template configuration, see the Appendix.

1.6 The Template Toolkit Language

The Template Toolkit language is a presentation language rather than a general-purpose programming language. It provides the kind of features that you would expect to see in a regular programming language, including loops, conditional tests, and the ability to manipulate variable values. However, in this case they serve a slightly different purpose. The Template Toolkit is designed for the task of generating content and presenting data, and it generally leaves more complex issues to a real programming language, namely, Perl.

We have already seen the basics of what a template looks like—a mixture of tags (known as directives) and other fixed text. The template processor interprets the directives and the remaining text is passed through unchanged.

By default, the start and end of a directive are marked by the sequences `[%` and `%]`, but the `TAGS` directive can be used to change them if you don't like these. The `TAGS` directive takes either one or two arguments. The single-argument version expects the name of a predefined tag set. For example, the `star` set replaces the tag delimiters with `[*` and `*]`:

```
[% TAGS star %]

People of [* planet *], your attention please.
```

If you give `TAGS` two arguments, they define the start and end tag markers that you want to use. For example, if you're processing plain text, you might find something like this more lightweight and easier to type:

```
[% TAGS { } %]

People of {planet}, your attention please.
```

Or if you are processing HTML and you prefer an HTML style, how about this:

```
[% TAGS <tt: > %]

<p>People of <tt:planet>, your attention please.
```

Changes to tags take effect immediately and affect only the current file.

You can also set these from the command line with *ttree* by using the `--start_tag`, `--end_tag`, and `--tag_style` options. From a Perl script, the corresponding configuration options for the `Template` module are `START_TAG`, `END_TAG`, and `TAG_STYLE`. For `Apache::Template`, the `TT2Tags` option can be used with one or two arguments, as per the `TAGS` directive.

In the rest of this book, we use the default tag style. We like it because it makes the directives stand out from the surrounding text, rather than making them blend in. We think it makes templates easier to read and write when you can more clearly distinguish one part from another.

1.7 Template Variables

The variables that we have used so far have been *scalar* variables. A scalar variable stores a single piece of information —either a string or a number.

The value of a scalar variable is inserted in a template by using the variable name inside a directive like this:

```
[% planet %]
```

A variable wouldn't be worthy of the name if you couldn't also set its value. We have seen examples of doing this using the --define option of the *tpage* command, but it is also possible to set a variable's value inside a template:

```
[% planet = 'Magrethea' %]

People of [% planet % ], your attention please.
```

1.7.1 Complex Variables

In addition to scalar variables, the Template Toolkit also supports two complex data types for storing multiple values: the *list* and *hash array* (also known as a *hash*). A list is an ordered array of other variables, indexed numerically and starting at element 0. A hash is an unordered collection of other variables, which are indexed and accessible by a unique name or *key*.

Perl programmers will already be familiar with these data structures. When you use the Template Toolkit from Perl you can easily define hash arrays and lists that are then passed as template variables to the process( ) method.

Example 1-9 shows a Perl program similar to Example 1-6, which defines a list of friends and a hash of terms as template variables.

Example 1-9. Perl program to process friends.tt

```
use Template;


my $tt    = Template->new( );

my $input = 'friends.tt';

my $vars  = {

    friends => [ 'Ford Prefect', 'Slartibartfast' ],

    terms  => {

        sass  => 'know, be aware of, meet, have sex with',

        hoopy => 'really together guy',

        frood => 'really, amazingly together guy',

    },

};


$tt->process($input, $vars)

    || die $tt->error( );
```

Example 1-10 is the *friends.tt* template that Example 1-9 processes.

Example 1-10. The friends.tt template

```
Your friends are:
[% FOREACH friend IN friends -%]
 * [% friend %]
[% END -%]

You know the following terms:
[% FOREACH term IN terms.keys.sort -%]
 [% term %]: [% terms.$term %]
[% END -%]
```

This is the output generated by Example 1-9:

```
Your friends are:
 * Ford Prefect
 * Slartibartfast

You know the following terms:
 frood: really, amazingly together guy
 hoopy: really together guy
 sass: know, be aware of, meet, have sex with
```

There will be times when you're using the Template Toolkit with *tpage* or *ttree* and don't want to have to write a Perl program, however simple, just to use some complex variables. The Template Toolkit allows you to define lists and hash data structures inside templates, using syntax similar (or identical if you prefer) to the Perl equivalents shown earlier.

The simple examples in the sections that follow should give you a flavor of how lists and hash data structures are defined and used in templates. Chapter 3 describes the Template Toolkit language in detail, showing the different variations in syntax that are permitted to satisfy both Perl programmers (who expect `=>` to be used to separate a hash key from a value, for example) and HTML designers (who probably don't know any different and are just as happy using the simpler `=`).

1.7.2 Lists

A list variable is defined in a template using the `[...]` construct. Here's how we would create the equivalent of the `friends` list from Example 1-9:

```
[% friends = [ 'Ford Prefect', 'Slartibartfast' ] %]
```

List elements are accessed using the dot operator (`.`). Follow the list name with a dot and then the element number, starting at zero for the first element:

```
[% friends.0 %]      # Ford Prefect
[% friends.1 %]      # Slartibartfast
```

It is also possible to access elements from the list using a variable containing an index value. Simply prefix the variable with a `$` character:

```
[% index = 1 %]
[% friends.$index %]   # Slartibartfast
```

1.7.3 Hashes

A hash is defined in a template using the `{...}` construct:

```
[%  terms = {
      sass  = 'know, be aware of, meet, have sex with'
      hoopy = 'really together guy'
      frood = 'really, amazingly together guy'
   }
%]
```

Each pair of items within the { and } is composed of the key, to the left of the = (or => if you prefer), and the value to the right. Separate pairs of items with commas, although it's not obligatory. Here is the same template written in a Perlish style:

```
[%  terms => {
      sass  => 'know, be aware of, meet, have sex with',
      hoopy => 'really together guy',
      frood => 'really, amazingly together guy',
   }
%]
```

Hash items are also accessed using the dot operator. In this case, the key for the required item is specified after the dot character:

```
[% terms.hoopy %]    # really together guy
```

You can also access hash items using a variable that contains a key. Again, the variable name should be prefixed with a $ character:

```
[% key = 'frood' %]
[% terms.$key %]     # really, amazingly together guy
```

1.7.3.1 Nesting list and hash definitions

Lists and hashes can be nested inside each other to create complex data structures:

```
[% arthur = {
     name    = 'Arthur Dent',
     planet  = 'Earth',
     friends = [
       { name = 'Ford Prefect'
         home = 'Betelgeuse'
         type = 'frood' }
       { name = 'Slartibartfast'
         home = 'Magrethea'
         type = 'hoopy' }
     ]
     terms = {
       sass  = 'know, be aware of, meet, have sex with'
       hoopy = 'really together guy'
       frood = 'really, amazingly together guy'
```

```
    }
  }
%]
```

You can access items buried deep within a nested data structure by chaining together a series of dot operations to create a *compound variable*:

```
[% arthur.friends.1.name %] # Slartibartfast
```

The Template Toolkit works out which dot operators are performing hash lookups (friends and name) and which are performing list lookups (1), and then automatically does the right thing to return the correct value. Comparing this to the equivalent Perl code, the Template Toolkit's uniform dot operator makes things much clearer:

```
# TT
arthur.friends.1.name


# Perl
$vars->{arthur}->{friends}->[1]->{name}
```

This illustrates one of the key benefits of using a presentation language like the Template Toolkit for generating content, rather than a programming language such as Perl.[1] When you write a program using a real programming language such as Perl, it's important to know which variables are scalars and which are lists, hashes, subroutines, objects, and so on. It's also critical that you use exactly the right kind of syntax relevant to each data type. Otherwise, your program might try to do something that it shouldn't, possibly corrupting the data, causing the program to exit with an error, or even failing to compile and run in the first place.

[1] Which of course, we still rely on a great deal, not only as the language in which the Template Toolkit is written, but also as the means by which you can extend it and add your own custom functionality to your templates, as we will see in the next section.

However, when you're writing templates to present your data as HTML pages, or in some other output format, these issues are of less concern. You're far more interested in how the data is going to be laid out, than in how it is stored or calculated on demand by the underlying Perl code (as we see in the next section). As long as the value for a user's name, for example, is inserted in the right place in the template when we ask for arthur.friends.1.name, we're happy. By the time the data is presented as output in a template, it is all text anyway.

You can also used dotted variables as hash keys to reference other variables. The following example shows how this is done using ${ ... } to explicitly scope the range of the second variable name:

```
[% arthur.terms.${arthur.friends.1.type} %]
```

The arthur.friends.1.type variable returns the value hoopy, resulting in a final expression equivalent to arthur.terms.hoopy. This ultimately provides us with the value really together guy.

You can use a temporary variable to break this down into smaller pieces. For example:

```
[% friend = arthur.friends.1 -%]

[% friend.name %] is a [% arthur.terms.${friend.type} %].
```

This generates the following output:

```
Slartibartfast is a really together guy.
```

1.7.4 Dynamic Variables

The examples that we've seen so far have used variables to store static values. When you set a variable to contain a scalar value or a reference to a list or hash array, it remains set to that value until the next time you explicitly modify it. Whenever the variable is used, the Template Toolkit simply looks up the current value for the variable and inserts it in the right place.

The Template Toolkit also allows subroutines and objects to be used to create dynamic variables. Each time such a variable is used, the Template Toolkit will call the subroutine or object method bound to it to return an appropriate value. Whereas static variables contain precomputed values, these dynamic variables return values that are recomputed each time they are used.

Example 1-11 shows a Perl program that defines two template variables, one bound to a subroutine, the other to an object.

Example 1-11. Dynamic data in template variables

```
use Acme::Planet;       # not a real module (yet)


my $vars = {

    help => sub {

        my $entry = shift;

        return "$entry: mostly harmless";

    },

    planet => Acme::Planet->new( name => 'Earth' ),

};
```

In this example, the help variable is a reference to a subroutine that expects a single argument, $entry. The planet variable references a hypothetical Acme::Planet object. This isn't a real module (at the time of this writing), but we're assuming that the new constructor method creates an Acme::Planet object against which we can invoke the name( ) method to return the value provided, Earth.

The following extract shows how these variables can be used in a template:

```
The guide has this to say about [% planet.name %].

  [% help(planet.name) %]
```

This would generate the following output:

```
The guide has this to say about Earth.

  Earth: mostly harmless
```

Notice that when we call the name method on planet we use the dot operator in exactly the same way as we would if planet were a hash with a key called name. The Template Toolkit doesn't care which of these we have, it just looks at the variable and works out what is the right thing to do. This illustrates how you are not tied down to any particular implementation for your underlying data structures, and can freely change from hashes to objects and back again without affecting the templates that use them.

Dynamic variables must be defined in Perl. There is no easy or clean way to define dynamic variables from within a template, other than by enabling the EVAL_PERL configuration option and using embedded Perl. The preferred solution is to write a simple Perl script that defines the relevant subroutines, objects, and other data items and then processes the appropriate template or templates. Another approach is to write a Template Toolkit plugin that encapsulates the Perl code and can be loaded into any template on demand. We look at plugins in detail in Chapter 6.

1.7.5 Virtual Methods

The Template Toolkit provides virtual methods for manipulating and accessing information about template variables. For example, the length virtual method can be applied to any scalar variable to return its string length in characters. The virtual method is applied using the dot operator:

```
[% name = 'Slartibartfast' %]

[% name %]'s name is [% name.length %] characters long.
```

This generates the output:

```
Slartibartfast's name is 14 characters long.
```

Virtual methods are provided for the three main variables types: scalars, lists, and hashes. The following example shows the join list virtual method being used to return the elements in a list joined into a single string. It adds a single space character between each item in the list by default, but you can provide a different delimiter by passing it as an argument in parentheses.

```
[% friends = [ 'Andy', 'Darren', 'Dave'] %]

Your friends are [% friends.join(', ') %].
```

This will display:

```
Your friends are Andy, Darren, Dave.
```

Some virtual methods alter the contents of the variable that they act on. For example, the pop method removes the last item from a list and returns it:

```
[% last = friends.pop %]

Your friends are [% friends.join(', ') %] and [% last %].
```

This will display:

```
Your friends are Andy, Darren and Dave.
```

We saw an example earlier of how virtual methods were combined in a dotted variable:

```
You know the following terms:

[% FOREACH term IN terms.keys.sort -%]

 [% term %]: [% terms.$term %]

[% END -%]
```

The part that we're particularly interested in is this:

```
terms.keys.sort
```

The terms variable contains a reference to a hash. The keys hash virtual method returns a reference to a list of the keys in the hash. The keys aren't returned in any particular order, but now that we have a list, we can go on to call the sort list virtual method to return a second list containing the items sorted in alphabetical order.

We can then go one step further and call the join virtual method on that list, to join the items into a single string:

```
[% terms.keys.sort.join(', ') %]
```

This generates the following output:

```
frood, hoopy, sass
```

Virtual methods are covered in detail in Chapter 3.

1.8 Template Directives

The examples we have looked at so far have concentrated on the use of variables. The Template Toolkit also provides more advanced language constructs called *directives*. These begin with an uppercase keyword such as PROCESS, IF, or FOREACH and tell the template processing engine to do something.

1.8.1 Variable Directives

Given that directives start with an uppercase keyword, you might be forgiven for thinking that the examples we have seen so far don't count as directives:

```
[% name = 'Arthur Dent' %]

[% planet = { name = 'Earth' } %]

Welcome [% name %] of [% planet.name %].
```

However, the syntax that we have been using until now to set and get variables is actually just a convenient shortcut for the full version, which uses the SET and GET keywords like so:

```
[% SET name = 'Arthur Dent' %]

[% SET planet = { name = 'Earth' } %]

Welcome [% GET name %] of [% GET planet.name %].
```

For obvious reasons, the shorter versions are used most of the time.

1.8.2 Template Processing Directives

Another use of template directives is for changing the way templates are processed. The PROCESS directive is one of the simplest. It loads another template file, processes the contents, and inserts the generated output in the calling template:

```
[% PROCESS header %]
```

The Template Toolkit provides the INCLUDE_PATH option, which allows you to specify one or more directories where your template files can be found. This allows you to specify your templates with simple names such *header*, rather than full file paths such as */home/dent/templates/lib/header*, for example.

The reason that it is called INCLUDE_PATH and not PROCESS_PATH becomes obvious when we mention that there is also an INCLUDE directive. The INCLUDE directive and related INCLUDE_PATH option have been part of the Template Toolkit, and the Text::Metatext module that preceded it, from the very beginning. The PROCESS directive, on the other hand, was added at a later date, and was able to reuse the INCLUDE_PATH option for the same purposes.

The difference between PROCESS and INCLUDE is revealed in Chapter 2. For now it suffices to know that INCLUDE is most often used when you want to pass variable values that should remain local to that one template:

```
[% INCLUDE header

   title = 'Vogon Poetry'

%]
```

The Template Toolkit is quite relaxed about how you lay out directives. You can add as little or as much whitespace as you like (including newlines) to help make your directive more readable. The only rule is that you must separate individual words and phrases in the directive (e.g., the INCLUDE keyword and the header template name that follows it) with at least one whitespace character. You don't need any spacing between the opening tag and the start of the directive, or between the end of the directive and the closing tag, but we recommend it to help make directives easier to read.

The following examples are all valid and equivalent ways of writing the same directive:

```
[%INCLUDE header title='Vogon Poetry'%]



[% INCLUDE header title='Vogon Poetry' %]
```

```
[% INCLUDE header
     title = 'Vogon Poetry'
%]
```

1.8.3 Loops

The FOREACH directive allows you to create loops, where a block of template content is processed, once for each item in a list. Here's the general form:

```
[% FOREACH item IN list %]
  block of template content...
  ...can contain directives...
  ...and reference the [% item %] variable...
[% END %]
```

We've already seen a real example of this in action:

```
You know the following terms:
[% FOREACH term IN terms.keys.sort -%]
 [% term %]: [% terms.$term %]
[% END -%]
```

We know from looking at virtual methods earlier that the terms.keys.sort variable returns a list of the items frood, hoopy, and sass. So our loop block will be repeated three times, with the term variable set to each of those values in turn. We print the term followed by its definition, fetched from the terms hash array using the value of term as the key. The term variable must be prefixed with $ to indicate that the value of the variable should be used rather than the literal string term:

```
 [% term %]: [% terms.$term %]
```

The output generated for the complete block is as follows:

```
You know the following terms:
 frood: really, amazingly together guy
 hoopy: really together guy
 sass: know, be aware of, meet, have sex with
```

1.8.4 Conditionals

Conditionals are another powerful language feature that allow your templates to make decisions about what to process and what not to process, based on the values of variables and more complex expressions.

We saw an example of the IF directive in Example 1-3, shown here in condensed form for brevity:

```
[% IF order.destruction %]
  As you will no doubt be aware...
[% ELSE %]
  Our representatives will be...
[% END %]
```

If the order.destruction variable is true, the first block, between the IF and ELSE directives, is processed. Otherwise, the block between the ELSE and END is used.

The notion of *truth* is, in this sense, the same as it is for Perl. If the variable is defined and contains any kind of value except an empty string or the number zero, both Perl and the Template Toolkit will consider it to be true. If the variable is undefined, or contains a zero-length string or the number zero, it is false. This applies to all Template Toolkit directives that perform operations based on evaluating a variable or more complex expressions for truth.

1.8.5 Filters, Plugins, and Macros

There's plenty more in the Template Toolkit that we introduce in the chapters that follow. The following examples give a taste of what is to come.

Filters allow you to postprocess the output of a block of template markup. The `html` filter, for example, will convert any HTML-sensitive characters, such as `<`, `>`, and `&`, into their equivalent HTML entities, `<`, `>`, and `&`.

```
[% FILTER html %]

  Home > Dent > Friends > Slartibartfast

[% END %]
```

This generates the following output, which, when displayed as HTML on a web browser, will show the original `>` characters as intended:

```
Home &gt; Dent &gt; Friends &gt; Slartibartfast
```

See Chapter 5 for further details.

Plugins allow you to load and use Perl modules in templates without having to write a Perl wrapper program to do it for you. The following examples show how the CGI plugin (which delegates to Lincoln Stein's `CGI.pm` module) can be used for CGI programming:

```
[% USE CGI %]

[% name   = CGI.param('name')   or 'Arthur Dent' %]

[% planet = CGI.param('planet') or 'Earth' %]

 Welcome [% name %] of planet [% planet %].
```

Plugins also have their own chapter, Chapter 6.

The final teaser that we're going to show you is the `MACRO` directive. This allows you to provide simple names for more complex commands, as the following example shows:

```
[% MACRO header(title, author)

    IF name =  = 'Arthur Dent';

      INCLUDE arthur/header

        title = "Arthur Dent: $title";

    ELSE;

      INCLUDE guest/header

        title = "Guest User: $title";

    END;

%]
```

Don't worry if you can't make much sense of that now. The point that we're illustrating is that sometimes Template Toolkit code can get quite complex. However, the `MACRO` directive allows you to define the complicated part in one place so that you can use a much simpler call to the macro in the rest of your templates:

```
[% header('Arthur Dent', 'My Home Page') %]
```

1.9 Integrating and Extending the Template Toolkit

A particular strength of the Template Toolkit is that it doesn't try and do everything by itself. It concentrates on providing features that are generally applicable to template processing, leaving application-specific functionality to be added using Perl.

We've seen how you can define dynamic variables to allow your templates to access subroutines and objects written in Perl. The plugin mechanism allows you to bundle Perl code in self-contained modules that can be loaded straight into a template with a USE directive, eliminating the need to write a Perl wrapper program.

If that isn't enough, you can also define your own filters and virtual methods, and even change the language itself if you're feeling brave. This is covered in Chapter 8.

The fundamental concept that we're trying to get across is that the Template Toolkit is, as the name suggests, a *toolkit* for building things. It was designed to be easily extended and integrated with other components so that it can work within your requirements. It is not a complete web programming language or content management system that tries to do everything, and thus forces you into its way of thinking and working.

Sometimes that means you've got a little more thinking to do for yourself, rather than just blindly following the One True Way that we could have chosen for you. However, the benefit is that your solutions will be more flexible and adaptable, as well as better suited to addressing the problems at hand.

No two web sites (or document systems in general) are alike. Similarly, no two web developers agree on every issue that presents itself in the design and implementation of a web site. They each have their own ideas about the best way to tackle different problems, and prioritize different concerns according to the unique perspective that their past experience affords them. Perfect solutions don't exist (or if they do, we've never encountered them). With this in mind, strive to build a system that works today and tomorrow, even if it doesn't solve every problem overnight. Know when to compromise ideals for the sake of a pragmatic solution and when to stand firm on the issues that are important.

So the golden rule of web programming is that there is no golden rule. There are golden tools, and we like to consider the Template Toolkit among them, but a tool is only as good as the person who uses it. In the next chapter, we look at using the Template Toolkit to generate web content so that you can become familiar with its ways and start crafting your own web sites.

Chapter 1. Getting Started with the Template Toolkit

The Template Toolkit is an all-Perl template processing system. Unlike many other Perl templating systems, it is as proficient at producing HTML as it is at producing XML, PDF, or any other output format. It has its own simple templating language, so templates can be written and edited by people who do not know Perl. And it supports command-line, modular, CGI, and mod_perl operation.

In this chapter, we compare the Template Toolkit to other templating systems such as HTML::Mason and HTML::Template, describe how to install it, then show you what templates look like and how to process them. The goal of this chapter is to get you started—you should be able to install the Template Toolkit, write and understand basic templates, and know how to process the templates from the command line, from Perl programs, and from mod_perl.

10.1 Simple XML Processsing

Example 10-1 shows an XML file that contains details of a company's current inventory of widgets. We have each widget's part number, name, price, and current stock. This data might be generated by a stock control system.

Example 10-1. Stock control data

```
<inventory>
  <product id="0050">
    <name>Basic Widget</name>
    <price>49.99</price>
    <stock>2500</stock>
  </product>
  <product id="0051">
    <name>Cheap Widget</name>
    <price>29.99</price>
    <stock>5000</stock>
  </product>
  <product id="0101">
    <name>Super Widget</name>
    <price>99.99</price>
    <stock>1000</stock>
  </product>
  <product id="0102">
    <name>Ultra Widget</name>
    <price>149.99</price>
    <stock>500</stock>
  </product>
</inventory>
```

Suppose that we want to produce a report based on this data and also want to include the value of the stock. We can use the XML.Simple plugin to do this. Example 10-2 shows one way that we might do it.

Example 10-2. Template to create a stock report

```
[% USE inventory = XML.Simple('products.xml') -%]
[% FOREACH product = inventory.product.keys.sort;
   current = inventory.product.$product -%]
[% current.id %] [% product %]
[%- current.stock | format('%5d') %] units @
[%- current.price | format('%6.2f') -%] =
```

```
[%- current.stock * current.price | format('%10.2f') %]

[%- total = total + current.stock * current.price %]

[% END -%]

            Total value: [% total | format('%10.2f')%]
```

XML.Simple is given the name of an XML document and it builds a data structure that contains all of the data from that document. The USE directive returns a reference to this data structure, which we can then access using standard Template Toolkit techniques. In this case, the data structure it builds is a multilevel hash.

At the top level, the hash has only one key, product (representing the <product> tags from the original document). The value is a reference to another hash. The keys in this second hash are the names of the products, and the values are references to other hashes containing the details of the product. We can therefore use the expression inventory.product.keys.sort to get a list of the product names in alphabetical order.

To cut down on typing, we create a temporary variable, current, which contains the hash representing the current product. We can then access various parts of that hash to get the data that we want. Notice that we calculate the value of the current stock in each product and also keep a running total (in total) that we can display in the end. We also make use of the format filter to ensure that all of the numbers line up neatly.

The output generated by Example 10-2 is shown in Example 10-3.

Example 10-3. Generated stock report

```
0050 Basic Widget 2500 units @  49.99 =  124975.00

0051 Cheap Widget 5000 units @  29.99 =  149950.00

0101 Super Widget 1000 units @  99.99 =   99990.00

0102 Ultra Widget  500 units @ 149.99 =   74995.00

            Total value:  449910.00
```

For many tasks, XML.Simple is a perfectly adequate approach, however there will certainly be times when you need something that is a little more sophisticated. We'll look at XML.DOM and XML.XPath later in this chapter, but first we'll take a short detour to look at how we might create XML documents using the Template Toolkit.

10.2 Creating XML Documents

In order to demonstrate how to create XML documents using the Template Toolkit, we will use the example of creating an XML document that contains data about a TV show. Let's use (to pick a show at random) *Buffy the Vampire Slayer*.

10.2.1 Modeling Data About a TV Show

A TV show consists of a number of seasons. Generally, one season is made each year. Each season will have a regular cast. A season consists of a number of episodes. We want to create an XML file that contains all of this data.

We won't go into the details of how we access the data about the TV show. We'll just assume the existence of a module called `TVShow.pm` that will be our interface to details about a show. `TVShow.pm` has a constructor, `new`, which is passed the name of a show and returns an object that contains all of the data we need. It also has access methods that return all of these values.

We'll further assume the existence of `Template::Plugin::TVShow`, which allows us to use a `TVShow` object in our templates.

10.2.2 DTD for a TV Show

When designing an XML document, it's useful to create a *Document Type Definition* (or DTD) that defines what the XML document will look like. A DTD simply helps you to focus on the structure of the document. None of the Template Toolkit XML tools currently makes any use of the DTD.

Here's the DTD that we'll be using for our XML:

```
<!ELEMENT show (name, creator, seasons)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT creator (#PCDATA)>

<!ELEMENT seasons (season+)>

<!ELEMENT season (cast, episodes)>

<!ATTLIST season number CDATA>

<!ATTLIST season year CDATA>

<!ELEMENT cast (regular+)>

<!ELEMENT regular (character, actor)>

<!ELEMENT character (#PCDATA)>

<!ELEMENT actor (#PCDATA)>

<!ELEMENT episodes (episode+)>

<!ELEMENT episode (name, summary)>

<!ATTLIST episode number CDATA>

<!ATTLIST episode date CDATA>
```

While there are a large number of elements in this DTD, it isn't very complex. In English, the description looks something like this:

- A TV show consists of a name, a creator, and a list of seasons.
- A list of seasons consists of one or more seasons.
- A season consists of a cast and a list of episodes. It has two attributes—the season number and the year of broadcast.
- A cast consists of one or more regulars.

- A regular has a character name and an actor name.
- An episode list consists of one or more episodes.
- An episode has a name and a summary. It has two attributes—the episode number and the date of first transmission.

For more information on creating and interpreting DTDs, see *XML in a Nutshell* by Elliotte Rusty Harold and W. Scott Means, or *Learning XML* by Eric T. Ray (both by O'Reilly).

10.2.3 XML Template

Example 10-4 shows a simple template that will use the TVShow module to create an XML document conforming to our DTD.

Example 10-4. Sample template to create an XML document

```
[% USE show = TVShow(name) -%]
<?xml version="1.0"?>
<show>
  <name>[% show.name | html %]</name>
  <creator>[% show.creator | html %]</creator>
  <seasons>
    [%- FOREACH season = show.seasons %]
    <season number="[% loop.count %]"
      year="[% season.year %]">
      <cast>
      [%- FOREACH part = season.regulars %]
      <regular>
        <character>[% part.character | html %]</character>
        <actor>[% part.actor | html %]</actor>
      </regular>
      [%- END %]
      </cast>

      <episodes>
      [%- FOREACH episode = season.episodes %]
        <episode number="[% loop.count %]"
                 date="[% episode.date %]">
          <name>[% episode.name | html %]</name>
          <summary>[% episode.summary | html %]</summary>
        </episode>
      [%- END %]
```

```
      </episodes>
    </season>
    [% END -%]
  </seasons>
</show>
```

This template takes one parameter, name, which can be passed in on the command line, so we can create a document for *Buffy the Vampire Slayer* using *tpage* like this:

```
$ tpage --define name='Buffy the Vampire Slayer' show.tt > show.xml
```

Example 10-5 shows the XML created. Repeated sections have been replaced with ellipses.

Example 10-5. XML document describing Buffy

```
<?xml version="1.0"?>
<show>
  <name>Buffy the Vampire Slayer</name>
  <creator>Joss Whedon</creator>
  <seasons>
    <season number="1"
            year="1997">
      <cast>
        <regular>
          <character>Buffy Summers</character>
          <actor>Sarah Michelle Gellar</actor>
        </regular>
        <regular>
          <character>Xander Harris</character>
          <actor>Nicholas Brendon</actor>
        </regular>

      ...

      </cast>

      <episodes>
        <episode number="1"
                 date="00:00:00 10-03-1997">
          <name>Welcome to the Hellmouth</name>
          <summary>Buffy Summers moves to Sunnydale</summary>
        </episode>
        <episode number="2"
```

```
                date="00:00:00 17-03-1997">
          <name>The Harvest</name>
          <summary>The Master plans to escape by harvesting people</summary>
        </episode>

...

      </episodes>
    </season>

...

  </seasons>
</show>
```

The template itself doesn't do anything complex. It simply uses access methods on the TVShow object to get the data that it needs. Notice that it uses the Date plugin to format the date and the loop.count variable to create the season and episode numbers.

Notice also that anywhere we are displaying text that could possibly include characters that have a special meaning in XML (&, <, >, or "), we use the html filter to convert these characters into their equivalent XML entity (&, <, >, and ", respectively).

10.3 Processing RSS Files with XML.RSS

Before we start looking at using the Template Toolkit to process arbitrary XML documents, let's take a look at a plugin that can be used to handle an industry-standard XML format: RSS.

RSS[1] is a method that web sites can use to exchange headlines and other data with each other. Web sites can produce RSS files that other web sites can periodically download and process. These files contain information that the subscriber web sites can display along with links to more detailed information on the publisher's web site. This gives the subscribers a relatively simple way to have frequently updated information on their web sites. A good example of this concept are the "slashboxes" that appear on the front page of http://slashdot.org/. You can get more information about RSS from *Content Syndication with RSS* by Ben Hammersley (O'Reilly).

> [1] RSS stands for Rich Site Summary, although exact translations of the abbreviation seem to vary on a daily basis.

An RSS file consists of a small number of tags that describe the web site that produced the file, together with a list of items. Example 10-6 is a sample RSS file. It is taken from CPAN and lists the most recent module uploads. You can see the most recent version of this file at http://search.cpan.org/rss/search.rss. We've removed all but two of the modules from the file to keep the example to a manageable size.

Example 10-6. Example RSS file from CPAN

```
<rss version="0.91">
<channel>
  <title>search.cpan.org</title>
  <link>http://search.cpan.org</link>
  <description>The CPAN search site</description>
  <language>en</language>
    <image>
      <title>searchDOTcpan</title>
      <url>http://search.cpan.org/s/img/cpanrdf.gif</url>
      <link>http://search.cpan.org</link>
      <width>88</width>
      <height>31</height>
      <description>All Modules, All the time</description>
  </image>
  <item>
    <title>DateTime-Format-Builder-0.62</title>
    <link>http://search.cpan.org/author/SPOON/DateTime-Format-Builder-0.62</link>
  </item>
  <item>
    <title>VCS-Lite-0.04</title>
    <link>http://search.cpan.org/author/IVORW/VCS-Lite-0.04</link>
  </item>
</channel>
</rss>
```

The structure of this file is easy to understand. The <channel> element contains a number of details about the web site providing the file in the <title>, <link>, <description>, and <language> tags. Then we see the <image> tag, which contains details of an image that we can use to illustrate our display of the information. Following this are a number of <item> tags, each of which includes information about one recently uploaded CPAN module.

The Template Toolkit's support for RSS is provided by Template::Plugin::XML::RSS, which is, in turn, a thin wrapper round Jonathan Eisenzopf's XML::RSS.

The RSS plugin makes it very simple to use RSS files in your templates. To use it, you need to add the line:

```
[% USE rss = XML.RSS(rssfile) %]
```

where rssfile is a variable that is set to the filename of the RSS file you want to use. You can then access individual items from the file using access methods on the rss object. Here is a very simple template to extract a list of the newest modules:

```
[% rss.channel.title -%]


[%- FOREACH item = rss.items %]
* [% item.title -%]
[% END %]
```

It's only a little more complex to build an HTML page, as shown in Example 10-7.

Example 10-7. Template to build HTML from an RSS file

```
[% USE rss = XML.RSS(rssfile) -%]
<html>
  <head>
    <title>[% rss.channel.title | html %]</title>
  </head>
  <body>
    <h1>[% rss.channel.title | html%]</h1>
    <p><a href="[% rss.image.link | html %]"><img
         src="[% rss.image.url | html %]"
         title="[% rss.image.title | html %]"
         alt="[% rss.image.title | html %]" /></a></p>
    <ul>
    [%- FOREACH item = rss.items %]
      <li><a href="[% item.link | html %]">[% item.title |html %]</a></li>
    [% END %]
    </ul>
  </body>
</html>
```

Notice that, as with the XML document we produced in the previous section, any text displayed is passed through the html filter to turn dangerous characters into HTML entities.

From processing one RSS file link, it's easy to move to processing a number of them on one page to create your own news page.

There is one slight complication with this scenario. You will find a number of different versions of the RSS file on the Internet. You will come across Versions 0.91, 0.92, 1.0, and 2.0.

The simple templates we've shown up to now will work with all versions equally well, but Versions 1.0 and 2.0 have a number of extensions that allow them to contain more information. The extensions in Version 1.0 are incompatible with those in 2.0. Luckily, the XML::RSS plugin gives us access to the version attribute from the RSS file, so our templates can make intelligent decisions on what data to expect to find.

For more details on support of the extensions to RSS 1.0 and 2.0, see the documentation for XML::RSS at http://search.cpan.org/dist/XML-RSS/.

10.4 Processing XML Documents with XML.DOM

There are a number of standards for XML document processing. One of the most popular is the DOM. The Template Toolkit supports this method through the plugin Template::Plugin::XML::DOM, which is, in turn, a thin wrapper around the XML::DOM module written by Enno Derksen.

Because the DOM is a mature standard, there are stable implementations of it in many languages. For this reason, it is very popular with programmers who often switch between different languages. XML::DOM parses the XML document into a tree structure that you can then query using a large set of defined method calls.

To demonstrate the use of the XML.DOM plugin, let's go back to the TV show XML document that we created earlier in this chapter. Example 10-8 shows a basic template that will transform that XML into an HTML page that describes a particular TV show.

Example 10-8. Creating HTML from XML using Template::Plugin::XML::DOM

```
[% USE date (format = '%d %b %Y') -%]
[% USE dom = XML.DOM;
      show = dom.parse('show.xml');
      name = show.getElementsByTagName('name').0.getFirstChild.getNodeValue
-%]
<html>
  <head>
    <title>[% name | html %]</title>
  </head>
  <body>
    <ul>
    [%- FOREACH season = show.getElementsByTagName('season');
              number = season.getAttribute('number') %]
      <li><a href="#season[% number %]">Season [% number %]</a></li>
    [% END -%]
    </ul>

    <h1>[% name | html %]</h1>
      <p>
        Created by
        [% show.getElementsByTagName('creator').getFirstChild.getNodeValue
         | html
         %]
      </p>

    [% FOREACH season = show.getElementsByTagName('season');
             number = season.getAttribute('number') -%]
    <h2><a name="season[% number %]">Season [% number %]</a>
```

```
    ([% season.getAttribute('year') %])</h2>

    <h3>Regular Cast</h3>
    <ul>
    [% FOREACH part = season.getElementsByTagName('regular', 1) -%]
    <li><b>[% part.getElementsByTagName('actor').getFirstChild.getNodeValue
            | html %]</b> as
      <i>[% part.getElementsByTagName('character').getFirstChild.getNodeValue
            | html %]</i></li>
    [%- END %]
    </ul>

    <h3>Episodes</h3>
    [%- FOREACH episode = season.getElementsByTagName('episode',1) %]
    <h4>[% episode.getAttribute('number') %] -
    [% episode.getElementsByTagName('name').getFirstChild.getNodeValue
         | html %]</h4>
    <p>
      <i>First broadcast
      [% date.format(episode.getAttribute('date')) %]</i><br />
      [% episode.getElementsByTagName('summary',1).getFirstChild.getNodeValue
         | html %]
    </p>
    [% END %]
    [% END %]
  </body>
</html>
```

The first thing to notice is that we parse the XML document in two stages:

```
[% USE dom = XML.DOM;
   show = dom.parse('show.xml') %]
```

On the first line, we create a DOM parser object called dom; on the second line, we use that object to parse our input file and create a DOM tree that we store in the variable show. We can then call various XML::DOM methods on this object to extract information about the show. You'll notice that you will often need to string several method calls together to get the information that you need. For example, to get the name of the show, we use the expression:

```
name = show.getElementsByTagName('name').0.getFirstChild.getNodeValue
```

The method getElementsByTagName returns a list of all of the elements that are children of the show element and have the name name. We then take the first node from that list (using the index 0) and get the first child of that node. This will be the text node that contains the name of the show. We can then use getNodeValue to get the value (i.e., the text) of that node.

As always, when we display any text extracted from the XML document, we pass it through the html filter to convert dangerous characters to their HTML entity equivalents.

The output from this code is shown in Example 10-9.

Example 10-9. HTML created from XML using Template::Plugin::XML::DOM

```
<html>
  <head>
    <title>Buffy the Vampire Slayer</title>
  </head>
  <body>
    <ul>
      <li><a href="#season1">Season 1</a></li>
    </ul>

    <h1>Buffy the Vampire Slayer</h1>
    <p>
      Created by
      Joss Whedon
    </p>

    <h2><a name="season1">Season 1</a>
      (1997)</h2>

    <h3>Regular Cast</h3>
    <ul>
      <li><b>Sarah Michelle Gellar</b> as
        <i>Buffy Summers</i></li>
      <li><b>Nicholas Brendon</b> as
        <i>Xander Harris</i></li>
    </ul>

    <h3>Episodes</h3>
    <h4>1 -
      Welcome to the Hellmouth</h4>
    <p>
      <i>First broadcast
        10 Mar 1997</i><br />
        Buffy Summers moves to Sunnydale
    </p>

    <h4>2 -
      The Harvest</h4>
```

```
    <p>

      <i>First broadcast

        17 Mar 1997</i><br />

        The Master plans to escape by harvesting people

    </p>

  </body>

</html>
```

You can get more details on using the DOM from the Template Toolkit by reading the module documentation for Template::Plugin::XML::DOM (at http://www.template-toolkit.org/docs/plain/Modules/Template/Plugin/XML/DOM.html) and XML::DOM (at http://search.cpan.org/dist/XML-DOM/). There is more information about the DOM standard in *XML in a Nutshell* by Elliotte Rusty Harold and W. Scott Means (O'Reilly).

As you can see, using the DOM to extract data from an XML document can get a little long-winded. Luckily, there are other ways to handle XML documents in the Template Toolkit. In the next section, we will look at another.

10.5 Processing XML Documents with XML.XPath

Another common standard for extracting data from XML documents is called *XPath*. XPath is structured vaguely like a filesystem path: consecutive elements are joined with a forward slash (/), beginning at the root, and each element in the path is nested below the previous. The XPath statement:

```
/html/head/title/text( )
```

retrieves "Welcome to Foo.com" from the following XML:

```
<html>
  <head>
    <title>Welcome to Foo.com</title>
  </head>
</html>
```

The Template Toolkit has support for XPath via the XML.XPath plugin, which wraps around Matt Sergeant's excellent XML::XPath module, available from CPAN (see http://search.cpan.org/dist/XML-XPath/). The XML.XPath plugin is given either the name of an XML document or a string containing XML.

Example 10-10 shows a template that uses the XPath plugin to create an HTML page from our XML file containing information about *Buffy the Vampire Slayer*. This is identical to the one we created in the previous section using the DOM (see Example 10-9).

Example 10-10. Creating HTML from XML using Template::Plugin::XML::XPath

```
[% USE date (format = '%d %b %Y') -%]
[% USE show = XML.XPath('show.xml') -%]
[% name = show.findvalue('/show/name/text( )') -%]
<html>
  <head>
    <title>[% name | html %]</title>
  </head>
  <body>
    <ul>
    [%- FOREACH season = show.findnodes('/show/seasons/season');
             number = season.findvalue('@number') %]
      <li><a href="#season[% number %]">Season [% number %]</a></li>
    [% END -%]
    </ul>

    <h1>[% name | html %]</h1>
    <p>
      Created by
    [% show.findvalue('show/creator/text( )') | html %]
```

```
    </p>

  [% FOREACH season = show.findnodes('/show/seasons/season');
            number = season.findvalue('@number') -%]
    <h2><a name="season[% number %]">Season [% number %]</a>
      ([% season.findvalue('@year') %])</h2>

    <h3>Regular Cast</h3>
    <ul>
    [% FOREACH part = season.findnodes('cast/regular') -%]
      <li><b>[% part.findvalue('actor/text( )') | html %]</b> as
        <i>[% part.findvalue('character/text( )') | html %]</i></li>
    [%- END %]
    </ul>

    <h3>Episodes</h3>
  [% FOREACH episode = season.findnodes('episodes/episode') -%]
    <h4>[% episode.findvalue('@number') %] -
        [% episode.findvalue('name/text( )') | html %]</h4>

    <p>
      <i>First broadcast
        [% date.format(episode.findvalue('@date')) %]</i><br />
        [% episode.findvalue('summary/text( )') | html %]
    </p>
  [% END %]
  [% END %]
  </body>
</html>
```

We are basically using three methods from the XML.XPath plugin. The line:

```
[% USE show = XML.XPath('show.xml') -%]
```

creates a new XML::XPath object based on the file *show.xml*. This object is a tree structure that models the XML structure of the XML document. We can then use the methods findvalue and findnodes to run XPath queries against this object. findvalue takes an XPath expression that will return a single value and returns the result of evaluating that expression. For example, we use:

```
[% name = show.findvalue('/show/name/text( )') -%]
```

to get the name of the show from the current document. The XPath query translates as "get the text for contained in the <name> element, which is a child of the <show> element, which is a child of the root." Any kind of XPath expression can be used. For example, we use @number to get the number attribute of the current node (which just happens to be an episode node at that point).

The findnode method is used to loop over a list of nodes. For example, we use:

```
[% FOREACH season = show.findnodes('/show/seasons/season') %]
```

to get each <season> node that is contained in the document, and use:

```
[% FOREACH episode = season.findnodes('episodes/episode') %]
```

to get each episode in a season. Notice that as findnodes returns a list of nodes, we use a variable to store each node in return as we work our way across the loop. These nodes are also XML::XPath objects and we can therefore run XPath queries against them in exactly the same way as we can with the original show object.

The current node that we are working from is called the *context node*. Continuing the filesystem analogy that we mentioned earlier, using a context node is like changing your current directory. Any XPath query that doesn't start with / is taken to be relative to your context node in the same way as a directory path that doesn't start with / is taken to be relative to your current directory. Any XPath query that starts with / is taken to be relative to the root node in the same way as a directory path that starts with / is taken as relative to the root directory.

You can get more details on using XPath from the Template Toolkit by reading the module documentation for Template::Plugin::XML::XPath (at http://www.template-toolkit.org/docs/plain/Modules/Template/Plugin/XML/Path.html) and XML::XPath (at http://search.cpan.org/dist/XML-XPath/). There is more information about the XPath standard in *XML in a Nutshell* by Elliotte Rusty Harold and W. Scott Means.

10.6 Processing XML Documents with XML.LibXML

All of the XML processors that we have seen up to now are based on the Perl module XML::Parser, which is, in turn, based on James Clark's *expat* XML parser. However, *expat* doesn't have support for newer XML features such as namespaces, so another parser has emerged as the first choice for many XML processing tasks. It is called *libxml2*, and you can find more details about it at http://www.libxml.org/.

Perl has a module, XML::LibXML, that gives access to the *libxml2* API, and Mark Fowler has written Template::Plugin::XML::LibXML, which allows the API to be used from the Template Toolkit. Both of these modules can be downloaded from CPAN at http://search.cpan.org/dist/XML-LibXML/ and http://search.cpan.org/Template-Plugin-XML-LibXML/, respectively.

libxml2 contains support for both DOM and XPath, so both of the previous examples will work almost unchanged. You will just need to alter the lines that load and parse the XML document.

10.7 Using Views to Transform XML Content

The XML processing methods that we have seen so far are very useful for *data-centric* XML documents. These are documents whose structure is very well-defined. This type of file is commonly seen when the file is modeling some kind of data structure, and is usually used for transferring data between different systems. The TV show example was a good example of this, as the relationships between the various data items in the document were well understood and unlikely to change.

There is another type of XML file, known as *narrative-centric*. In these files, the data is less well structured. A good example of this kind of document is a book. Although a book will have some high-level structure (table of contents, chapters, appendixes, and index), once you get down to the text in a chapter, the structure is much less defined. A paragraph can contain italic text, bold text, references to footnotes, URLs, and any number of other types of text, all of which will need to be processed differently.

While it is possible to handle these kinds of documents using the techniques we have seen previously, using the *VIEW* directive makes it far easier to process narrative-centric XML.

Example 10-11 shows a narrative-centric XML document.

Example 10-11. A narrative-centric XML document

```
<faq>
  <qna id="q1">
    <question>
      What is the ultimate answer to life, the universe and everything?
    </question>

    <answer author="Deep Thought">
      <para>42</para>
      <note>The problem may well be that you don't <i>actually</i>
         know what the question is!</note>
    </answer>
  </qna>

  <qna id="q2">
    <question>
      Where shall we have lunch?
    </question>

    <answer author="Milliways Marketing Dept.">
      <para>Have you considered <froody>Milliways</froody>, the restaurant
        at the end of the universe.</para>

      <quote>If you've done six impossible things today then why
        not top it off with dinner at Milliways?</quote>
```

```
    </answer>
  </qna>
</faq>
```

Notice that while the higher levels of the document are well structured, once you get into the answer tag, the text is unstructured. The para, note, and quote tags are used interchangeably, and other tags are used as well—you can see i and froody.

To process this file, we will create a *VIEW* called faq_html that will convert the FAQ to HTML. For our first attempt, we will create a "do nothing" view that will simply pass the document through unchanged. This view is shown in Example 10-12.

Example 10-12. faq_view1

```
[% VIEW faq_html
  notfound='passthru';

  BLOCK text;
    item;
  END ;

  BLOCK passthru;
    item.starttag;
    item.content(view);
    item.endtag;
  END;
END
%]
```

The *[% VIEW %]* directive defines a block that can contain other named blocks. In this *VIEW*, we defined two blocks. The first is called text. This is the default name for a block that will be called to process text nodes from the document. Our text block is simple and just displays the current item. Note that from within a VIEW template, the current node is available in the item variable and the current view is in the view variable.

The other block we defined is the block that is called if no matching block is found for a node. This is defined using the notfound parameter to the *VIEW* directive. Our passthru block displays the start and end tags for the node, and between them it calls the current node's content method, passing it the current view. The content method finds all of the current node's child nodes and displays them using the given view. This is an important method. If you want child nodes to be processed, your template must call it.

In order to use this template, we need to have a parsed XML document. VIEWs work well with any of the XML modules that we have seen before, but support for the XPath plugin is the most advanced. We can create and process an XML::XPath object with code like this:

```
[% USE doc=XML.XPath(file => 'faq.xml');
   node = doc.findnodes('/faq');
   faq_html.print(node) %]
```

Calling the print method on the VIEW and passing it the starting node starts the VIEW processing the document. Each type of node in the document is handled by the block with the same name. Any type of node that doesn't match a block in the VIEW is handled by the notfound block.

Currently our template has no named blocks, so all nodes are handled by the notfound block. We can add blocks that handle any nodes that need more than this default processing. Example 10-13 fills in processing for a number of tags.

Example 10-13. A more complex view

```
[% VIEW faq_html notfound='xmlstring' %]

[% BLOCK faq -%]
<h1>Frequently Asked Questions</h1>
[%- item.content(view) %]
[%- END %]

[% BLOCK question -%]
<h2>[% item.content(view) %]</h2>
[%- END %]

[% BLOCK answer %]
[% item.content(view) %]
<p>Answer by [% item.getAttribute('author') %]</p>
[% END %]

[% BLOCK para -%]
<p>[% item.content(view) %]</p>
[%- END %]

[% BLOCK note -%]
<p>Note: [% item.content(view) %]</p>
[%- END %]

[% BLOCK quote -%]
<blockquote><i>[% item.content(view) %]</i></blockquote>
[%- END %]

[% BLOCK qna;
  item.content(view);
  END;

  BLOCK text;
    item;
  END;

  BLOCK xmlstring;
    item.starttag;
```

```
    item.content(view);

    item.endtag;

  END

%]

[% END %]



[% USE doc = XML.XPath(file => 'faq.xml');

   node = doc.findnodes('/faq');

   faq_html.print(node)

%]
```

We should note a couple of points. First, we have created a block for the qna node, which does nothing but process its children. This is because if we left it to the default block, the opening and closing qna tags would be displayed, and we don't want that. Second, we haven't defined a block for the i tag. This is because we are happy for it to pass through unchanged, so it becomes part of the HTML page that is created.

Our input document also contains a froody tag. Currently this tag is passed through untouched (and presumably is ignored by the browser that displays the finished page). But when the management of Milliway's complain that their text should be displayed in a certain manner, it will be simple for us to add a block that handles it. For example:

```
[% BLOCK froody -%]

<font size="20" color="red"><i>[% item.content(view) %]</i></font>

[%- END %]
```

It is this extensibility that makes VIEW a perfect tool for processing narrative-centric XML documents. It is very simple to add processing for new tags, and it doesn't matter where they appear in the document structure.

Chapter 10. XML

XML is becoming one of the most ubiquitous data formats. It is used for both data storage and data exchange. The Template Toolkit can be used to both create XML documents and convert them into other formats.

In this chapter, we'll take a look at some of the tools that the Template Toolkit provides for working with XML. We show how to populate template variables with fields from XML, how to generate XML, how to process RSS, how to extract information with the Document Object Model (DOM) and XPath, and even how to use XML transforms.

Before we get into some of the more complex tools for processing XML, let's start simply by looking at Template::Plugin::XML::Simple, which allows us to take a very simple approach to our XML.

11.1 Getting Started

A few basic tasks need to be done when starting out a project for a Template Toolkit-driven web site. The first thing is to create somewhere for the project files to go. It's a good idea to keep everything related to the project in one place. If all the files are located in subdirectories of one common parent directory, the entire project can easily be relocated to another server, or perhaps to another directory on the same machine. It is much harder to keep track of files when they are dotted around a filesystem.

For this project, we will generate static HTML pages from templates. All the output files will be written to an *html* subdirectory of the project directory. From here they can be accessed via an appropriately configured web server. We'll be looking at a simple configuration for the Apache web server that demonstrates this.

The tool of choice for this kind of project is *ttree*. It also needs a configuration file detailing the various directories and other Template Toolkit options in effect. In this file, we can also specify which templates should be used as headers, footers, or wrappers to be automatically applied to each generated page. With these configuration files and standard templates in place, we can then begin to generate HTML pages.

So let's walk through the complete process, from creating the project directory to generating the first HTML page.

11.1.1 Directory Structure

The first task is to create a directory structure for the web site project. We'll be using */home/dent/web/ttbook* as the base directory in these examples:

```
$ mkdir /home/dent/web/ttbook

$ cd /home/dent/web/ttbook
```

Some further subdirectories are required underneath the new project directory:

```
$ mkdir bin etc templates html images
```

The directories follow a fairly standard naming convention. Here *bin* will be used to store executable programs or scripts to assist in building the site or performing other housekeeping tasks. The *etc* directory is for configuration and other miscellaneous files. The *templates* directory is for source templates from which HTML pages are generated. These are written to the *html* directory from where they are ready to be accessed by a web server, along with any images or other binary files for the site, stored under the *images* directory.

Two more subdirectories are required under the *templates* directory:

```
$ mkdir templates/src templates/lib
```

The *templates* directory is where most of the action takes places. The *templates/src* directory contains the source templates for the pages of the web site, or more generally, the site *content*. The *templates/lib* directory alongside it contains the library of general-purpose template components: headers, footers, menus, and so on. These typically relate to the user interface or *presentation* aspects.

You'll need to create further directories for content and component templates as we progress through the examples in this chapter. We'll assume from now on that you can do that without us having to tell you.

One final thing to note is that the names of templates cited in `INCLUDE`, `PROCESS`, and `WRAPPER` directives in these examples relate to files in the *templates/lib* directory, as defined in the `lib` option in *etc/ttree.cfg*. So a directive such as `[% PROCESS menu/item %]`, for example, refers to the *templates/lib/menu/item* template.

11.1.2 Web Server Configuration

The Template Toolkit isn't tied into any particular web server. At the simplest level, it is just a tool for generating content that can be read directly by a file editor or web browser, or can be served across a network by a web server. It operates independent of any delivery mechanism.

We will be using the Apache web server in these examples. A sample configuration file for Apache is shown in Example 11-1. This file should be created in the project *etc* directory.

Example 11-1. etc/httpd.conf

```
Alias /ttbook/images/   /home/dent/web/ttbook/images/

Alias /ttbook/          /home/dent/web/ttbook/html/
```

```
<Directory /home/dent/web/ttbook/>
    Options MultiViews Indexes FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

You will also need to edit your main Apache *httpd.conf* file (typically */usr/local/apache/conf/httpd.conf* or */etc/httpd.conf*) to Include the project configuration file. Example 11-2 shows the relevant line that is added for our configuration file, */home/dent/web/ttbook/etc/httpd.conf*.

Example 11-2. Addition to Apache httpd.conf configuration file

```
Include /home/dent/web/ttbook/etc/httpd.conf
```

You will need to restart Apache for these changes to take effect. For an Apache installation in */usr/local/apache*, the command would be as follows:

```
# /usr/local/apache/bin/apachectl restart
```

Another approach is to use symbolic links from an existing location that is already visible to the web server. For example, if the directory */home/dent/public_html/* can be accessed via the URL *http://localhost/~dent/*, you can create a symbolic link from here to the project *html* directory. On a Unix machine, the relevant command would be something like this:

```
$ cd /home/dent/public_html
$ ln -s /home/dent/web/ttbook/html ttbook
```

The *html* directory would then be accessible via the web server URL *http://localhost/~dent/ttbook/*.

Be warned that Apache doesn't follow symbolic links by default, so you'll need to add FollowSymLinks to the relevant section of the *httpd.conf* configuration file if you choose this approach:

```
<Directory /home/*/public_html>
    ...
    Options FollowSymLinks
    ...
</Directory>
```

With this directive in place, you can also use a symbolic link in the *html* directory to make the *images* directory accessible:

```
$ cd /home/dent/web/ttbook/html
$ ln -s ../images images
```

If you're not using Apache, you'll need to consult the documentation for your own web server to find out how to make the contents of the *html* and *images* directories accessible.

We'll assume in the following examples that the root document URL is */ttbook/* and the root images URL is */ttbook/images/*, in both cases assuming the default host, *http://localhost/*.

11.1.3 ttree Configuration

We need to provide a configuration file to tell *ttree* everything it needs to know to build the site content. Example 11-3 shows the complete file.

Example 11-3. etc/ttree.cfg

```
# directories
src  = /home/dent/web/ttbook/templates/src
lib  = /home/dent/web/ttbook/templates/lib
dest = /home/dent/web/ttbook/html

# copy images and other binary files
copy = \.(png|gif|jpg|pdf)$

# ignore CVS, RCS, and Emacs temporary files
ignore = \b(CVS|RCS)\b
ignore = ^#

# misc options
verbose
recurse
recursion

# TT options
pre_process = config/main
wrapper     = site/wrapper

# define some location variables
define rootdir = /home/dent/web/ttbook
define rooturl = /ttbook
define debug   = 0
```

The configuration file is very similar to the example we saw in Chapter 2. The first section defines the three important template directories:

```
# directories
src  = /home/dent/web/ttbook/templates/src
lib  = /home/dent/web/ttbook/templates/lib
dest = /home/dent/web/ttbook/html
```

The src directory contains the source templates for HTML pages. Each is processed by *ttree*, and the output is written to the corresponding file in the dest directory. The lib directory contains the library of various template components that don't comprise complete page templates in their own right. This directory is added to the INCLUDE_PATH option that *ttree* passes to the Template Toolkit. You can specify multiple lib directories in the configuration file, and each will be added to the INCLUDE_PATH in the order defined.

For now we plan to keep all images under the *images* directory, separate from the source templates in *templates/src*. However, there may be occasions when we want to put an image or other binary file in the same directory as an HTML page. To accommodate this, we set the copy option to a regular expression matching any filename extensions that indicate files that should be copied directly from *templates/src* to *html* without being processed through the Template Toolkit:

```
# copy images and other binary files
copy = \.(png|gif|jpg|pdf)$
```

We can also tell *tree* to look out for certain files that should be completely ignored—in this case, any CVS or RCS files that we may be using for version control, and also any temporary files that our favorite editor may have left lying around:

```
# ignore CVS, RCS, and Emacs temporary files

ignore = \b(CVS|RCS)\b

ignore = ^#
```

The next section sets some basic *ttree* flags:

```
# misc options

verbose

recurse

recursion
```

The first is verbose, which enables various useful messages so that we can see what's going on as *ttree* is doing its work. The second is recurse, which tells *ttree* to recurse into any directories it finds under the src directory and process any templates and further subdirectories it finds therein. The last option, recursion, is confusingly similar to recurse but serves a slightly different purpose. This tells the Template Toolkit that it's OK for a template to recursively process itself. Don't worry if you're not sure what that means right now. We're going to be using this feature later on when it comes to building a menu for the site, so all will become clear.

The next section defines two options that are passed to the Template Toolkit as the PRE_PROCESS and WRAPPER options:

```
# TT options

pre_process = config/main

wrapper     = site/wrapper
```

The pre_process option denotes that the *config/main* template should be preprocessed before each source page template. The wrapper option gives the name of a template that is used to provide a wrapper around the generated page output—in this case, to add HTML headers, footers, and any other user interface elements common to all pages in the site.

The final section defines two template variables that indicate the root directory for the project and the root URL for accessing the pages. The third defines a debug flag, which we'll leave disabled for now:

```
# define some location variables

define rootdir = /home/dent/web/ttbook

define rooturl = /ttbook/index.html

define debug   = 0
```

It is common (and sensible) practice to develop and test a web site offline, uploading it to its final URL only when it is finished and ready for public consumption. The only drawback to this is that the URLs you use to access pages under development will be different from those you use when the site goes live. One workaround to this problem is to use relative URLs when linking between pages. This approach works fine for small and simple sites but doesn't scale very well to larger, more complex sites, which can become more fragile when held together by relative links.

A better approach is to use a variable such as rooturl to define a root URL from which all other relative URLs in the site are constructed. If we need to relocate our site to be served under a different URL, we need only change this value and have *ttree* rebuild the site.

We'll see how this works in practice when we define some URLs a little later on in this chapter.

11.1.4 Simple pre_process and wrapper Templates

We now need to provide the pre_process and wrapper templates that were named in the *etc/ttree.cfg* configuration file.

For now we can just use some simple templates to get started and test that everything is working. The configuration template is shown in Example 11-4. It sets a single variable, msg. We will be displaying this value in a test page later on to demonstrate that the template is being preprocessed and the value correctly set.

Example 11-4. templates/lib/config/main

```
[% message = 'Hello World' -%]
```

The wrapper template displays the content inside a minimal set of HTML elements required for a valid HTML page (see Example 11-5).

Example 11-5. templates/lib/site/wrapper

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

  <head>

    <title>[% template.title %]</title>

  </head>

  <body>

    [% content %]

  </body>

</html>
```

We'll start off by defining a title for each page in a META tag in the source template. In the *wrapper* template, this value is accessed as the template.title variable.

11.1.5 Creating the Build Script

Building the site is now a simple matter of invoking *ttree* using the -f option to tell it where to find the configuration file:

```
$ ttree -f /home/dent/web/ttbook/etc/ttree.cfg
```

The configuration file can be specified using an absolute filename as shown earlier, or a relative filename as shown in the following examples. Note the leading dot character (.) on the first example, which is required.

```
$ cd ~/web/ttbook

$ ttree -f ./etc/ttree.cfg     # OK

$ cd src

$ ttree -f ../etc/ttree.cfg    # OK
```

This can get a little tiresome when you have to type it several dozen times in a day, especially if the path to the configuration file is long and complicated. So to make life a little easier, we create a simple build script that calls *ttree* with the right -f option along with any other command-line arguments we specify, as shown in Example 11-6.

Example 11-6. bin/build

```
    ttree -f /home/abw/web/ttbook/etc/ttree.cfg $@
```

The build script is just a thin wrapper of convenience around *ttree* (for now). You can continue to use any of the usual *ttree* command-line options. For example:

```
$ bin/build              # build any modified pages

$ bin/build -a           # build all pages

$ bin/build index.html   # build just this page

$ bin/build -h           # show help
```

See Chapter 2 for further examples of using a build script.

11.1.6 A First HTML Page

With our basic presentation system in place, we can now start to create content for the web site. Each HTML page starts off as a source template in *templates/src*. All the headers, footers, menus and other user interface components are added automatically, so these templates need to provide only the core content for the page.

It is traditional to begin any demonstration such as this with the universal greeting to all of humanity. Example 11-7 shows a page template that displays the familiar "Hello World" message.

Example 11-7. templates/src/index.html

```
[% META title = 'Template Toolkit Test' %]


<p>

  This is the index page.  Testing!  Testing!

  <br />

  The message is '[% message %]'.

</p>
```

The page contains two directives. The first defines a title in a META tag. This value will then be displayed in the HTML head tag by the *templates/lib/site/wrapper* template that we defined earlier. The title is accessed, as are all META items, through the template variable—e.g., template.title.

The second directive prints the value of the message variable that we defined in the preprocessed *config/main* template.

Run *bin/build* to process the source template and generate the HTML page:

```
$ bin/build


ttree 2.63 (Template Toolkit version 2.10)


      Source: /home/dent/web/ttbook/templates/src

 Destination: /home/dent/web/ttbook/html

Include Path: [ /home/dent/web/ttbook/templates/lib ]

      Ignore: [ \b(CVS|RCS)\b, ^# ]

        Copy: [ \.(png|gif|jpg|pdf)$ ]

      Accept: [ * ]


  + index.html
```

The + to the left of index.html on the last line indicates that the file was processed successfully. This creates an *index.html* file in the *html* directory that looks like Example 11-8.

Example 11-8. html/index.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

  <head>

    <title>Template Toolkit Test</title>

  </head>

  <body>

    <p>

      This is the index page.  Testing!  Testing!

      <br />

      The message is 'Hello World'.
```

```
  </p>

 </body>

</html>
```

The source template has been processed and the [% message %] directive resolved to provide the familiar Hello World greeting. The page content has also been enclosed in the HTML wrapper template with the title of the page template (template.title) correctly inserted.

If your web server configuration is working as expected, you should now also be able to view this page as *index.html* under the root URL you specified (e.g., */ttbook/index.html*).

11.1.7 Automating the Project Configuration Process

If you take a look back over the files that we've created for the purposes of project administration—*bin/build*, *etc/ttree.cfg*, and *etc/httpd.conf*—you'll notice that all three reference the project root directory, */home/dent/web/ttbook*, and two use the base URL, */ttbook*. As when we want to move the project to another directory or URL, we need only edit these three files, and everything else should fall into place as part of the build process.

Three files may not sound like many, but that number will most likely grow as you add more functionality to your system. Sooner or later you'll relocate the site and forget to update one of the critical files. Much head scratching will ensue while you try to figure out why the site isn't building properly or the pages aren't being displayed.

If alarm bells aren't already ringing in your head, they should be because this is a perfect application area for some template processing. We said earlier that the Template Toolkit wasn't just for processing HTML, and this is a great example of what we mean. Rather than hardcoding a directory and URL in several configuration files, we can define them as templates, and have these and any other project-related variables inserted automatically to construct the build script and configuration files for us.

Here's how we do it. First, we create a directory for storing the skeleton templates for our project files. We'll call this directory *skeleton* to avoid confusing it with our HTML templates in *templates*. Under this directory, we also add *bin* and *etc* sub-directories.

```
$ cd /home/dent/web/ttbook

$ mkdir skeleton

$ mkdir skeleton/bin skeleton/etc
```

Copy the files *bin/build*, *etc/ttree.cfg*, and *etc/httpd.conf* (if you're using it, that is) into the relevant *skeleton* directories:

```
$ cp bin/build skeleton/bin

$ cp etc/ttree.cfg skeleton/etc

$ cp etc/httpd.conf skeleton/etc
```

Now use your favorite text editor to peform a global search for the project directory (e.g., */home/dent/web/ttbook*) and replace it with the Template Toolkit directive [% dir %]. Similarly, replace the base URL (e.g., */ttbook*) with [% url %]. Finally, replace the 0 for the debug value defined in *skeleton/etc/ttree.cfg* with [% debug %]. You can use Perl to do this if you prefer, using something like the following incantation:

```
$ perl -pi -e 's{/home/dent/web/ttbook}{[% dir %]}g; \

>           s{/ttbook}{[% url %]}g; \

>           s{(debug\s*=)\s*0}{$1 [% debug %]}' \

>           skeleton/*/*
```

The files should now look like those shown in Examples Example 11-9, Example 11-10, and Example 11-11.

Example 11-9. skeleton/bin/build

```
ttree -f [% dir %]/etc/ttree.cfg $*
```

Example 11-10. skeleton/etc/ttree.cfg

```
# directories
src  = [% dir %]/templates/src
lib  = [% dir %]/templates/lib
dest = [% dir %]/html

# copy images and other binary files
copy = \.(png|gif|jpg|pdf)$

# ignore CVS, RCS, and Emacs temporary files
ignore = \b(CVS|RCS)\b
ignore = ^#

# misc options
verbose
recurse
recursion

# TT options
pre_process = config/main
wrapper     = site/wrapper

# define some location variables
define rootdir = [% dir %]
define rooturl = [% url %]
define debug   = [% debug %]
```

Example 11-11. skeleton/etc/httpd.conf

```
Alias [% url %]/images/   [% dir %]/images/
Alias [% url %]/          [% dir %]/html/

<Directory [% dir %]/>
    Options MultiViews Indexes FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

Now all we need is a configuration script to figure out what the right values should be and process the templates. Another wrapper around *ttree* will do the job nicely, as shown in Example 11-12.

Example 11-12. bin/configure

```
#!/usr/bin/perl -w                                  # -*- perl -*-
#
# configure
#
# This script determines the correct root directory
# for the project (the parent of the 'bin' directory
# in which it is located), prompts for some configuration
# values if not set via command-line options, and then
# calls ttree to process all files under the skeleton
# directory, storing output relative to the project root
# directory (e.g., skeleton/bin/build => bin/build).
#
# Copyright 2003 Andy Wardley.
#
# This is free software distributed under the same terms as Perl.
#

use strict;
use warnings;
use FindBin qw( $Bin );
use Getopt::Std;
local $|=1;

# defaults
my $URL  = '/ttbook';

# get options
our ($opt_d, $opt_u, $opt_y, $opt_h);
getopts('yhdu:');

# display usage and exit on -h
die <<END_USAGE if $opt_h;
usage: configure [options]

options:
   -u url    url for HTML pages  (default: $URL)
   -d debug  set debug flag      (default: 0)
   -y        Accept defaults
```

```
    -h         This help
END_USAGE

# work out where we are in the filesystem
my @dirs = File::Spec->splitdir($Bin);
pop @dirs;  # remove 'bin'
my $dir  = File::Spec->catdir(@dirs);
my $skel = File::Spec->catfile($dir, 'skeleton');

# prompt for root URL
my $url  = prompt('root page URL', $opt_u || $URL);
my $dbg  = prompt('enable debugging?', $opt_d ? 'yes' : 'no')
         =~ /^y(es)?/ ? 1 : 0;

# hand over to ttree
my @args = ( 'ttree',
             '-r', '-p', '-v', '-a',
             '-s', $skel,
             '-d', $dir,
             '--ignore', '\b(CVS|RCS)\b',
             '--define', "dir=$dir",
             '--define', "url=$url",
             '--define', "debug=$dbg",
             @ARGV );

system(@args) =  = 0
    or die "ttree failed: $?\n";

#------------------------------------------------------------------------
# prompt($message, $default)
#
# Prompt user to input value or accept default.
#------------------------------------------------------------------------

sub prompt {
    my ($msg, $def) = @_;
    my $ans = '';
    $def = '' unless defined $def;
```

```
    print "$msg [$def] ";

    if ($opt_y) {    # accept default
        print "$def\n";
    }
    else {           # read user input
        chomp($ans = <STDIN>);
    }

    return length($ans) ? $ans : $def;
}
```

The script first determines the root directory of the project and then prompts the user for the base URL, defaulting to */ttbook*.

```
$ bin/configure
root page URL [/ttbook]
```

It also prompts the user to confirm the debugging option. Answer y or yes to set the debugging option, or press Enter to accept the default, leaving debugging disabled:

```
enable debugging? [no]
```

This flag doesn't have any effect on the Template Toolkit, although there are plenty of others that do. We're just defining another template variable, this time called debug, which we'll be using later.

The script then calls *ttree*, passing the various options required to have it process the files under the *skeleton* directory and copy the generated output into place under the project root directory:

```
ttree 2.63 (Template Toolkit version 2.10)

      Source: /home/dent/web/ttbook/skeleton
 Destination: /home/dent/web/ttbook
Include Path: [  ]
      Ignore: [ \b(CVS|RCS)\b ]
        Copy: [  ]
      Accept: [ * ]

  + bin/build
  + etc/ttree.cfg
  + etc/httpd.conf
```

The output files generated—*bin/build*, *etc/ttree.cfg*, and *etc/httpd.conf*—will contain exactly the same content as they did before. However, we can now easily move the project to a new directory or locate it under a different URL. Instead of editing the configuration files by hand, we let the *bin/configure* script take care of it.

An illustration of this is shown in the first line of the following example. Command-line options are used to define the new root URL (-u) and to accept all defaults (-y). The *bin/configure* script then regenerates the configuration files for the project. The second command then calls on the *bin/build* script to rebuild all the pages in the site (-a) using the new values defined.

```
$ bin/configure -u /newtturl -y
$ bin/build -a
```

Even the Apache configuration file, *etc/httpd.conf*, has been updated to account for the new URL, as shown in Example 11-13.

Example 11-13. etc/httpd.conf

```
Alias /newtturl/images/   /home/abw/web/ttbook/images/

Alias /newtturl/          /home/abw/web/ttbook/html/



<Directory /home/abw/web/ttbook/>

   Options MultiViews Indexes FollowSymLinks

   AllowOverride None

   Order allow,deny

   Allow from all

</Directory>
```

All you need to do is to restart Apache to have it read the new configuration. The web site will then be accessible via the URL *http://localhost/newtturl/*.

11.2 Library Templates

The templates for this project fall into two categories. Each HTML page has a corresponding source template in *templates/src* such as that shown in Example 2-1 in Chapter 2. These are referred to as *page templates* and generally map one-to-one with each static page in the site.

The other templates are *library templates*, also known as *template components*. Rather than defining complete HTML pages, these templates encode smaller chunks of HTML markup or Template Toolkit code to perform one task. We've also seen some simple examples of these in Examples Example 2-3 and Example 2-4. We're going to be looking at these in more detail now.

11.2.1 Configuration Templates

The purpose of the PRE_PROCESS configuration template, *config/main*, is to define any sitewide variables required to specify URLs, colors, images, and anything that we don't want to hardcode in the HTML page content or user interface components.

Rather than define everything in one monolithic configuration file, something that would quickly lead to a poor separation of concerns, a separate *config* directory will be used to contain various different configuration templates, each one representing one particular aspect of the site. These templates are loaded by one master template, *config/main*, shown in Example 11-14, using the PROCESS directive.

Example 11-14. templates/lib/config/main

```
[%  PROCESS config/page
      + config/site
      + config/url
      + config/col
      + config/images
-%]
```

This approach allows you to easily change one configuration file without affecting the others. This is particularly useful when you want to customize a web site to provide different presentation styles, a process known as *branding* or *skinning,* which we will be covering later in this chapter.

Now let us look at each configuration file in turn to find out what they do. The first, *config/page*, defines a page data structure containing various bits of information relating to the current page (i.e., source template) being processed. This is shown in Example 11-15.

Example 11-15. templates/lib/config/page

```
[%  USE Date;

   # define page data structure
   page = {
     file    = template.name
     title   = template.title
     about    = template.about
     type    = template.type or 'html'
     date    = template.date or Date.format(template.modtime)
```

```
   };


-%]
```

We're using the template variable here that references the Template::Document object for the current page template being processed (or about to be processed, given that this is all happening during the preprocess phase). Through the template variable we can access details about the template file itself, including the filename, template.name (specified relative to the *templates/src* directory in this case) and the modification time, template.modtime. Any metadata items defined in META tags within the template are also made available through the template variable—here we look specifically for title, about, and type. We also look for a date item, and otherwise construct human-readable data from the template modification time (template.modtime) formatted using the Date plugin.

The remaining templates define configuration data that relates to the site as a whole. The *config/site* template, shown in Example 11-16, defines a site data structure that contains some miscellaneous items.

Example 11-16. templates/lib/config/site

```
[%  site = {
        name      = 'Template Toolkit Web Site'
        server    = 'http://template-toolkit.org'
        admin     = 'webmaster@template-toolkit.org'
        copyright = '1996-2003 Andy Wardley'
    }
-%]
```

Example 11-17 shows the *config/url* template. This uses the rooturl variable to construct a set of page and section URLs that are stored in the site.url hash. Recall that the value for rooturl is defined as a ttree configuration option in the *etc/ttree.cfg* file.

Example 11-17. templates/lib/config/url

```
[%  site.url = {
        root    = rooturl
        home    = "$rooturl/index.html"
        images  = "$rooturl/images"
        logo    = "$rooturl/images/logo"
        css     = "$rooturl/css"
    }
-%]
```

The *config/col* template defines an rgb hash mapping color names to RGB hex triplets in the format required for HTML pages. This is also aliased to site.rgb. The template then defines a site.col hash that maps various style names to specific rgb colors. This is shown in Example 11-18.

Example 11-18. templates/lib/config/col

```
[%  rgb = {
        white    = '#FFFFFF'
        black    = '#000000'
        red      = '#ED2328'
        orange   = '#F08900'
        skyblue  = '#00AAF0'
```

```
        paleblue = '#80C0F0'
        midblue  = '#6080C0'
        darkblue = '#202060'
        misty    = '#C0C0F0'
        ltgrey   = '#E0E0E0'
        vltgrey  = '#F0F0F0'
    }

    site.rgb = rgb
    site.col = {
        back  = rgb.white
        text  = rgb.black
        link  = rgb.skyblue
        vlink = rgb.midblue
        alink = rgb.red
        mlink = rgb.orange
        line  = rgb.skyblue
        head  = rgb.darkblue
    }
-%]
```

The color names being used here are entirely arbitrary. It should be obvious that you can extend and adapt these and all the other data structures for your own use.

The *config/images* template, shown in Example 11-19, defines a site.image data structure containing some useful information about the logos that we're using in the site in various sizes.

Example 11-19. templates/lib/config/images

```
[%  site.image = {
        logo = {
            large = {
                src    = "$site.url.logo/tt2_180x60.gif"
                alt    = "TT2 Logo"
                width  = 180
                height = 60
            }
            small = {
                src    = "$site.url.logo/tt2_120x40.gif"
                alt    = "TT2 Logo"
                width  = 120
                height = 40
            }
```

```
        }
        name = {
            src    = "$site.url.logo/ttdotorg.gif"
            alt    = "template-toolkit.org"
            width  = 180
            height = 24
        }
    }

    site.logo = site.image.logo.large

-%]
```

The configuration templates collectively define two data structures: site and page. It is a good idea to define as few "top-level" variables like this as possible. The more variables you have, the harder it is to keep track of them, and the more likely you are to overwrite an important piece of predefined configuration data with a temporary or "local" variable of the same name.

Another benefit to this approach is that it allows us to replace the site or page data structures at a later date with alternate implementations. For example, we might decide to define the site data in an XML file, in an SQL database, or as a Perl module. All we have to do is arrange the data in the right format and make it available as the site and page variables, and it will integrate seamlessly into the existing structure.

Finally, defining all your sitewide data in a single site variable makes it easy to use compile-time constant folding at a later date if you need to optimize your templates for efficiency. As described in Chapter 3, the constant folding feature allows you to provide a set of variables in a namespace (constants by default, but you can easily change it to site, for example), which should be resolved once when the template is compiled instead of being resolved each time the template is processed. This can be particularly benefical when generating large amounts of template-driven dynamic content through a web server. It effectively gives each template less work to do each time it is processed by doing some of the work when the template is compiled.

11.2.2 Layout Templates

Now we can start to define the overall look and feel of the web site, using the same techniques that we introduced in Chapter 2.

11.2.2.1 Page wrappers

The wrapper option is used in the *etc/ttree.cfg* file to denote the name of a template that is used to automatically enclose the content generated from each page template. In this, the template is *site/wrapper*, shown in Example 11-20.

Example 11-20. template/lib/site/wrapper

```
[% content WRAPPER site/html + site/layout -%]
```

Two templates are being used to wrap the generated page content. The first and outermost wrapper in this case is *site/html*, shown in Example 11-21.

Example 11-21. templates/lib/site/html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
 <head>
  <title>[% page.title %]</title>
```

```
  <link rel="stylesheet" href="[% site.url.css %]/tt2.css" />
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
  <meta name="robots" content="all" />
 </head>

 <body bgcolor="[% site.col.back %]"
       text="[% site.col.text %]" link="[% site.col.link %]"
       vlink="[% site.col.vlink %]" alink="[% site.col.alink %]">
  [% content %]
 </body>
</html>
```

It adds the standard headers and footers required to construct a valid HTML page, interpolating a number of variables along the way. These include the page title from page.title and several colors from site.col.

Example 11-22 shows the second and innermost template, *site/layout*. It defines an overall layout for the page content and other sitewide user interface components.

Example 11-22. templates/lib/site/layout

```
<table width="100%" border="0" cellpadding="0" cellspacing="5">

  <tr valign="middle">
    <td width="[% site.logo.width + 10 %]" align="center">
      [% PROCESS site/logo %]
    </td>
    <td>
      [% PROCESS site/header %]
    </td>
  </tr>

  <tr>
    <td></td>
    <td>[% PROCESS misc/line %]</td>
  </tr>

  <tr valign="top">
    <td align="center">
      [% PROCESS site/menu %]
    </td>
    <td>
      [% content %]
    </td>
```

```
  </tr>

  <tr>
    <td></td>
    <td>[% PROCESS misc/line %]</td>
  </tr>

  <tr valign="bottom">
    <td></td>
    <td align="center">
      [% PROCESS site/footer %]
    </td>
  </tr>

</table>
```

It does this by combining them in an HTML table to define the overall layout, but leaves the implementation specifics of each element to be handled by other template components. This approach allows you to get a clear overview of the layout without the distraction of too much messy detail. Each component does just one thing, making it easy to understand, modify, or replace.

11.2.2.2 Layout components

Example 11-23 shows the other user interface components that we're using in the overall layout for the site.

Example 11-23. templates/lib/site/logo

```
<a href="[% site.url.home %]">
[%- INCLUDE misc/image image=site.logo | trim -%]
</a>
```

The *site/logo* template shown in Example 11-23 uses *misc/image* to generate an appropriate image tag. This has leading and trailing whitespace removed with the trim filter and is enclosed in an element making it a link to the site home page. The *misc/image* template in Example 11-24 simply generates an HTML image tag.

Example 11-24. templates/lib/misc/image

```
<img src="[% image.src %]" alt="[% image.alt %]"
     width="[% image.width %]" height="[% image.height %]" border="0" />
```

The *misc/line* template in Example 11-25 is so simple that you might wonder why we're using it at all. It contains only an hr element to create a horizontal rule (i.e., line) across the page.

Example 11-25. templates/lib/misc/line

```
<hr />
```

This example is rather trivial but it illustrates the principle of creating a library of reusable presentation components. They define a particular look and feel for the site that can easily be changed at a later date. Although it is slightly more tedious in this case to write [% PROCESS misc/line %] than to embed the <hr/> HTML element directly in a template, it has the benefit of allowing us to make it more complicated later.[1]

[1] As indeed we will, later on in this chapter.

As indeed we will, later on in this chapter.

Using a template component from the start to generate this feature means that we will have to make changes in only one place. When we do make a change, all the templates that use the component will get the benefit of the update. You don't have to generate your entire user interface like this, only the parts that you think you might want to do differently at a later date.

When you're designing the look and feel for a site, you'll probably want to try out a few different combinations of user interface elements in various styles, colors, positions, and so on. If you create each as a separate template component, you can easily switch between them to find something that you like. This is also ideal for showing different possibilities to your customer, manager, or whoever has the ultimate responsibility for how the site should look. They may not care too much about how the bike shed was built, but you can be sure they will have some opinion on what color it should be painted.[2]

[2] See http://www.unixguide.net/freebsd/faq/16.19.shtml for the origins of this analogy.

The *site/header* template is also very simple. It displays the page title and any information about the page, defined in page.title and page.about, respectively. This is shown in Example 11-26.

Example 11-26. templates/lib/site/header

```
<h1 class="title">[% page.title %]</h1>
[% IF page.about -%]
<div class="info">
  [% page.about %]
</div>
[% END -%]
```

We will be looking at generating menus and other navigation components later in this chapter. For now we'll start with something simple such as the template in Example 11-27, which provides a basic menu linking to various pages in the site.

Example 11-27. templates/lib/site/menu

```
[%  menu = {
      index = 'Home'
      about = 'About'
      news  = 'News'
    };

    order = ['index' 'about' 'news'];

    FOREACH item IN order;
-%]
<a href="[% site.url.root %]/[% item %].html">[% menu.$item %]</a>
<br />
[%  END -%]
```

Last but not least we have the *site/footer* template in Example 11-28. This adds a standard copyright message and some general information about the page.

Example 11-28. templates/lib/site/footer

```
<p class="info">
  &copy; Copyright [% site.copyright %].
  All Rights Reserved.
 <br />
  [% page.file %] last modified [% page.date %]
</p>
```

11.3 Content Templates

We now have a library of template components in place that defines a common configuration and presentation for our web site content. This is applied automatically by *ttree* for each page template it processes so that we don't have to worry about it. Our page templates can concentrate on defining core page content without being obscured by elements of the user interface.

11.3.1 HTML Pages

In Example 11-7, we saw how a title for a page can be defined in a META directive. In addition to this, we can now also provide an about item, as shown in Example 11-29.

Example 11-29. *templates/src/index.htm*

```
[% META title = 'Template Toolkit Home'
        about = 'Home page for the Template Toolkit'
%]

<p>
  Welcome to the Template Toolkit web site.
</p>

<p>
  This page would have more content but the editor
  is currently out enjoying an extended lunch break.
</p>

<p>
  We expect him back before the end of the year.
</p>
```

The title and about items are extracted automatically and displayed by the *site/header* template, along with the logo, menu, and footer. The rest of the template provides the page content, clean and simple.

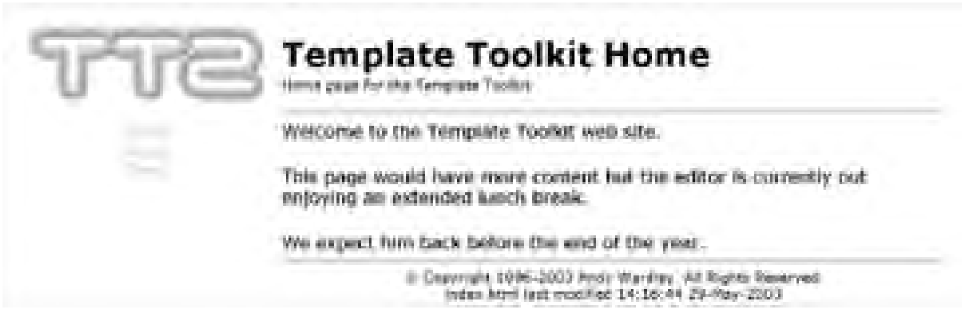
Now you can run the *bin/build* script to generate the HTML output page:

```
$ bin/build
```

The output file *html/index.html* is generated. Figure 11-1 shows what it looks like when viewed using a web browser.[3]

[3] In the screenshots in this chapter, we have deliberately increased the size of the user interface in proportion to the page content. On the real site, the logo, menu, and other navigation components are much smaller, leaving more room for the core page content, which is of course the most important thing.

Figure 11-1. The generated HTML index page



The benefits of separating the common user interface elements from the core page content should by now be obvious. Adding a new page to the web site is a simple matter of adding a page template to the *templates/src* directory. These templates contain only the core content of the page, and authors don't need to concern themselves with adding headers, footers, menus, or anything else that is common to the site as a whole. The only requirement is that they define the title and about values in a META tag, although both of these are strictly optional. If they don't define either the title or about, the relevant page.title or page.about values will be empty. If we want to be more strict, we could easily modify our *config/page* template to throw an error if one or another was undefined.

You will of course need to run the *bin/build* script whenever you add new pages. Assuming they process without error, the generated HTML output pages will then be accessible via the relevant URL for your web server. When you're happy with the new pages, you can then go and update your *site/menu* template to make them accessible via the menu. Remember that you'll need to rebuild the entire site when you make a change to a sitewide component such as *site/menu*, so invoke *bin/build* with the -a option.

11.3.2 CSS and Other Non-HTML Pages

With the wrapper and layout templates in place, we can enjoy the benefits of having the user interface elements added automatically. However, there may be pages for which we don't want this window dressing automatically added. We're going to look at a Cascading Style Sheet (CSS) as an example of such a page, but the principle applies equally well to JavaScript libraries, text files, XML files, and so on.

We could just define these files outside of the *templates/src* directory so that they bypass the regular build process. We would of course need to manually copy them into the *html* directory or configure the web server to locate them correctly. Or we could store them in the *templates/src* directory along with all the other page templates, but add *css, js, txt*, and any other relevant file extensions to the copy option in the *etc/ttree.cfg* configuration file, indicating the files that should be copied into place rather than processed.

However, these approaches bypass the Template Toolkit processing stage, which isn't what we want in this case. We have already defined various colors in the pre-processed configuration template *templates/lib/config/col,* and we would like to use these values in the CSS file. Assuming then that we are going to process the CSS file through the Template Toolkit, we can take advantage of this by adding any other directives that will simplify the job of maintaining the document—for example, by defining font information in one place at the start of the file and then using it by variable reference in numerous different places throughout the file.

Example 11-30 shows the start of the CSS file to illustrate the principle. For a detailed discussion of CSS, see *Cascading Style Sheets: The Definitive Guide* by Eric Meyer (O'Reilly). As far as the Template Toolkit is concerned, it is just another text format.

Example 11-30. templates/src/css/tt2.css

```
[% META type = 'text' %]

[%  font = {
      text = 'Verdana, Arial, Helvetica, sans-serif'
      mono = '"Courier New", Courier, monospace'
    }
-%]

body {
```

```
    font-family: [% font.text %];
    font-size: 12px;
}

.info {
    font-size: 10px;
}

.title {
    font-family: [% font.text %];
    font-size: 24px;
    line-height: 30px;
    font-weight: bold;
    color: [% site.col.text %];
    margin-top: 0px;
    margin-bottom: 2px;
}

a {
    font-family: [% font.text %];
    font-size: 12px;
    line-height: 14px;
    text-decoration: none;
    color: [% site.col.link %];
}

a:hover {
    color: [% site.col.alink %];
}

a.menu {
    white-space:nowrap;
}

a.menu:hover {
    color: [% site.col.alink %];
```

```
}

a.menuselect {
    font-weight: bold;
    color: [% site.col.mlink %];
    white-space:nowrap;
}

a.menuselect:hover {
    font-weight: bold;
    color: [% site.col.alink %];
}

...etc...
```

The META directive in the first line declaring a text template type is the key to bypassing the usual wrapper mechanism. You may recall it was one of the items that the *config/page* template examined, in this case copying it into the page.type variable. The default value, if not explicitly set in a META directive, is html.

All that needs to be done is a quick change to the *site/wrapper* template to handle different values for page.type. This is shown in Example 11-31.

Example 11-31. templates/lib/site/wrapper

```
[%  SWITCH page.type;
        CASE 'text';
            content;

        CASE 'html';
            content WRAPPER site/html
                          + site/layout;
        CASE;
            THROW page.type "Invalid page type: $page.type";
    END;
-%]
```

If the page type is text, the page content is passed through unaltered. If the page type is html, we apply the usual wrappers. Otherwise we throw an error reporting that we can't handle pages of whatever unknown type they claim to have.

You can achieve the same effect in other ways without using a META item. For example, the *config/page* template could examine the template path or file extension to determine the file type, or consult a lookup table or database mapping filenames to type.

11.3.3 Content Components

As you develop more content for your site you'll undoubtedly find yourself doing the same kinds of things over and over again. At this point it might be a good idea to see whether you can isolate what you're doing and create a template component or components that do it for you.

We're going to look at an example of laying out information in a table. The HTML table element is a complex beast with many options, but we're not going to try and emulate or replicate it. Instead, we're going to define one particular table style and a few different cell styles according to the look and feel of our site.

The first thing we need to do is to define some colors for our table. Example 11-32 shows the definition of a site.col.table data structure, added to the bottom of the *config/col* template.

Example 11-32. templates/lib/config/col

```
[%

  .

  .

  .

  site.col = {

    .

    .

    .

    line  = rgb.skyblue

    head  = rgb.darkblue

    table = {

      edge = rgb.skyblue

      back = rgb.white

      head = rgb.misty

      cell = rgb.ltgrey

    }

  }

-%]
```

The *table/edge* template shown in Example 11-33 generates a table element nested inside another. This provide us with a colored border (site.col.table.edge) around the table. The template is designed to be used with the WRAPPER directive, so it expects the contents of the table to be defined in the content variable.

Example 11-33. templates/lib/table/edge

```
<table border="0" cellspacing="1" cellpadding="0"

       bgcolor="[% site.col.table.edge %]">

  <tr>

    <td>

      <table border="0" bgcolor="[% site.col.table.back %]"

             cellspacing="2" cellpadding="4">

        [%- content -%]

      </table>

    </td>

  </tr>

</table>
```

Here's a simple way in which you would use the template defined in Example 11-33:

```
[% WRAPPER table/edge %]
<tr>
 <th>Forename</th>
 <td>Arthur</td>
</tr>
<tr>
 <th>Surname</th>
 <td>Dent</td>
</tr>
[% END %]
```

The *table/row* template, also designed for use with WRAPPER, generates an HTML tr element with the content embedded inside. This is shown in Example 11-34.

Example 11-34. templates/lib/table/row

```
<tr valign="top">
  [% content %]
</tr>
```

The *table/head* and *table/cell* templates both generate HTML td elements, but use different background colors from the site.col.table hash (see Examples Example 11-35 and Example 11-36).

Example 11-35. templates/lib/table/head

```
<td bgcolor="[% site.col.table.head %]">
  <b>[% content %]:</b>
</td>
```

Example 11-36. templates/lib/table/cell

```
<td bgcolor="[% site.col.table.cell %]">
  [% content %]
</td>
```

Now we can use these different components to do the hard work of generating HTML tables in a consistent style.

11.3.4 Debugging Pages

When you're creating components such as these you'll want somewhere to test them and get them working just right. It's a good idea to create a separate directory in your site for doing this, but don't throw the test pages away when you're done. If you create a page for debugging a component or set of components in isolation, you can use it to check that the components are working as expected right now, as well as in the future, when you decide to change the layout style and modify the template components.

So let's start by creating a page for debugging the table components that we've just created. Example 11-37 shows three different examples of tables created using these components.

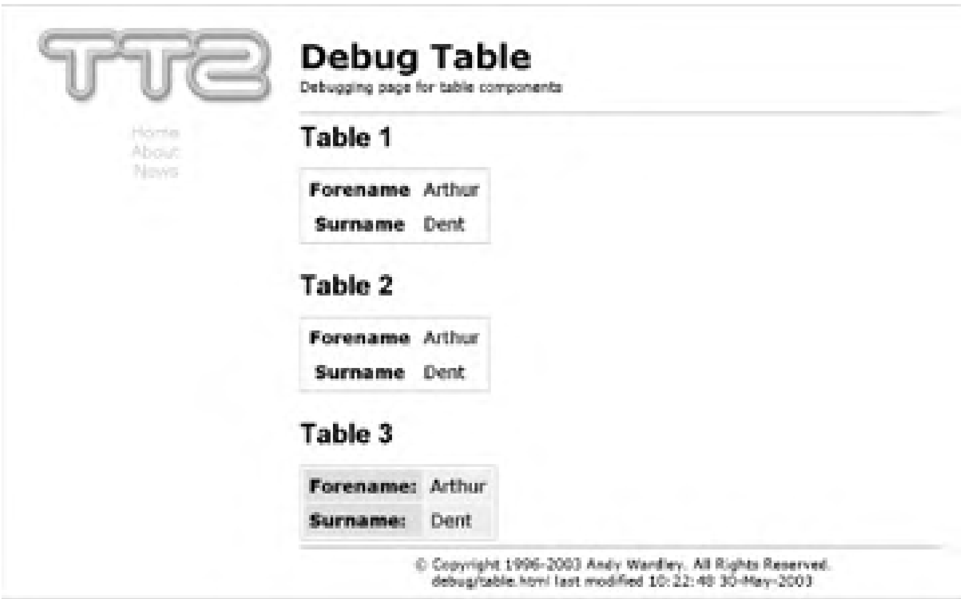
Example 11-37. templates/src/debug/table.html

```
<h2>Table 1</h2>

[% WRAPPER table/edge %]
  <tr>
    <th>Forename</th>
    <td>Arthur</td>
  </tr>
  <tr>
    <th>Surname</th>
    <td>Dent</td>
  </tr>
[% END %]

<h2>Table 2</h2>

[% WRAPPER table/edge %]
[%   WRAPPER table/row %]
       <th>Forename</th>
       <td>Arthur</td>
[%   END %]
[%   WRAPPER table/row %]
       <th>Surname</th>
       <td>Dent</td>
[%   END %]
[% END %]

<h2>Table 3</h2>

[% WRAPPER table/edge;
     WRAPPER table/row;
       INCLUDE table/head content='Forename';
       INCLUDE table/cell content='Arthur';
     END;
     WRAPPER table/row;
       INCLUDE table/head content='Surname';
       INCLUDE table/cell content='Dent';
```

```
    END;
  END
%]
```

Figure 11-2 shows the page generated by Example 11-37. Everything seems to be working as expected.

Figure 11-2. Debugging page for table components



Now let's add a page showing the contents of the site data structure. Or rather, let's write a *generic* template component that displays the contents of any hashlike data structure (see Example 11-38).

Example 11-38. templates/lib/debug/hash

```
[%  WRAPPER table/edge;
      FOREACH key = hash.keys;
          val = hash.$key;
        WRAPPER table/row;
          INCLUDE table/head content=key;
          WRAPPER table/cell;
            IF val.keys;
              INCLUDE debug/hash hash=val;
            ELSE;
              val;
            END;
          END;
        END;
      END;
  END;
-%]
```

Then all we need to do is to call the component passing the site data structure as the hash variable (see Example 11-

39).

Example 11-39. templates/src/debug/site.html

```
[% META title = 'Debug Site'
        about = 'Debugging page for the site data'
-%]


[% INCLUDE debug/hash hash=site %]
```

Figure 11-3 shows part of the page generated by the template in Example 11-39.

Figure 11-3. Debugging page for site data

It is a good idea to create a few debugging pages such as this that test any nontrivial template components you create. Whenever you make any changes to a component, you can check the relevant test page to ensure that it is still working as expected. Think of these pages as your test suite, designed to alert you quickly to any problems that may arise.

11.4 Navigation Components

Good navigation components are critical to making your web site accessible and allowing your visitors to find what they're looking for. A good general rule of user interface design is that a menu should have between three and seven items. Any more, and the user is faced with a daunting list of options to read through. Any fewer, and it's hardly a menu at all.

Given that a typical web site is likely to have more than seven pages, we need to consider how the pages and menus will be organized into some kind of structure. We'll look first at how a configuration file can be used to predefine this structure, automatically compute certain parts of it such as the URL for each page, and then make it accessible as part of the global site data. Then we'll create some template components that use this data structure to generate a menu and other navigation components.

We'll be keeping this example fairly simple so that we can concentrate on how the menus are constructed without getting bogged down in too much detail. Nevertheless, we will show how the site structure can be nested to any depth (within a reasonable limit), and also how it can be extended at runtime based on certain conditions, such as the value of the debug variable we set earlier in the *etc/ttree.cfg* file.

11.4.1 Adding Site Structure

The first rule of navigation is to have a good map.

Mapmaking is generally a laborious and time-consuming task, so we're going to get the Template Toolkit to do as much of the tedious work as possible. The map will be defined in the *config/map* template, so we need to modify the *config/main* template to PROCESS it, as shown in Example 11-40.

Example 11-40. Additions to templates/lib/config/main

```
[%  PROCESS config/page

         + config/site

         + config/url

         + config/col

         + config/images

         + config/map     # add this line

-%]
```

The *config/map* template is shown in Example 11-41.

Example 11-41. templates/lib/config/map

```
[%  # define map of pages in site

    map = {

        name = 'template-toolkit.org'

        menu = [ 'index', 'about', 'news', 'docs' ]

        page = {

            index    = { name = 'Home'     }

            about    = { name = 'About'    }

            news     = { name = 'News'     }

            docs     = {

                name = 'Documentation'

                menu = [ 'index', 'faq', 'manual' ]
```

```
            page = {
                index  = { name = 'Introduction' }
                faq    = { name = 'FAQ' }
                manual = {
                    name = 'Manual'
                    menu = [ 'index', 'syntax', 'directives' ]
                    page = {
                        index      = { name = 'Introduction' }
                        syntax     = { name = 'Syntax'      }
                        directives = { name = 'Directives'  }
                    }
                }
            }
        }
    }
};

IF debug;
    # add debugging pages
    map.page.debug = {
        name = 'Debug'
        menu = [ 'site' 'table']
        page = {
            site  = { name = 'Site'  }
            table = { name = 'Table' }
        }
    };

    # add debug item to main menu
    map.menu.push('debug');
END;

# save map in site
site.map = map;

# expand map recursively...
PROCESS config/expand;

-%]
```

11.4.1.1 Map nodes

The first section defines a nested map data structure:

```
map = {
  .
  .
  .
}
```

Each node in the map is represented by a hash array. This corresponds to a section or page in the site that has a unique location and a page associated with it. For example, the syntax page toward the bottom of the map correponds to the path *docs/manual/syntax.html* relative to the *templates/src* directory, and hence also to the */ttbook* URL or equivalent.

The one item that each node must contain is a name. This provides a short, readable name suitable for use in a menu.

```
syntax = { name = 'Syntax' }
```

If a node is a container for other pages, such as the manual node that contains the syntax page, the pages should be defined in a page hash:

```
manual = {
  name = 'Manual'
  menu = [ 'index', 'syntax', 'directives' ]
  page = {
    index      = { name = 'Introduction' }
    syntax     = { name = 'Syntax'       }
    directives = { name = 'Directives'   }
  }
}
```

The final addition is the menu item, also shown in this example. This defines the order in which the pages should be displayed in a menu. Remember that hash arrays don't retain the order of the items they contain, so we need to add this to make it explicit.

What we end up with is a complete page node that can be added to the page hash of a parent container:

```
docs = {
  name = 'Documentation'
  menu = [ 'index', 'faq', 'manual' ]
  page = {
    index  = { name = 'Introduction' }
    faq    = { name = 'FAQ' }
    manual = {
      .
      .  [ the manual node ]
      .
    }
  }
}
```

That node can then be added to another, which is added to another, and so on, until the complex site structure, or the part that is currently relevant to you, is defined.

11.4.1.2 XML site map

For a large site, the map could quickly become complex and difficult to maintain. However, you don't have to define it all at once, or all in the same place. You can just as easily store the information in an external XML file or SQL database and use one of the XML or DBI plugins to load it into place.

Example 11-42 shows how the same data information could be defined in an XML file.

Example 11-42. xml/sitemap.xml

```
<map>
  <name>template-toolkit.org</name>
  <menu>index</menu>
  <menu>about</menu>
  <menu>news</menu>
  <menu>docs</menu>
  <page id="index" name="Home" />
  <page id="about" name="About" />
  <page id="news"  name="News" />
  <page id="docs"  name="Documentation">
    <menu>index</menu>
    <menu>faq</menu>
    <menu>manual</menu>
    <page id="index"  name="Introduction" />
    <page id="faq"    name="FAQ" />
    <page id="manual" name="Manual">
      <menu>index</menu>
      <menu>syntax</menu>
      <menu>directives</menu>
      <page id="index"      name="Introduction" />
      <page id="syntax"     name="Syntax" />
      <page id="directives" name="Directives" />
    </page>
  </page>
</map>
```

Example 11-43 shows a variation of the *lib/map* template from Example 11-42. It uses the XML::Simple plugin to load the XML file and define the map variable. The KeyAttr parameter tells it to use the id attribute to index items.

Example 11-43. templates/lib/config/mapx

```
[%  USE map = XML.Simple(
        "$rootdir/xml/sitemap.xml"
```

```
        KeyAttr = ['id']
    );

    IF debug;
        # as before
        .
        .
        .

-%]
```

11.4.1.3 Selective mapmaking

Another approach to making a complex sitemap easier to maintain is to add bits in stages—for example, by defining the structure of each major section of the site in separate files. These can then be loaded via PROCESS and merged into a single map, much in the same way that we use several different configuration templates to build up the site data structure.

The next section of the *site/map* template shows one way this can be done. Here we define a submenu for our debugging pages, but only if the debug variable is set to true.

```
IF debug;
    # add debugging pages
    map.page.debug = {
        name = 'Debug'
        menu = [ 'site' 'table']
        page = {
            site  = { name = 'Site'  }
            table = { name = 'Table' }
        }
    };

    # add debug item to main menu
    map.menu.push('debug');
END;
```

If you want to enable the debugging pages, run bin/configure with the -d command-line option, or answer yes when prompted. Then run bin/build -a to rebuild the site with the debugging menu enabled.

The final part of the file saves the map structure in site.map and then calls *config/expand* to walk the map structure and expand it with additional items:

```
# save map in site
site.map = map;

# expand map recursively...
PROCESS config/expand;
```

11.4.2 Walking the Structure

The *config/expand* template is where all the deep magic behind our navigation system takes place. We're cramming a lot into a small space, and the template is rather complex as a result. In fact, this is probably the most complicated template that we're using to build the site.

Templates such as this often start simple and grow more complex as you develop the site further. For a real web site, we would probably implement this complex functionality as a Perl subroutine or plugin module. More likely, we would prototype it as a template and later implement it in Perl when we have a better idea about exactly what we want.

Nevertheless, we'll continue to use this as an example of the kind of complicated task that can be undertaken using the Template Toolkit, should you choose to do so.

Example 11-44 shows what the *config/expand* template looks like.

Example 11-44. templates/lib/config/expand

```
[%  # page.trail tracks path to the current page
    DEFAULT page.trail = [ ];

    # list of menu items we're constructing
    map.items = [ ];

    # walk through item names in map.menu
    FOREACH id IN map.menu;
        # fetch page from map.page
        THROW map "Invalid menu item in $map.name: $id"
            UNLESS (item = map.page.$id);

        # add location data
        item.id   = id;
        item.path = path ? "$path/$id" : id;
        item.file = item.page
                  ? "${item.path}/index.html"
                  : "${item.path}.html";
        item.url  = "$site.url.root/$item.file";

        # is this item on the path to the current page?
        item.hot  = page.file.match("^$item.path");
        item.subs = item.hot and item.menu;
        item.here = (item.file =  = page.file);

        # set next/last if this is the actual page
        IF item.here;
            page.prev = map.page.${loop.last};
            page.next = map.page.${loop.next};
```

```
        END;

        # add item to map items list
        map.items.push(item);

        # also to the trail if the page is hot
        page.trail.push(item) IF item.hot;

        # expand any submenu for this item
        IF item.subs;
           INCLUDE config/expand
              map  = item
              path = item.path;
        END;
    END;

-%]
```

It expects to be passed a map variable referencing a page node in the format defined in config/map. It walks through each page element defined within it in the order specified in the menu item. It calls itself recursively to process all the pages within pages within pages, to ensure that each node in the map is visited.

The purpose of visiting each node is to define additional data items that we are too lazy to add by hand. It's not just that we can't be bothered to go to the effort of adding relative paths, full URLs, and so on to each page. The real reason is that there is so much repetition of the same values that it's going to be tedious, time-consuming, and error-prone work that can be much better handled by a machine. Furthermore, some of these items are based on values that we will want to change from time to time (such as the base URL), so it makes sense to compute them at runtime.

Another reason for visiting each node is to construct an items list within the map that contains references to the pages in page in the order defined by menu. This will allow us to iterate directly through the page items in a map node in the correct order, without having to explicitly reference the page using an identifier each time. In other words, we're making life easier for ourselves later on.

The final reason is to determine which nodes are on the path to the current page and which pages, if any, come before or after the page in the menu. We'll be using this later to create a "bread-crumb trail" and links to the previous and next pages.

The list of page nodes on the path to the current file will be stored in page.trail, so the first thing *config/expand* does is to make sure it exists:

```
DEFAULT page.trail = [ ];
```

Then it creates a new items list in the current map node:

```
map.items = [ ];
```

Then it iterates through each page identifier, id, in the menu, map.menu:

```
FOREACH id IN map.menu;
    # fetch page
    THROW map "Invalid menu item in $map.name: $id"
        UNLESS (item = map.page.$id);
```

```
    .
    .
    .
END
```

It uses the identifier to index into the page map, map.page.$id, to fetch a page hash. This is then stored in the item variable, or an error is thrown if an invalid identifier is used. The id, path, file, and url items are then computed and added to item.

```
# add location data
item.id   = id;
item.path = path ? "$path/$id" : id;
item.file = item.page
         ? "${item.path}/index.html"
         : "${item.path}.html";
item.url  = "$site.url.root/$item.file";
```

Notice how the path variable is being used to construct the item.path, which is then used in item.file and item.url. We'll see how this works when we look at how the config/expand template calls itself recursively. But first, we should look at the other values that are added to each item.

```
# is this item on the path to the current page?
item.hot  = page.file.match("^$item.path");
item.subs = item.hot and item.menu;
item.here = (item.file =  = page.file);
```

The item.hot flag is set if the path to the item matches the beginning (or all) of the path for the current page being processed. In other words, it indicates that the node is on the path to the current page. For example, if the page.file variable contains the value docs/manual/index.html, the nodes marked as hot in the map would be docs, manual, and index, whose paths are docs, docs/manual, and docs/manual/index, respectively.

The item.subs flag goes a little further, indicating that the node is hot and also has further items contained within it. The last flag, item.here, indicates that the item is the actual node for the current page being processed.

If the item.here flag is set, we've found the node for the page we're processing, in which case we can set page.prev and page.next to point to the data structures for the previous and next pages:

```
# set next/last if this is the actual page
IF item.here;
   page.prev = map.page.${loop.last};
   page.next = map.page.${loop.next};
END;
```

The loop.last and loop.next variables provide us with the identifiers for the previous and next pages in the FOREACH loop. We use these to key into the map.page structure to fetch references to the hash arrays for the pages, if they exist.

Now that we've got a complete item we can add it to the map.items list:

```
# add item to list
map.items.push(item);
```

If the item is hot, we also add it to page.trail:

```
# also to the trail if the page is hot
page.trail.push(item) IF item.hot;
```

Then if the item.subs flag is set, the *config/expand* template recursively processes itself to expand the children and further descendants of the item:

```
# expand any submenu for this item
IF item.subs;
   INCLUDE config/expand
      map  = item
      path = item.path;
END;
```

The current item variable is passed as map and a new value for path is provided so that all the paths generated within it will be relative to the path for the current item.

As we already mentioned, this is perhaps the most complicated template in the site, so don't be surprised if you find it daunting. It can take a little time and patience to get something as complicated as this working properly, but it is usually something you have to do only once and can then forget.

It is also worth reiterating that when things start getting complicated, you can always recode in Perl and load the functionality in using a plugin, for example. That would certainly be the approach we would adopt if this template needed to become any more complex than it already is.

11.4.3 Building a Nested Menu

Now that we have a complete map defined, we can write a template that builds a menu from this data structure. Example 11-45 shows one way this can be done.

Example 11-45. templates/lib/menu/nest

```
[%  DEFAULT pad = '';

   FOREACH item = menu.items;
      pad;

      INCLUDE menu/text
        link = {
          text     = item.name
          url      = item.url
          class    = item.hot ? 'menuselect' : 'menu'
        };

      IF item.subs;
         "<br />\n";
         INCLUDE menu/nest
           menu = item
           pad  = pad ? "   $pad"
                      : " - ";
      END;

      "<br />\n";
```

```
  END
-%]
```

The *menu/nest* template also calls itself recursively to generate nested menus representing the structure of the site. For each invocation, the menu variable references the current site map node being processed. The pad variable contains a string used to indent each item by an amount appropriate to the current nesting depth.

The template iterates through each item in the menu.items list that now contains references to complete page structures, thanks to the work of the *config/expand* template:

```
FOREACH item = menu.items;
  .
  .
  .
END
```

Inside the loop, it prints the current pad string and then calls *menu/text* to generate a text link for the menu item:

```
pad;

INCLUDE menu/text
  link = {
    text    = item.name
    url     = item.url
    class   = item.hot ? 'menuselect' : 'menu'
  };
```

The *menu/text* template is passed a link hash that contains values extracted from the current menu item. The class value is set to correspond to one of the styles defined in the *templates/src/css/tt2.css* file, according to whether the item is hot and on the path to the current page. Example 11-46 shows the *menu/text* template.

Example 11-46. templates/lib/menu/text

```
<a href="[% link.url %]"
[%- " class=\"$link.class\""
      IF link.class
-%]
>[%- link.text -%]</a>
```

The final task of the *menu/nest* template is to process any nested items if the item.subs flag is set:

```
IF item.subs;
  "<br />\n";
  INCLUDE menu/nest
    menu = item
    pad  = pad ? "   $pad"
               : " - ";
END;
```

When the *menu/nest* template is called recursively, the item is passed as the new menu target and the pad is set to provide a deeper level of indenting.

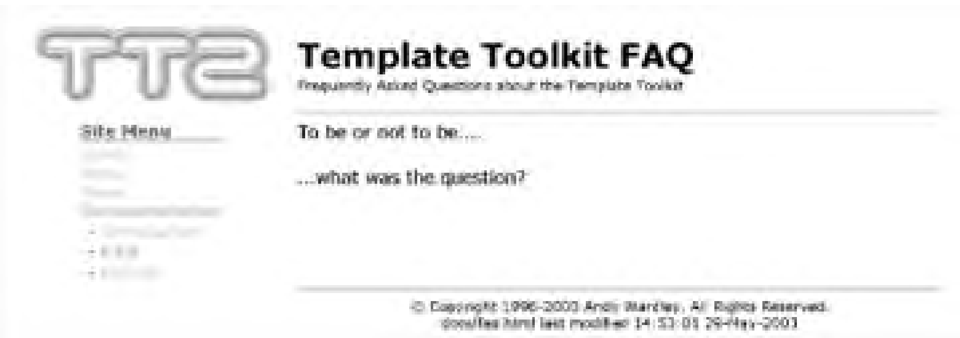
Now all we need to do is to modify the *site/menu* template to use the new *menu/nest* component, passing the top-level site map node, site.map, as the starting value for menu. While we're at it, we'll also add a title bar for the menu. Example 11-47 shows the changes made to *site/menu*.

Example 11-47. Changes made to templates/lib/site/menu

```
<table border="0" cellpadding="0" cellspacing="0">
  <tr>
    <td align="left" class="menutitle">
      Site Menu
    </td>
  </tr>
  <tr>
    <td>[% PROCESS misc/line %]</td>
  </tr>
  <tr valign="top">
    <td align="left">
      [% INCLUDE menu/nest
           menu = site.map
      -%]
    </td>
  </tr>
</table>
```

Figure 11-4 shows a screenshot containing the new menu.

Figure 11-4. Page with nested menu



Notice how the hot items in the menu are shown in bold orange[4] text as defined by the menuselect CSS style. Other menu items are displayed in the normal menu style.

[4] Not that you can tell in a grayscale image, but trust us, they're orange.

11.4.4 A Stacked Menu

The nested menu style works well when we need to nest menus that are only two or perhaps three levels deep. Any more than that and the menu will start to occupy more horizontal space that will cut into the page content.

We can easily create a new menu component that stacks menus on top of each other instead of nesting them. This is shown in Example 11-48.

Example 11-48. templates/lib/menu/stack

```
[%  pending = [ menu ];

    WHILE pending.size;
        menu = pending.shift;

        "<p>\n";
        FOREACH item = menu.items;
            PROCESS menu/text
              link = {
                text     = item.name
                url      = item.url
                class    = item.hot ? 'menuselect' : 'menu'
              };

            "<br />\n";

            pending.push(item)
              IF item.subs;
        END;
        "</p>\n";
    END;
-%]
```

The pending variable is used to keep a list of the menus that require processing, starting with the menu passed in as an argument, as per menu/nest:

```
pending = [ menu ];
```

The WHILE block repeats while there are menus in the pending list, removing the first menu in the list each time around:

```
WHILE pending.size;
    menu = pending.shift;

    .
    .
    .

END;
```

Other than adding a few HTML elements, the main part of the body of the WHILE block simply iterates over the items in the current menu, calling *menu/text* to process each:

```
FOREACH item = menu.items;
    PROCESS menu/text
      link = {
```

```
        text     = item.name
        url      = item.url
        class    = item.hot ? 'menuselect' : 'menu'
      };

    "<br />\n";

    pending.push(item)
      IF item.subs;
END;
```

When an item is found that has the subs flag set, it is added to the list of pending items. It will be processed after the current menu is complete, and will appear underneath it.

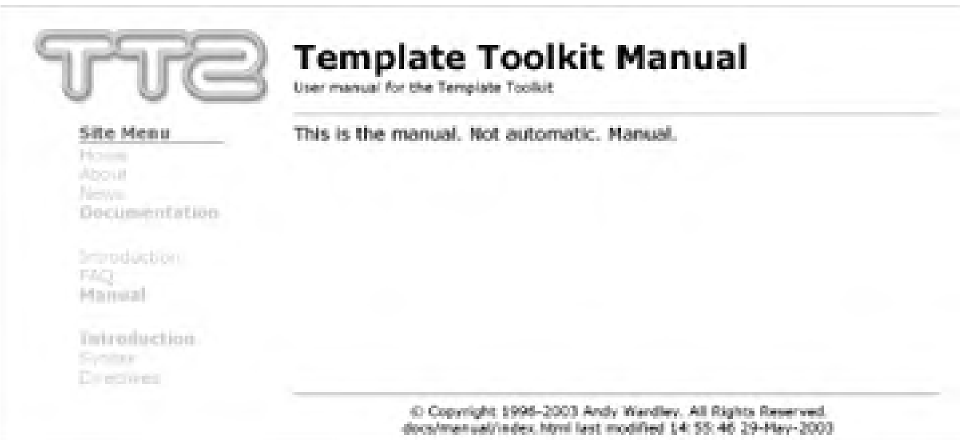
A quick change in *site/menu* from menu/nest to menu/stack is all that is required to use the new menu, as shown in Example 11-49.

Example 11-49. Changes to templates/lib/site/menu

```
 .
 .
 .
 <tr valign="top">
   <td align="left">
     [% INCLUDE menu/stack
          menu = site.map
     -%]
   </td>
 </tr>
</table>
```

Figure 11-5 shows a page with stacked menus.

Figure 11-5. Page with stacked menus



11.4.5 Bread-Crumb Trail

The name *bread-crumb trail* is borrowed by web developers from the story of Hansel and Gretel. They left a trail of bread-crumbs through the woods to help them find their way back from the wicked witch's edible house.[5] In the context of a web site, it refers to a commonly used navigation component that shows the steps a visitor has taken from the site home page down to the current page location.

[5] Alas, the hungry birds ate the bread-crumbs, but things turned out alright for them in the end.

The config/expand template has already stored the list of hot page nodes in the page.trail list. All we need is a template to display the information. This is shown in Example 11-50.

Example 11-50. templates/lib/menu/trail

```
<table border="0" cellpadding="0" cellspacing="2">
  <tr valign="middle">
[% FOREACH item IN trail %]
    <td class="info"></td>
    <td>[% PROCESS menu/text
             link = {
                 text     = item.name
                 url      = item.url
                 class    = 'menu'
             };
       %]</td>
[% END %]
  </tr>
</table>
```

Then we can update the site/layout to include it in an appropriate place, as shown in Example 11-51.

Example 11-51. Adding the bread-crumb trail to templates/lib/site/layout

```
.
    .
    .
    <td width="100%">
      [% PROCESS site/header %]
    </td>
  </tr>

  <!-- new section added -->
  <tr>
    <td align="center">
      [% PROCESS site/name %]
```

```
    </td>

    <td>

      [% PROCESS site/navigate %]

    </td>

  </tr>

  <!-- end of new section -->


  <tr>

    <td></td>

    <td>[% PROCESS misc/line %]</td>

    .

    .

    .
```

Two new templates are being added, *site/name* and *site/navigate*. The first adds a nameplate underneath the logo (see Example 11-52).

Example 11-52. templates/lib/site/name

```
<a href="[% site.url.home %]">

[%- INCLUDE misc/image image=site.image.name | trim -%]

</a>
```

This is purely for aesthetic reasons to help keep the layout balanced when we add the bread-crumb trail. The *site/navigate* component does nothing more than display the bread-crumb trail (see Example 11-53). However, we will be adding more to this template shortly.

Example 11-53. templates/lib/site/navigate

```
[% PROCESS menu/trail trail=page.trail %]
```

Now you can run `bin/build -a` to rebuild the entire site and see the pages with the bread-crumb trail added. Figure 11-6 shows a screenshot of a page containing the new bread-crumb trail.

Figure 11-6. Bread-crumb trail

11.4.6 Previous and Next Pages

We can also add a navigation component to add links to the previous and next pages relative to the current one. These were also determined by the *config/expand* template and set in the page.prev and page.next variables. Either of these values could be undefined, so we need to be sure to cover those cases. Example 11-54 shows the *menu/prevnext* template component that generates these links.

Example 11-54. templates/lib/menu/prevnext

```
<table border="0" cellpadding="2" cellspacing="2">
  <tr valign="middle">
  [% IF page.prev -%]
    <td align="right">
      [% PROCESS menu/text
           link = {
             text  = page.prev.name
             url   = page.prev.url
             class = 'menu'
           };
     -%]
    </td>

    [% IF page.next -%]
    <td>|</td>
    [% END -%]
  [% END %]

  [% IF page.next %]
    <td align="left">
      [%- PROCESS menu/text
            link = {
              text  = page.next.name
              url   = page.next.url
              class = 'menu'
            };
      %]
    </td>
  [% END %]
  </tr>
</table>
```

Once again, *menu/text* is being used to generate the individual text links. This template is mostly just providing the layout logic.

The *site/navigate* template can now be modified to incorporate the new navigation component, as shown in Example

11-55.

Example 11-55. Adding menu/prevnext to templates/lib/site/navigate

```
<table width="100%" border="0" cellpadding="0" cellspacing="0">
  <tr valign="middle">
    <td align="left">
      [% PROCESS menu/trail trail=page.trail %]
    </td>
    <td align="right">
      [% PROCESS menu/prevnext %]
    </td>
  </tr>
</table>
```

Figure 11-7 shows a page with the bread-crumb trail on the left, with links to the previous and next pages on the right of the page header.

Figure 11-7. Previous and next pages

11.5 Structuring Page Content

We've looked at different ways that template components can be used to generate shared user interface components such as headers, menus, and footers. Now we are going to turn our attention to the page content itself, showing how the Template Toolkit can be used to help structure and present content in different ways.

11.5.1 Defining Sections

Web pages containing any more than a few paragraphs will typically be organized into sections, subsections, or some other kind of logical division. A simple HTML page may use nothing more than <h1> and <h2> elements to break up a document into small chunks. A more complex page might add all manner of fancy HTML markup to indicate section breaks or other structural parts of a document. You might also want to include a table of contents at the top of the page, linking to sections of the document below.

Needless to say, all this involves extra work that requires a lot of repetition. We want to make it easy to add and update site content, and don't want to burden page authors with the task of adding presentation markup, generating and maintaining tables of contents, and so on. Furthermore, we want to keep the presentation aspects separate so that we can restyle the site at a later date without having to rewrite all the content.

The solution is of course to use templates to define the presentation elements, which are then automatically applied to the page content. We will also show how a table of contents can be automatically generated from the structure of the content.

11.5.1.1 Section headers

Adding a standard block of HTML markup at the start of each section in a page is as easy as calling a template component. Example 11-56 shows a page that uses the INCLUDE directive to add a section header in two places.

Example 11-56. Adding section headers

```
[% META title = 'About the Template Toolkit'
        about = 'A brief overview of and introduction
                 to the Template Toolkit'
%]

[% INCLUDE section/header
     title = 'Overview'
%]
<p>
  The <b>Template Toolkit</b> is a fast,
  powerful, and easily extensible template
  processing system written in <b>Perl</b>...
</p>

[% INCLUDE section/header
     title = 'Mailing Lists'
%]
<p>
  A number of mailing lists are provided for discussing
```

```
  the Template Toolkit...

</p>
```

A simple template for generating each section header is shown in Example 11-57. Here we are using the *misc/line* template component to add a line across the page, followed by the section title in a <h1> element.

Example 11-57. templates/lib/section/header

```
[% PROCESS misc/line %]


<h1>[% title %]</h1>
```

You might also want to define macros to make using these components as easy as possible. These can be defined at the top of the page or, better still, in a preprocessed configuration template. For example:

```
[% MACRO Section(title) INCLUDE section/header %]
```

With this macro defined, the page content can be simplified, as shown in Example 11-58.

Example 11-58. Using a section macro

```
[% META title = 'About the Template Toolkit'

        about = 'A brief overview of and introduction

                 to the Template Toolkit'

%]


[% Section('Overview') %]

<p>

  The <b>Template Toolkit</b> is a fast,

  powerful, and easily extensible template

  processing system written in <b>Perl</b>...

</p>


[% Section('Mailing Lists') %]

<p>

  A number of mailing lists are provided for discussing

  the Template Toolkit...

</p>
```

11.5.1.2 Section wrappers

If you want to add some markup at the start of a section and some more at the end, you could use separate *section/header* and *section/footer* templates. But as we know from looking at page headers and footers, a better approach is to create a single wrapper template.

Let's say that we want to add the title at the start of the section, but move the line generated by *misc/line* to come *after* the content for the section. Example 11-59 shows a wrapper template to do this.

Example 11-59. templates/lib/section/wrapper

```
<h1>[% title %]</h1>

[% content %]

[% PROCESS misc/line %]
```

To use this component, the page template should use the WRAPPER directive, enclosing the content for each section between WRAPPER and END. This can be seen in Example 11-60.

Example 11-60. Using a section wrapper

```
[% META title = 'About the Template Toolkit'

        about = 'A brief overview of and introduction

                 to the Template Toolkit'

%]


[% WRAPPER section/wrapper

     title = 'Overview'

%]

<p>

  The <b>Template Toolkit</b> is a fast,

  powerful, and easily extensible template

  processing system written in <b>Perl</b>...

</p>

[% END %]


[% WRAPPER section/wrapper

     title = 'Mailing Lists'

%]

<p>

  A number of mailing lists are provided for discussing

  the Template Toolkit...

</p>

[% END %]
```

11.5.1.3 Sections and subsections

You can create as many different components as you require for sections, subsections, subsubsections, and any other page elements. Example 11-61 shows a page with a more complex structure, including subsections nested within sections.

Example 11-61. Sections and subsections

```
[% META title = 'About the Template Toolkit'

        about = 'A brief overview of and introduction

                 to the Template Toolkit'
```

```
%]
[% MACRO Section(title)    INCLUDE page/section;
   MACRO Subsection(title) INCLUDE page/subsection
%]

[% Section('Overview') %]
<p>
  The <b>Template Toolkit</b> is a fast,
  powerful, and easily extensible template
  processing system written in <b>Perl</b>...
</p>

[% Subsection('Features' ) %]
<ul>
  <li>Fast, powerful, and extensible...</li>
  <li>Powerful presentation language...</li>
  <li>And so on...</li>
</ul>

[% Section('Mailing Lists') %]
<p>
  A number of mailing lists are provided for discussing
  the Template Toolkit.
</p>

[% Subsection('templates') %]
<p>
  The <b>templates</b> mailing list exists
  for reporting information, asking questions, and
  discussing development or any other topic
  relevant to the Template Toolkit.
</p>

[% Subsection('templates-announce') %]
<p>
  The <b>templates-announce</b> mailing list
  is a low-volume list used for announcing
```

```
  new versions of the Template Toolkit
  or other related information.
</p>
```

Example 11-62 shows the *page/section* template and Example 11-63 shows the *page/subsection* template.

Example 11-62. templates/lib/page/section

```
<table width="100%" border="0" cellpadding="0" cellspacing="4">
  <tr>
    <td align="left">
      <a name="[% id %]">
        <h2 class="section">[% title %]</h2>
      </a>
    </td>
    <td align="right">
      [% UNLESS no_top -%]
      <a href="#top" class="navlink">Top</a>
      [% END -%]
    </td>
  </tr>
</table>
```

Example 11-63. templates/lib/page/subsection

```
<a name="[% id %]">
<h3 class="subsection">[% title %]</h3>
</a>
```

The template in *page/section* is a little more involved than the simpler *page/subsection* template. Both templates generate an HTML anchor around the title using an optional id variable as the identifier. We'll be looking at this in the next section when we build a table of contents to link down to the different sections and subsections in a document.

11.5.2 A Table of Contents

We now have the page content defined in terms of sections and subsections. From this, we can generate a table of contents to help the reader navigate around the document structure.

11.5.2.1 Anchor points

We saw in the previous section how the *page/section* and *page/subsection* templates in Examples Example 11-62 and Example 11-63, respectively, generate an HTML <a> element to create an anchor point in the document. To use this feature, a value must be provided for the id variable:

```
[% INCLUDE page/subsection
     title = 'Testing 123'
        id = 'testing'
%]
```

This generates the following HTML:

```
<a name="testing">
<h3 class="subsection">Testing 123</h3>
</a>
```

This subsection can now be linked to by appending #testing to the end of the page URL—e.g., *http://localhost/ttbook/about.html#testing*.

11.5.2.2 Better page macros

The first task is to enhance the Section and Subsection macros. We'll define these in a separate *config/macros* template, shown in Example 11-64.

Example 11-64. templates/lib/config/macros

```
[%  page.items = [ ];


    MACRO Section(title) BLOCK;
        id   = title.replace('\W+', '_');
        item = {
            url   = "#$id"
            name  = title
            items = [ ]
        };
        CALL page.items.push(item);
        PROCESS page/section;
    END;


    MACRO Subsection(title) BLOCK;
        id   = title.replace('\W+', '_');
        item = {
            url  = "#$id"
            name = title
        };
        CALL page.items.last.items.push(item);
        PROCESS page/subsection;
    END;
-%]
```

The first line creates a reference to an empty list and assigns it to page.items. This will be used to keep track of each section in the page.

```
page.items = [ ];
```

The Section expects a title argument, as before. The body of the macro is defined as a BLOCK continuing down to the corresponding END directive.

```
MACRO Section(title) BLOCK;
   # macro body
END;
```

The title is used to generate an HTML-compliant identifier for the section by replacing all sequences of one or more nonword characters with a single underscore:

```
id = title.replace('\W+', '_');
```

The item variable is then defined as a hash array containing values for url and name. It also defines an items list for storing information about any subsections contained within this section.

```
item = {
   url   = "#$id"
   name  = title
   items = [ ]
};
```

The new item is added to the page.items list:

```
CALL page.items.push(item);
```

Finally, the *page/section* template is processed to generate the appropriate HTML markup for the section heading:

```
PROCESS page/section;
```

The Subsection macro differs in a few minor details. To keep things simple for this example, we are not providing any support for nesting subsubsections within subsections, although it would be easy to add. As a result, there is no need for an items list in the item hash.

```
item = {
   url  = "#$id"
   name = title
};
```

Instead of being pushed onto the page.items list, the new item is added to the items list for the current section—that is, the last item on the page.items list:

```
CALL page.items.last.items.push(item);
```

Of course it uses the *page/subsection* template rather than the *page/section* template to generate the subsection header.

To make these MACRO definitions visible, we need to update the *config/main* template to add *config/macros* to the list of templates in the PROCESS directive. Example 11-65 shows the relevant change.

Example 11-65. Addition to config/main

```
[%  PROCESS config/page
         + config/site
         + config/url
         + config/col
         + config/images
         + config/map
         + config/macros   # add this line
-%]
```

11.5.2.3 Generating the table of contents

These macros build up information about the structure of the page content and store it in the page.sections list. Generating a table of contents is then a simple matter of iterating through this data and presenting it nicely as a set of formatted links.

Given that this data isn't complete until the page is processed in its entirety, you may be wondering how we can generate a table of contents to be inserted at the top of the page. The answer is that we use a WRAPPER around the page, as shown in Example 11-66. For the sake of clarity, we removed the page content to show only the directives in question.

Example 11-66. Page layout wrapper

```
[% META title = 'About the Template Toolkit'

        about = 'A brief overview of and introduction

                 to the Template Toolkit'

%]


[% WRAPPER page/tocpage %]


[% Section('Overview') %]

   ...


[% Subsection('Features' ) %]

   ...


[% Section('Mailing Lists') %]

   ...


[% Subsection('templates') %]

   ...


[% Subsection('templates-announce') %]

   ...


[% END %]
```

The page content is enclosed in a WRAPPER ... END block. The content is processed first, thereby triggering the Section and Subsection macros, and is then passed off to the *page/tocpage* template for presentation (see Example 11-67).

Example 11-67. templates/lib/page/tocpage

```
<h2>Contents</h2>


<ul>

[% FOREACH section IN page.items -%]

<li><a href="[% section.url %]">[% section.name %]</a></li>

[% PROCESS subs IF section.items.size -%]
```

```
[% END -%]

</ul>


[% BLOCK subs -%]

<ul>

[% FOREACH sub IN section.items -%]

<li><a href="[% sub.url %]">[% sub.name %]</a></li>

[% END -%]

</ul>

[% END %]


[% content %]
```

The first section generates the main table of contents using a FOREACH loop to iterate through each section in the page.items list:

```
<ul>

[% FOREACH section IN page.items -%]

<li><a href="[% section.url %]">[% section.name %]</a></li>

[% PROCESS subs IF section.items.size -%]

[% END -%]

</ul>
```

If a section contains subsections, the subs block is called to create a nested menu. This works in an identical way to the main body, but iterates over the items in section.items rather than page.items.

```
[% BLOCK subs -%]

<ul>

[% FOREACH sub IN section.items -%]

<li><a href="[% sub.url %]">[% sub.name %]</a></li>

[% END -%]

</ul>

[% END %]
```

The page content then follows after the table of contents:

```
[% content %]
```

11.5.2.4 Reusing menu components

You may have noticed that page.items data defined by the Section and Subsection macros has the same basic structure as for our site menu. Each item has a name, a url, and a list of nested items. This choice was deliberate. It allows us to reuse our menu template components to generate the table of contents.

Example 11-68 shows a different version of the *page/tocpage* template from what we saw in Example 11-67.

Example 11-68. Table of contents generated using menu/nest

```
[% FOREACH section IN page.items;

     SET section.subs = 1
```

```
        IF section.items.size;
   END
-%]

<h2>Contents</h2>

[% INCLUDE menu/nest menu=page %]

[% content %]
```

There is one modification we need to make to the data. The *menu/nest* template is programmed to descend into nested items if the subs value is set. The first block of the template in Example 11-68 uses a FOREACH directive to iterate through each item, setting the subs value to 1 if it contains any items:

```
[% FOREACH section IN page.items;
       SET section.subs = 1
        IF section.items.size;
   END
-%]
```

This ensures that the *menu/nest* template will display the entire table of contents, including nested subsections. The *menu/nest* template is called, passing page as the local value for the menu variable. It will then walk through the entries in the page.items list, and also through any nested items within them.

```
<h2>Contents</h2>

[% INCLUDE menu/nest menu=page %]
```

As before, we display the page content after the table of contents. Figure 11-8 shows an HTML page built this way.

Figure 11-8. HTML page with table of contents

11.5.2.5 Adding the table of contents automatically

To make life as easy as possible, we can modify the *site/wrapper* template to automatically wrap the page in the *page/tocpage* template. So that we have some control over which pages this is applied to, we will add a new page type, tocpage. Example 11-69 includes a new CASE for this page type that adds page/tocpage to the list of wrappers for the page.

Example 11-69. Adding a tocpage page type to site/wrapper

```
[%  SWITCH page.type;

        CASE 'text';

            content;


        CASE 'html';

            content WRAPPER site/html

                        + site/layout;


        CASE 'tocpage';

            content WRAPPER site/html

                        + site/layout

                        + page/tocpage;


        CASE;

            THROW page.type "Invalid page type: $page.type";

    END;

-%]
```

With this in place, there is no need for a page to explicitly wrap itself in the *page/tocpage* template. Instead, it should define a type of tocpage in a META directive and leave it to *site/wrapper* to add the table of contents:

```
[% META type  = 'tocpage'

        title = 'About the Template Toolkit'

        about = 'A brief overview of and introduction

                 to the Template Toolkit'

%]


[% Section('Overview') %]

   ...etc...
```

11.5.3 Declarative Markup Using XML

The Template Toolkit allows you to decouple your core content from any particular presentation style. However, the techniques that we've shown in this section are very much specific to the Template Toolkit and to a particular way of generating pages.

That isn't going to be a problem in many cases, but you might prefer to define your content in a format that can be read and manipulated by other tools as well as by the Template Toolkit. XML is of course the perfect example of an open format that you might like to use.

XML allows you to write declarative markup instead of the more procedural markup of the Template Toolkit. Rather than embedding a set of instructions in the document that say "add a section header here" or "generate a table of contents over there," XML simply states things for the record. It says "this is a section" or "this is a subsection," and allows you to do what you like with the information.

The Template Toolkit is quite happy working with XML. It will do the hard work of transforming it into HTML, using template components to apply the current presentation style for your site along the way.

11.5.3.1 XML page content

Example 11-70 shows a page template that uses XML to define the core content.

Example 11-70. XML page template

```
[% META type  = 'xml'
       title = 'About the Template Toolkit'
       about = 'A brief overview of and introduction
                to the Template Toolkit'
%]

<page>
  <section title="Overview">
    <p>
      The <b>Template Toolkit</b> is a fast,
      powerful, and easily extensible template
      processing system written in <b>Perl</b>...
    </p>

    <subsection title="Features">
      <ul>
        <li>Fast, powerful, and so on...</li>
      </ul>
    </subsection>
  </section>

  <section title="Mailing Lists">
    <p>
      A number of mailing lists are provided for discussing
      the Template Toolkit.
    </p>

    <subsection title="templates">
      <p>
        The <b>templates</b> mailing list...
```

```
    </p>
  </subsection>

  <subsection title="templates-announce">
    <p>
      The <b>templates-announce</b> mailing list...
    </p>
  </subsection>
 </section>
</page>
```

The page content is enclosed within a <page> element. Sections and subsections are declared using the appropriate <section> and <subsection> elements. We can include any kind of valid XHTML markup within these elements.

11.5.3.2 XML page wrapper

A minor change is required to our presentation framework for it to handle XML files. We've declared the page type for Example 11-70 to be xml in the META tag. We must therefore add the appropriate handler to the *site/wrapper* template, as shown in Example 11-71.

Example 11-71. Adding an XML page type to site/wrapper

```
[%  SWITCH page.type;
       CASE 'text';
          content;

       CASE 'html';
          content WRAPPER site/html
                        + site/layout;

       CASE 'tocpage';
          content WRAPPER site/html
                        + site/layout
                        + page/tocpage;

       CASE 'xml';
          content WRAPPER site/html
                        + site/layout
                        + site/xmlpage;

       CASE;
          THROW page.type "Invalid page type: $page.type";
   END;
```

```
-%]
```

The *site/xmlpage* template is used as an additional wrapper to process XML page content. Example 11-72 shows how it works.

Example 11-72. templates/lib/site/xmlpage

```
[%  USE xmldoc  = XML.XPath( text = content );

    USE xmlview = view(
          prefix   = 'xmlpage/'
          notfound = 'xmltag'
    );


    FOREACH xnode = xmldoc.findnodes('/page');
        xmlview.print(xnode);
    END;
-%]
```

It uses the XML.XPath plugin, passing the XML content of the page as the text variable. The plugin then returns an object through which we can query the XML document, assigned to the xmldoc variable:

```
USE xmldoc  = XML.XPath( text = content );
```

It then creates a VIEW plugin object called xmlview. This will be used to map XML elements to corresponding templates in the *xmlpage/* subdirectory of *templates/lib*. The xmltag template will be used to render any XML elements for which no template is defined:

```
USE xmlview = view(
      prefix   = 'xmlpage/'
      notfound = 'xmltag'
);
```

The final section iterates through each page node,[6] calling on the xmlview view to print it using the appropriate template:

[6] There's only one in this case, but findnodes returns a list anyway.

```
FOREACH xnode = xmldoc.findnodes('/page');
    xmlview.print(xnode);
END;
```

11.5.3.3 XML view templates

The view first calls the *xmltag/page* template to process the outermost page XML node. It calls the item.content method passing the current view as an argument. This generates the view-specific content for the page that can then be wrapped using the existing *page/tocpage* template to add a table of contents (see Example 11-73).

Example 11-73. templates/lib/xmlpage/page

```
[% item.content(view)
     WRAPPER page/tocpage
-%]
```

The call to item.content(view) causes the view to iterate over the content of the page XML node. In this case, it will find section nodes, which are sent off to the *xmlpage/section* for processing (see Example 11-74).

Example 11-74. templates/lib/xmlpage/section

```
[%  Section( item.getAttribute('title') );

    item.content(view)

-%]
```

This template calls the Section macro, fetching the value for the title from the XML title attribute. The section content is then displayed, again by calling the item.content method.

The *xmlpage/subsection* template is called whenever a subsection XML element is encountered. It is almost identical to *xmlpage/section*, as shown in Example 11-75.

Example 11-75. templates/lib/xmlpage/subsection

```
[%  Subsection( item.getAttribute('title') );

    item.content(view)

-%]
```

Whenever the view finds an XML element that it doesn't have a template for, it calls on xmlpage/xmltag, which regenerates the original XML element. This allows us to pass XHTML content through without it requiring any further transformation (see Example 11-76).

Example 11-76. templates/lib/xmlpage/xmltag

```
[% item.starttag;

   item.content(view);

   item.endtag

-%]
```

We also need a simple template to reproduce any plain-text parts as they are (see Example 11-77).

Example 11-77. templates/lib/xmlpage/text

```
[% item -%]
```

That's all there is to it. Any time you want to define some specific handling for an XML element, simply add the appropriately named template to the *templates/lib/xmlpage* directory. The view will take care of the rest.

These simple templates don't do much in themselves. They just provide the glue between XML.XPath nodes and our existing Section and Subsection macros. We get to reuse all of our existing presentation framework, but can now define content in XML, HTML, and various other formats, all of which can be freely intermixed with Template Toolkit directives.

11.6 Creating a New Skin

In the final section of this chapter, we are going to show how a new set of template components can be created to rebrand, or *skin*, the site. Rather than modify our existing components, we will create a new set in a different directory. For these examples, the directory will be *templates/skin/droplet*, relative to the current project directory of */home/dent/web/ttbook*. We can create as many different skins as required as long as each has its own unique name and corresponding component directory. The name we are using for this skin is droplet, for no reason in particular.

11.6.1 Creating a Skin

First, we must create a directory for the skin-specific templates:

```
$ cd /home/dent/web/ttbook

$ mkdir templates/skin

$ mkdir templates/skin/droplet
```

The INCLUDE_PATH configuration option and the corresponding lib option for *ttree* allow multiple directories to be specified for the location of template files. The *templates/skin/droplet* directory should be added to *etc/ttree.cfg* as a new lib option coming before the existing one. Example 11-78 shows the new line added to the first block of the *etc/ttree.cfg* file.

Example 11-78. Adding a lib option to etc/ttree.cfg

```
src  = /home/dent/web/ttbook/templates/src

# add lib option for new skin

lib  = /home/dent/web/ttbook/templates/skin/droplet

lib  = /home/dent/web/ttbook/templates/lib

dest = /home/dent/web/ttbook/html
```

You may also want to update the corresponding skeleton template, *skeleton/etc/ttree.cfg*. Or you can update the skeleton file and then run the *bin/configure* script to have it regenerate *etc/tree.cfg*.

We will need to define some configuration data for the new skin, so we create a *config/skin* template and add it to the list in config/main (see Example 11-79).

Example 11-79. Adding config/skin to config/main

```
[%  PROCESS config/page

        + config/site

        + config/url

        + config/col

        + config/images

        + config/map

        + config/macros

        + config/skin     # add this line

-%]
```

The *config/skin* configuration template for the droplet skin is shown in Example 11-80. It defines a URL, some colors, and other information relating to a set of icons that will be used by various template components.

Example 11-80. templates/skin/droplet/config/skin

```
[%  site.url.icon = "$site.url.images/icon"
    site.col.icon = {
        on   = 'orange'
        off  = 'blue'
        roll = 'red'
        dead = 'gray'
    }
    site.image.icon = {
        large = {
            url    = "$site.url.icon/large"
            src    = "$site.url.icon/large/blue/dot.png"
            alt    = 'dot icon'
            width  = 36
            height = 36
        }
        small = {
            url    = "$site.url.icon/small"
            src    = "$site.url.icon/small/blue/dot.png"
            alt    = 'dot icon'
            width  = 24
            height = 24
        }
        tiny = {
            url    = "$site.url.icon/tiny"
            src    = "$site.url.icon/tiny/blue/dot.png"
            alt    = 'dot icon'
            width  = 18
            height = 18
        }
    }
-%]
```

In case we later decide to generate the site without this skin, we must also provide a dummy *config/skin* template in the default *templates/lib* directory (see Example 11-81).

Example 11-81. templates/lib/config/skin

```
[%# hook for skins to perform any
  # additional extra configuration
-%]
```

11.6.2 Custom Navigation Components

Now we can add our own custom components to the *templates/skin/droplet* directory. They will be used in preference to those in the default *templates/lib* directory.

We can start by defining a new *misc/line* component, as shown in Example 11-82.

Example 11-82. templates/skin/droplet/misc/line

```
<table border="0" width="100%" cellpadding="0" cellspacing="0">
  <tr>
    <td height="1" bgcolor="[% site.col.line %]"><img
        width="1" height="1" /></td>
  </tr>
</table>
```

The design of this skin is based around some simple icons. Example 11-83 shows a template component to generate the HTML for the various icons we are using.

Example 11-83. templates/skin/droplet/misc/icon

```
[%  # misc/icon - generate image tag for icon

    DEFAULT
      size = 'small'
      icon = 'dot'
      col  = 'blue';

    IF (image = site.image.icon.$size);
        PROCESS misc/image
          image.src  = "$image.url/$col/${icon}.png"
          image.alt  = "$icon icon";
    ELSE;
        THROW logo "invalid icon size: $size";
    END;
-%]
```

11.6.2.1 Nested menu

The *misc/icon* template can be used to spice up the *menu/nest* template that we introduced in Example 11-68. The new version can be seen in Example 11-84.

Example 11-84. templates/skin/droplet/menu/nest

```
[%  DEFAULT
        global.linkno = 0
        icon = site.image.icon.tiny;
```

```
    colroll = site.col.icon.roll;

    WRAPPER menu/table;
      FOREACH item = menu.items;
        linkno  = (global.linkno = global.linkno + 1);
        colicon = item.hot ? site.col.icon.on
                          : site.col.icon.off;

        INCLUDE menu/link
          link = {
            name     = "menu_$linkno"
            text     = item.name
            url      = item.url
            icon     = "$icon.url/$colicon/right.png"
            rollover = "$icon.url/$colroll/right.png"
            size     = icon.width
            class    = item.hot ? 'menuselect' : 'menu'
          };

        INCLUDE menu/submenu menu=item
          IF item.subs;
      END;
    END;
-%]
```

Figure 11-9 shows a screenshot of a page containing the droplet-style nested menu.

Figure 11-9. Droplet-style nested menu

11.6.2.2 Menu elements

Various HTML table elements and other components are used to generate the menu in this style. They have been moved into the templates shown in Examples 11-85 through 11-93 to promote modularity and to help keep the *menu/nest* template clutter-free.

Example 11-85. templates/skin/droplet/menu/table

```
<table border="0" cellpadding="0" cellspacing="2">
[%- content -%]
</table>
```

Example 11-86. templates/skin/droplet/menu/row

```
<tr valign="middle">
[%- content -%]
</tr>
```

Example 11-87. templates/skin/droplet/menu/blank

```
<tr>
  <td></td>
  <td> </td>
</tr>
```

Example 11-88. templates/skin/droplet/menu/line

```
<tr>
  <td colspan="2">[%- PROCESS misc/line -%]</td>
</tr>
```

Example 11-89. templates/skin/droplet/menu/name

```
[% PROCESS menu/blank -%]
<tr>
  <td colspan="2" class="menutitle">[% menu.name %]</td>
</tr>
[% PROCESS menu/line -%]
```

Example 11-90. templates/skin/droplet/menu/link

```
<tr valign="middle">
  <td align="middle" width="[% item.size %]" height="[% item.size %]">
[%- PROCESS menu/icon -%]</td>
  <td align="left">
[%- PROCESS menu/text -%]</td>
</tr>
```

Example 11-91. templates/skin/droplet/menu/submenu

```
<tr>
    <td></td>
    <td>
      [% PROCESS menu/nest %]
    </td>
  </tr>
```

Example 11-92. templates/skin/droplet/menu/icon

```
<a href="[% link.url %]"
[% IF link.target -%]
   target="[% link.target %]"
[% END -%]
[% IF link.rollover -%]
   onmouseover="[% link.name %].src = '[% link.rollover %]';"
   onmouseout="[% link.name %].src = '[% link.icon %]';"
[% END -%]
><img
   name="[% link.name %]" src="[% link.icon %]"
   width="[% link.size %]" height="[% link.size %]" border="0" /></a>
```

Example 11-93. templates/skin/droplet/menu/text

```
<a href="[% link.url %]"
[% IF link.class -%]
   class="[% link.class %]"
[% END -%]
[% IF link.target -%]
   target="[% link.target %]"
[% END -%]
[% IF link.rollover -%]
   onmouseover="[% link.name %].src = '[% link.rollover %]';"
```

```
   onmouseout="[% link.name %].src = '[% link.icon %]';"
[% END -%]
>
[%- link.text -%]
</a>
```

11.6.2.3 Stacked menu

We can also create a new version of the stacked menu by reusing these menu components, as shown in Example 11-94.

Example 11-94. templates/skin/droplet/menu/stack

```
[%  DEFAULT
        global.linkno = 0
        icon = site.image.icon.tiny;

    pending = [ menu ];
    colroll = site.col.icon.roll;

    WRAPPER menu/table;
        WHILE pending.size;
            menu = pending.shift;

            FOREACH item = menu.items;
                linkno  = (global.linkno = global.linkno + 1);
                colicon = item.hot ? site.col.icon.on
                                   : site.col.icon.off;

                INCLUDE menu/link
                    link = {
                        name     = "item_$linkno"
                        text     = item.name
                        url      = item.url
                        icon     = "$icon.url/$colicon/right.png"
                        rollover = "$icon.url/$colroll/right.png"
                        size     = icon.width
                        class    = item.hot ? 'menuselect' : 'menu'
                    };

                pending.push(item)
```

```
            IF item.subs;
        END;

        PROCESS menu/name menu=pending.first
          IF pending.size;
      END;

    END;
-%]
```

Figure 11-10 shows the end result.

Figure 11-10. Nested menu

11.6.2.4 Other page components

To complete the set, we can also define new templates for the bread-crumb trail, the next and previous page menu, and the page sections and subsections (see Examples 11-95 through 11-98).

Example 11-95. templates/skin/droplet/menu/trail

```
[%  DEFAULT
      icon  = site.image.icon.tiny;
      page.linkno = 0;

    colicon = site.col.icon.off;
    colroll = site.col.icon.roll;

    WRAPPER menu/table
```

```
          + menu/row;

        FOREACH item IN trail;
           INCLUDE menu/trail/crumb
              link = {
                  name     = "trail_$loop.count"
                  text     = item.name
                  url      = item.url
                  icon     = "$icon.url/$colicon/right.png"
                  rollover = "$icon.url/$colroll/right.png"
                  size     = icon.width
                  class    = 'menu'
              };
        END;
    END;
-%]

[%- BLOCK menu/trail/crumb -%]
  <td align="middle" width="[% item.size %]" height="[% item.size %]">
[%- PROCESS menu/icon -%]</td>
  <td align="left">
[%- PROCESS menu/text -%]</td>
[%- END -%]
```

Example 11-96. templates/skin/droplet/menu/prevnext

```
[%  size  = 'tiny'
    icon  = site.image.icon.$size
    width = icon.width;

    colicon = site.col.icon.off;
    colroll = site.col.icon.roll;

    WRAPPER menu/table
          + menu/row;
%]

[% # is there a previous page?
   IF page.prev;
      link = {
```

```
            name     = "prev"
            text     = page.prev.name
            url      = page.prev.url
            icon     = "$icon.url/$colicon/left.png"
            rollover = "$icon.url/$colroll/left.png"
            size     = icon.width
            class    = 'menu'
        };
-%]
    <td align="right">
      [%- PROCESS menu/text -%]
    </td>
    <td width="[% width %]">
      [%- PROCESS menu/icon -%]
    </td>
[% ELSE %]
    <td></td>
    <td width="[% width %]">
      [%- INCLUDE misc/icon
            size = 'tiny'
            col  = site.col.icon.dead
            icon = 'left'
      %]
    </td>
[% END %]

    <td width="[% width %]">
      [%- INCLUDE misc/icon
            col  = site.col.icon.on
            icon = 'dot'
            size = 'tiny'
      -%]
    </td>

[% # is there a next page?
   IF page.next;
       link = {
            name     = "next"
            text     = page.next.name
```

```
            url      = page.next.url
            icon     = "$icon.url/blue/right.png"
            rollover = "$icon.url/red/right.png"
            size     = icon.width
            class    = 'menu'
        };
-%]
    <td width="[% width %]">
      [%- PROCESS menu/icon -%]
    </td>
    <td align="left">
      [%- PROCESS menu/text  -%]
    </td>
[% ELSE %]
    <td width="[% width %]">
      [%- INCLUDE misc/icon
            col  = site.col.icon.dead
            icon = 'right'
            size = 'tiny'
      %]
    </td>
    <td></td>
[% END %]

[% END   # WRAPPER %]
```

Example 11-97. templates/skin/droplet/page/section

```
[%  size = 'small';
    imgsize = site.image.icon.$size;
-%]
<p>
<table width="100%" border="0" cellpadding="0" cellspacing="4">
  <tr valign="middle">
    <td width="[% imgsize.width %]">
      [%- PROCESS misc/icon %]</td>
    <td align="left" width="100%">
      <a name="[% id %]"><b class="section">[% title %]</b></a>
```

```
    </td>
    <td align="right">
     [%- UNLESS no_top %]
     <a href="#top">[%
         INCLUDE misc/icon
           size = 'small'
           icon = 'up'
           col  = site.col.icon.off
      %]</a>
      [% END %]
    </td>
  </tr>
  <tr>
    <td></td>
    <td colspan="2">
     [% PROCESS misc/line %]
    </td>
  </tr>
  <tr valign="top">
    <td></td>
    <td colspan="2">
     [% content %]
    </td>
  </tr>
</table>
</p>
```

Example 11-98. templates/skin/droplet/page/subsection

```
[%  size = 'tiny';
    imgsize = site.image.icon.$size;
-%]
<p>
<table width="100%" border="0" cellpadding="0" cellspacing="4">
  <tr valign="middle">
    <td width="[% imgsize.width %]">
     [%- PROCESS misc/icon %]</td>
    <td align="left">
     <a name="[% id %]"><b class="subsection">[% title %]</b></a>
```

```
    </td>
  </tr>
  <tr valign="top">
    <td></td>
    <td>
      [% content %]
    </td>
  </tr>
</table>
```

Figure 11-11 is what it looks like when it is all put together. Remember that none of the core content has changed, only the template components that handle the presentation.

Figure 11-11. Complete page in the droplet style

Chapter 11. Advanced Static Web Page Techniques

In Chapter 2, we looked at some simple examples of using the Template Toolkit to generate web content. In this chapter, we will look at some more advanced techniques for building web sites and manipulating HTML page content. We will start out with a minimal setup that illustrates some useful techniques that can easily be adapted and applied to any web site. The basic system will be extended throughout the chapter as we add functionality to address more complex requirements and provide more advanced features.

The emphasis in this chapter will be on generating static HTML web content. The examples will be loosely based around the Template Toolkit web site, http://template-toolkit.org/. However, we're not going to be looking at content of any of the individual pages in any great detail, so the subject matter is largely immaterial.

Most of the techniques demonstrated are equally applicable to web sites delivering dynamically generated content and running web applications. More generally, this chapter shows how a general-purpose *presentation framework* can be built using the Template Toolkit. This can then be used to apply a consistent look and feel to all pages in a site, including static HTML pages (as discussed later in this chapter) and dynamic content (described in Chapter 12).

12.1 CGI Scripts

The Common Gateway Interface (CGI) provides a simple mechanism for generating dynamic web content and running web applications. The web server receives a request and maps it to a CGI program, which is then run. These are often located in a special *cgi-bin* directory or have a particular file extension such as *.cgi*. Various parameters relating to the CGI request are passed to the program as environment variables. Additional data may be piped in through the program's standard input in the case of a POST request. The program does whatever it needs to do in the way of application processing, and then prints a simple header and then the content of the page to standard output. The web server sends this back to the client's browser as the response.

Perl is a very popular language for writing CGI scripts. The CGI module provides a wealth of functionality for CGI programming. For a full tour of CGI programming and the CGI module, see *CGI Programming with Perl* by Scott Guelich, Shishir Gundavaram, and Gunther Birznieks (O'Reilly).

12.1.1 Simple CGI Script

Using the Template Toolkit in a CGI script is easy. The Template `process( )` method prints its output to STDOUT by default. For simple cases, very little work is required on our part to turn any Perl program using the Template Toolkit into a CGI script. Example 12-1 shows such a script.

Example 12-1. ttcgi1.pl

```
#!/usr/bin/perl


use strict;
use warnings;
use Template;


$| = 1;
print "Content-type: text/html\n\n";


my $tt    = Template->new( );
my $input = 'destruction1.html';
my $vars  = {
    planet  => 'Earth',
    captain => 'Prostetnic Vogon Jeltz',
    time    => 'two of your earth minutes',
};


$tt->process($input, $vars)
    || die $tt->error( );
```

The only lines that are specific to CGI programming are these:

```
$| = 1;
print "Content-type: text/html\n\n";
```

The first of these lines disables buffering on standard output. This ensures that any content printed is sent back to the client right away. The second line prints a standard CGI header, telling the browser that we're sending it an HTML page. The other difference between this example and the simple text version that we first saw in Chapter 1 is that our template must now be marked up as valid HTML, as shown in Example 12-2.

Example 12-2. destruction1.html

```
<html>
 <head>
  <title>Destruction of [% planet %] is Imminent!</title>
 </head>
 <body>
  <p>
    People of [% planet %], your attention please.
  </p>
  <p>
    This is [% captain %] of the
    Galactic Hyperspace Planning Council.
  </p>
  <p>
    As you will no doubt be aware, the plans
    for development of the outlying regions
    of the Galaxy require the building of a
    hyperspatial express route through your
    star system, and regrettably your planet
    is one of those scheduled for destruction.
  </p>
  <p>
    The process will take slightly less than
    [% time %].
  </p>
 </body>
</html>
```

12.1.1.1 Using standard templates

The Template Toolkit provides a set of standard templates for adding HTML headers and footers to pages. On Unix systems, they are typically installed in */usr/local/tt2/templates*. On Windows platforms, they are installed in *C:\Program Files\Template Toolkit 2\templates*. The *Template::Config* module provides the `instdir( )` method to determine the location in a portable way. By adding this directory to the `INCLUDE_PATH` configuration option, we can then use the standard *html/page* template as a `WRAPPER` for the page, as shown in Example 12-3.

Example 12-3. ttcgi2.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;
use Template::Config;

$| = 1;
print "Content-type: text/html\n\n";

my $tdir  = Template::Config->instdir('templates');
my $tt    = Template->new({
    INCLUDE_PATH => [ '.', $tdir ],
    WRAPPER      => 'html/page'
});
my $input  = 'destruction2.html';
my $vars   = {
    planet  => 'Earth',
    captain => 'Prostetnic Vogon Jeltz',
    time    => 'two of your earth minutes',
    html    => {
        head => {
            title => "Destruction of Earth is Imminent!",
        },
    },
};

$tt->process($input, $vars)
    || die $tt->error( );
```

The location of the *templates* directory is determined by the following line and stored in the $tdir variable:

```
my $tdir  = Template::Config->instdir('templates');
```

The $tdir directory is then added to the INCLUDE_PATH, along with the current working directory (.):

```
my $tt    = Template->new({
    INCLUDE_PATH => [ '.', $tdir ],
    WRAPPER      => 'html/page'
});
```

The *html/page* wrapper template adds the <html>, <head>, and <body> elements around the generated page content. It inserts the value of the html.head.title variable in the <title> of the <head> element, to set the page title. Accordingly, we define an appropriate title in the $vars hash:

```
my $vars   = {
    planet  => 'Earth',
    captain => 'Prostetnic Vogon Jeltz',
    time    => 'two of your earth minutes',
    html    => {
        head => {
            title => "Destruction of Earth is Imminent!",
        },
    },
};
```

The *destruction2.html* template can now be made much simpler, as shown in Example 12-4. The HTML headers and footers are all added automatically, leaving us to concentrate on the content. We're also using the html_para filter to add the <p> and </p> tags around each paragraph.

Example 12-4. destruction2.html

```
[% FILTER html_para %]
People of [% planet %], your attention please.


This is [% captain %] of the
Galactic Hyperspace Planning Council.


As you will no doubt be aware, the plans
for development of the outlying regions
of the Galaxy require the building of a
hyperspatial express route through your
star system, and regrettably your planet
is one of those scheduled for destruction.


The process will take slightly less than
[% time %].
[% END %]
```

If you've been working through the examples in Chapter 11, you'll probably have developed your own wrappers and other user interface templates that you can use in place of *html/page*.

12.1.2 Using the DATA Section

You can also define the main page template in a DATA section following the main part of the CGI script, as shown in Example 12-5.

Example 12-5. ttcgi3.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;

$| = 1;
print "Content-type: text/html\n\n";

my $tt   = Template->new({
    INCLUDE_PATH => '/home/dent/vogon/templates',
    WRAPPER      => 'vogon/page'
});
my $vars   = {
    planet  => 'Earth',
    captain => 'Prostetnic Vogon Jeltz',
    time    => 'two of your earth minutes',
};

$tt->process(\*DATA, $vars)
    || die $tt->error( );

_ _DATA_ _
[% FILTER html_para %]
People of [% planet %], your attention please.

This is [% captain %] of the
Galactic Hyperspace Planning Council.

As you will no doubt be aware, the plans
for development of the outlying regions
of the Galaxy require the building of a
hyperspatial express route through your
star system, and regrettably your planet
is one of those scheduled for destruction.
```

```
The process will take slightly less than

[% time %].

[% END %]
```

The _ _DATA_ _ (or _ _END_ _) marker indicates the point where the script stops and the template starts. Perl provides the DATA filehandle to read the text from this block. We pass a reference to the filehandle as the first argument to the process( ) method and leave it to do the rest:

```
$tt->process(\*DATA, $vars)

    || die $tt->error( );
```

The approach is great for small and simple CGI scripts. It allows you to keep everything together and contained in one file. You can see both the Perl code and the main page template in the same place, but they are still kept nicely separate from each other. Other components or layout templates such as *html/page* or the hypothetical *vogon/page* wrapper used in this example can be kept out of the way in separate files so that they don't obstruct the core content and can be reused between different CGI scripts.

Be warned that you can't use the DATA section if you want to run your CGI scripts under Apache::Registry. Apache:Registry allows you to run unaltered CGI scripts under mod_perl for a significant speedup. Instead of being loaded and compiled each time a request is made, the script is kept in compiled form in the memory space of the web server. It can then be executed quickly and repeatedly on demand.

However, a CGI script gets only one chance to read the DATA section. When it has been read once, there is no going back to read it again. If you plan to use Apache::Registry, you should use separate page template files rather than embedding them in a DATA section.

12.1.3 Using the CGI Module

The CGI module does everything you'll ever need to in CGI programming and a whole lot more. Example 12-6 shows how we create a CGI object and pass it to the template as the cgi variable.

Example 12-6. ttcgi4.pl

```
#!/usr/bin/perl



use strict;

use warnings;

use Template;

use CGI;



$| = 1;



my $cgi   = CGI->new( );

my $tt    = Template->new( );

my $input = 'cgiparams.html';

my $vars  = {

    cgi   => $cgi,
```

```
};

print $cgi->header;

$tt->process($input, $vars)
    || die $tt->error( );
```

The template processed by the script, *cgiparams.html*, is shown in Example 12-7. It calls the param( ) method of the CGI object first to fetch a list of request parameters, and then again to fetch the value for each parameter within the FOREACH loop.

Example 12-7. cgiparams.html

```
<h1>CGI Parameters</h1>

<ul>
[% FOREACH p = cgi.param -%]
   <li><b>[% p %]</b> [% cgi.param(p) %]</li>
[% END -%]
</ul>
```

Example 12-8 shows some typical output generated by the CGI script. In this case, the request URL used was */cgi-bin/ttcgi4.pl?pi=3.14&e=2.718&message=Hello%20World*. We didn't add any HTML page wrapper in this example to keep things simple. But that would of course be required for any CGI script operating in the real world.

Example 12-8. Output of cgiparams.html

```
<h1>CGI Parameters</h1>

<ul>
   <li><b>pi</b> 3.14</li>
   <li><b>e</b> 2.718</li>
   <li><b>message</b> Hello World</li>
</ul>
```

If you want to use the CGI object to manipulate headers, cookies, or anything else outside of generating content, you'll probably need to do it in the calling CGI script.

12.1.3.1 Setting cookies

Let's look at an example of how cookies can be set using values supplied from within a template. We start by defining a cookies template variable in the CGI script as a reference to an initially empty list. This will be used to store any cookies that should be added to the CGI header.

```
my @cookies;
my $vars  = {
    cgi     => $cgi,
    cookies => \@cookies,
};
```

The CGI object provides the cookie method for creating cookies. We call this from within the template to create a cookie object.

```
[% cookie = cgi.cookie(
     name    = 'SessionID',
     value   = 12345678,
     expires = '+1m'
   )
%]
```

The newly created cookie is then pushed onto the cookies list:

```
[% cookies.push(cookie) %]
```

Back in the CGI script, we need to process the template first and then check to see whether any cookies have been added to the list. Cookies must be added to the response header before any content is sent back to the client. Rather than let the Template process( ) method print its output directly to standard output, we provide it with a reference to an $output variable. This is used to store the generated HTML page until we have set the cookie headers and are ready to send a response back to the client.

```
my $output;

$tt->process($input, $vars, \$output)
    || die $tt->error( );
```

Then we check for any cookies and provide them as an option to the CGI header( ) method before printing the page content stored in $output:

```
if (@cookies) {
    @cookies = ("-cookie", [ @cookies ]);
}
print $cgi->header(@cookies), $output;
```

The complete CGI script is shown in Example 12-9.

Example 12-9. ttcgi5.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;
use CGI;

$| = 1;

my $cgi   = CGI->new( );
my $tt    = Template->new( );
my $input = 'cgicookie.html';
my @cookies;
my $vars  = {
    cgi     => $cgi,
```

```
    cookies => \@cookies,
};
my $output;

$tt->process($input, $vars, \$output)
    || die $tt->error( );

if (@cookies) {
    @cookies = ('-cookie', [ @cookies ]);
}

print $cgi->header(@cookies), $output;
```

The *cgicookie.html* template is listed in Example 12-10.

Example 12-10. cgicookie.html

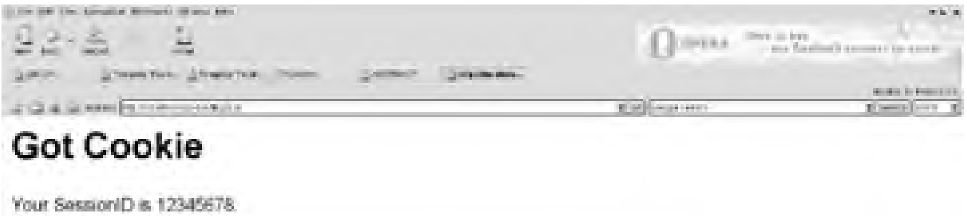
```
[% IF (cookie = cgi.cookie('SessionID')) %]

  <h1>Got Cookie</h1>

  <p>
    Your SessionID is [% cookie %].
  </p>

[% ELSE %]
  [% cookie = cgi.cookie(
        name    = 'SessionID',
        value   = 12345678,
        expires = '+1m'
     );
     cookies.push(cookie)
  %]

  <h1>Set Cookie</h1>

  <p>
    Cookie has been set.  Please reload page.
  </p>

[% END %]
```

Figure 12-1 shows the cookie being set the first time we access the page. We've enabled a feature on our browser that displays the details of each cookie being set so that we can confirm that the CGI script is working as expected.

Figure 12-1. cookieset.png

When the page is reloaded, the cookie is read and the value for SessionID printed, as shown in Figure 12-2.

Figure 12-2. cookieget.png



12.1.4 CGI Script Web Application

Now we're going to look at an example of a more complete CGI script that provides a simple web interface to a database containing entries for a fictional travel guide. Each entry has a name (e.g., Earth) as well as a unique numerical identifier (e.g., 42). We would like to be able to display an entry from the guide by specifying either the name or id. We would also like to be able to search the database to help find entries of interest. We'll be using MySQL in this example, but the techniques apply to any relational database.

12.1.4.1 CGI script

Let's start by walking through the CGI script to explain what each section of code does.

12.1.4.1.1 Preparation

The CGI script starts with the usual preamble. We first load the various modules that we are going to use:

```
#!/usr/bin/perl


use strict;

use warnings;


use DBI;
```

```
use CGI;

use CGI::Carp qw(fatalsToBrowser);

use Template;

$| = 1;
```

Then we define some configuration data:

```
my $ROOTDIR = '/home/dent/web/guide';

my $ROOTURL = '/~dent/guide';

my $ROOTCGI = '/cgi-bin/dent/guide.pl';

my $DBDSN   = 'DBI:mysql:guide';

my $DBUSER  = 'dent';

my $DBPASS  = 'ruhtra';
```

More preparation follows as we create a CGI object, make a connection to the database, and declare some variables, including the $vars hash containing template variables. The $template variable is used to store the name of the template that is processed to generate the page content. We'll be setting it shortly.

```
my $cgi = CGI->new( );

my $dbh = DBI->connect($DBDSN, $DBUSER, $DBPASS)

    || die "failed to connect to database: $DBI::errstr";



my ($param, $template);

my $vars = {

    rootdir => $ROOTDIR,

    rooturl => $ROOTURL,

    rootcgi => $ROOTCGI,

};
```

12.1.4.1.2 Application

Now we can get down to the application processing phase. The flow of control is determined by one of the request parameters being provided—name, id, or search. The if ... elsif ... else construct selects the right block of code accordingly.

```
if ($param = $cgi->param('name')) {

    # ...

}

elsif ($param = $cgi->param('id')) {

    # ...

}

elsif ($param = $cgi->param('search')) {

    # ...

}

else {

    # ...

}
```

If a name parameter is provided, the appropriate SELECT query is sent to the database. The entry is returned as a reference to a hash array, hopefully without error,[1] and is added to the $vars hash as the entry template variable. The $template variable is then set to entry.html.

[1] Note the use of the CGI::Carp module. This will catch our calls to die and generate an HTML page for sending back to the browser.

```
if ($param = $cgi->param('name')) {

    my $entry = $dbh->selectrow_hashref(

        "SELECT id, name, author, about, date

         FROM entry WHERE name=?", { }, $param)

        || die $DBI::errstr;

    $vars->{ entry } = $entry;

    $template = 'entry.html';

}
```

The handling of the id parameter is much the same as it is for name:

```
elsif ($param = $cgi->param('id')) {

    my $entry = $dbh->selectrow_hashref(

        "SELECT id, name, author, about, date

         FROM entry WHERE id=?", { }, $param)

        || die $DBI::errstr;

    $vars->{ entry } = $entry;

    $template = 'entry.html';

}
```

The search parameter requires a slightly different process to allow for the multiple entries that can be returned. Here the entries template variable is set to contain the list of entries returned, each of which is a hash reference, and the $template is set to entries.html:

```
elsif ($param = $cgi->param('search')) {

    $vars->{ search } = $param;

    $param =~ s/*/\%/g;  # change '*' to '%'

    my $sth = $dbh->prepare(

        'SELECT id, name, author, about, date

         FROM entry WHERE name LIKE ?' )

        || die $DBI::errstr;

    $sth->execute($param) || die $sth->errstr( );

    $vars->{ entries } = $sth->fetchall_arrayref({  });

    $template = 'entries.html';

}
```

This application allows the user to specify wildcards in a pattern using the * character—e.g., ear*. MySQL, on the other hand, uses % to denote wildcards. To cater for this, the appropriate transformation is made to the search term in $param before it is used in the query. A copy of the original search term is saved as the search template variable.

```
$vars->{ search } = $param;

$param =~ s/*/\%/g;  # change '*' to '%'
```

If none of the name, id, or search parameters is provided, the index page is displayed:

```
else {
    $template = 'index.html';
}
```

12.1.4.1.3 Presentation

At this point, the $template variable tells us which template needs to be processed, and $vars contains any variables required to process it. We create a Template object specifying various options indicating the location of templates, and naming a template for preprocessing (*config*) and another for wrapping around the page content (*wrapper*).

```
my $tt = Template->new({
    INCLUDE_PATH => [
        "$ROOTDIR/templates/cgi",
        "$ROOTDIR/templates/lib",
    ],
    PRE_PROCESS => 'config',
    WRAPPER     => 'wrapper',
});
```

Then we print the CGI header and process the template to generate the dynamic HTML page content:

```
print $cgi->header( );

$tt->process($template, $vars)
    || die $tt->error( );
```

All done! The complete CGI script is shown in Example 12-11.

Example 12-11. guide/cgi-bin/guide.pl

```
#!/usr/bin/perl

use strict;
use warnings;

use DBI;
use CGI;
use CGI::Carp qw(fatalsToBrowser);
use Template;
$| = 1;

#--------------------------------------------------
# configuration
#--------------------------------------------------

my $ROOTDIR = '/home/dent/web/guide';
```

```
my $ROOTURL = '/~dent/guide';
my $ROOTCGI = '/cgi-bin/dent/guide.pl';
my $DBDSN   = 'DBI:mysql:guide';
my $DBUSER  = 'dent';
my $DBPASS  = 'ruhtra';

my $cgi = CGI->new( );
my $dbh = DBI->connect($DBDSN, $DBUSER, $DBPASS)
    || die "failed to connect to database: $DBI::errstr";

my ($param, $template);
my $vars = {
    rootdir => $ROOTDIR,
    rooturl => $ROOTURL,
    rootcgi => $ROOTCGI,
};

#--------------------------------------------------
# application
#--------------------------------------------------

if ($param = $cgi->param('name')) {
    my $entry = $dbh->selectrow_hashref(
        "SELECT id, name, author, about, date
         FROM entry WHERE name=?", { }, $param)
        || die $DBI::errstr;
    $vars->{ entry } = $entry;
    $template = 'entry.html';
}
elsif ($param = $cgi->param('id')) {
    my $entry = $dbh->selectrow_hashref(
        "SELECT id, name, author, about, date
         FROM entry WHERE id=?", { }, $param)
        || die $DBI::errstr;
    $vars->{ entry } = $entry;
    $template = 'entry.html';
}
elsif ($param = $cgi->param('search')) {
    $vars->{ search } = $param;
```

```
    $param =~ s/*/\%/g;  # change '*' to '%'
    my $sth = $dbh->prepare(
        'SELECT id, name, author, about, date
         FROM entry WHERE name LIKE ?' )
        || die $DBI::errstr;
    $sth->execute($param) || die $sth->errstr( );
    $vars->{ entries } = $sth->fetchall_arrayref({  });
    $template = 'entries.html';
}
else {
    $template = 'index.html';
}

#------------------------------------------------------------------------
# presentation
#------------------------------------------------------------------------

my $tt  = Template->new({
    INCLUDE_PATH => [
        "$ROOTDIR/templates/cgi",
        "$ROOTDIR/templates/lib",
    ],
    PRE_PROCESS => 'config',
    WRAPPER     => 'wrapper',
});

print $cgi->header( );

$tt->process($template, $vars)
    || die $tt->error( );
```

12.1.4.2 Template components

The preprocessed *config* template, shown in Example 12-12, loads the Date plugin, defines a date MACRO that uses it, and then defines site and page data. See Chapter 11 for a full discussion on writing and using configuration templates.

Example 12-12. guide/templates/lib/config

```
[%  USE Date;

    MACRO date(d) BLOCK;
      # entry dates contain both date and
      # time, but we just want the date
      items = d.split('-');
      Date.format(
        "0:00:00 $items.2/$items.1/$items.0"
        format = '%d-%B-%Y'
      );
    END;

    site = {
      title     = "TT Hitch Hiker's Guide"
      admin     = 'webmaster@template-toolkit.org'
      copyright = '2003 Andy Wardley'
    }

    site.url = {
      guide  = rootcgi
      index  = "$rooturl/index"
      images = "$rooturl/images"
      css    = "$rooturl/css/tt2.css"
    }

    site.col = {
        back  = '#FFFFFF'  # white
        text  = '#000000'  # black
        line  = '#00AAF0'  # sky blue
    }

    site.logo = {
      src    = "$site.url.images/logo/tt2_120x40.gif"
      alt    = "TT2 Logo"
      width  = 120
      height = 40
```

```
    }

    page = {
      name  = template.name
      file  = template.name
      title = template.title
      about = template.about
      type  = template.type or 'html'
      date  = template.date or Date.format(template.modtime)
    }
-%]
```

Example 12-13 shows the *wrapper* template, which applies the *html* and *layout* templates as further wrappers around the generated page content. The use of wrapper templates is also discussed in Chapter 11.

Example 12-13. guide/templates/lib/wrapper

```
[%  SWITCH page.type;
        CASE 'text';
            content;

        CASE 'html';
            content WRAPPER html
                          + layout;
        CASE;
            THROW page.type "Invalid page type: $page.type";
    END;
-%]
```

The *html* and *layout* templates are shown in Examples Example 12-14 and Example 12-15, respectively.

Example 12-14. guide/templates/lib/html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
 <head>
  <title>
    [% site.title %]
    [% ": $page.title" IF page.title %]
  </title>
  <link rel="stylesheet"
        href="[% site.url.css %]" />
```

```
  <meta http-equiv="Content-Type"
        content="text/html; charset=iso-8859-1" />
 </head>

 <body bgcolor="[% site.col.back %]"
       text="[% site.col.text %]">
  [% content %]
 </body>
</html>
```

Example 12-15. guide/templates/lib/layout

```
[% MACRO line BLOCK -%]
  <tr>
    <td colspan="3">
      [% PROCESS line %]
    </td>
  </tr>
[%- END %]

<table width="100%" height="70%" border="0" cellpadding="4" cellspacing="4">
  <tr valign="middle">
    <td width="150" align="center">
      [% PROCESS logo %]
    </td>
    <td align="left">
      [% PROCESS header %]
    </td>
    <td align="right">
      [% PROCESS form %]
    </td>
  </tr>

  [% line %]

  <tr valign="top" height="100%">
    <td colspan="3">
      <!-- page content -->
      [% content %]
      <!-- end of page content -->
```

```
    </td>
  </tr>

  [% line %]

  <tr valign="bottom">
    <td colspan="3" align="center">
      [% PROCESS footer %]
    </td>
  </tr>
</table>
```

The *header* template uses the values defined in the page data structure to generate a page header, as shown in Example 12-16.

Example 12-16. guide/templates/lib/header

```
<h1 class="title">[% page.title %]</h1>
[% IF page.about -%]
<div class="info">
  [% page.about %]
</div>
[% END -%]
```

The *form* template, shown in Example 12-17, provides the search form. Any current value for the search template variable is displayed in the input field.

Example 12-17. guide/templates/lib/form

```
<form action="[% site.url.guide %]"
      method="POST" enctype="application/x-www-form-urlencoded">
  <table border="0">
    <tr valign="middle">
      <td>
        <input type="submit"
               name="submit"
               value=" Search " />
      </td>
      <td>
        <input type="text"
               name="search"
               size="30"
               value="[% search %]" />
```

```
      </td>
    </tr>
    <tr valign="middle">
      <td></td>
      <td class="info">
        e.g., <i>earth</i>, <i>magrethea</i>, <i>ear*</i>, <i>*th*</i>
      </td>
    </tr>
  </table>
</form>
```

The *logo* and *footer* templates, shown in Examples Example 12-18 and Example 12-19, respectively, also do what their names suggest.

Example 12-18. guide/templates/lib/logo

```
[% image = site.logo -%]
<a href="[% site.url.index %]"><img
   src="[% image.src %]" alt="[% image.alt %]"
   width="[% image.width %]" height="[% image.height %]"
   border="0" /></a>
```

Example 12-19. guide/templates/lib/footer

```
<p class="info">
  &copy; Copyright [% site.copyright %].
  All Rights Reserved.
 <br />
  [% page.name %] last modified [% page.date %]
</p>
```

12.1.4.3 Page templates

The *entry.html* page template is used to display a single entry. The template source is shown in Example 12-20. It sets the appropriate page values from the entry returned from the database. This allows the *header* template to display appropriate values when it is automatically added by the wrapper templates. In this simple example, the only real page content comes from the about.entry field.

Example 12-20. guide/templates/cgi/entry.html

```
[% # set various page items
   page.title = entry.name;
   page.name  = "Entry for $entry.name";
   page.date  = date(entry.date);
```

```
  page.about = "by $entry.author on $page.date"
%]

[% entry.about %]
```

Figure 12-3 shows a screenshot of an HTML page generated from this template.

Figure 12-3. earth.png

The *entries.html* page template, shown in Example 12-21, displays a list of the entries returned by a search.

Example 12-21. guide/templates/cgi/entries.html

```
[% page.title = 'Search Results' %]

[% n = entries.size or 'no' %]

<h3>There [% n =  = 1 ? 'is' : 'are' %] [% n %]
[% n =  = 1 ? 'entry' : 'entries' %] matching your search.</h3>

[% IF entries.size %]
  <ul>
  [%- FOREACH entry IN entries -%]
    <li><a href="[% site.url.guide %]?id=[% entry.id %]">[% entry.name %]</a>
      <br />
      <span class="info">by [% entry.author %] on [% date(entry.date) %].</span>
    </li>
  [%- END -%]
  </ul>
[% END %]
```

Figure 12-4 shows the results of a search for *th*.

Figure 12-4. search.png


Search Results
Search
There are 2 entries matching your search.

12.2 CGI Templates

Often dynamic content is generated as a response to a web query. The user types something into a form and a CGI program runs to extract the parameters, search the database, and generate the response. The easiest way to do this is to have the CGI program generate the HTML response. In this section we show a more maintainable way: use the CGI plugin from within a template to access query parameters.

12.2.1 Using the CGI Plugin

The Template Toolkit provides the CGI plugin as a simple wrapper around the CGI module. If you don't have any particular need to use a CGI object in the calling Perl program—say, to read request parameters or set headers—don't create one. Instead, use the CGI plugin to create a CGI object from within any templates that require access to it. Example 12-22 shows a template identical to that in Example 12-7, with the exception of the first line, which loads the CGI plugin.

Example 12-22. cgiplugin.html

```
[% USE cgi %]

<h1>CGI Parameters</h1>

<ul>
[% FOREACH p = cgi.param -%]
  <li><b>[% p %]</b> [% cgi.param(p) %]</li>
[% END -%]
</ul>
```

The plugin name can be specified in upper- or lowercase. The CGI object will be assigned to the variable of the same name in matching case. In Example 12-22, the lowercase `cgi` variable is used in keeping with Example 12-7. We could just as easily use the uppercase `CGI` name when the plugin is loaded:

```
[% USE CGI %]
```

and then again whenever the plugin is used:

```
[% FOREACH p = CGI.param %]
```

You cannot instantiate more than one CGI per request. If you create a CGI request in the controlling Perl script, you should pass it as a variable to the template instead of using a plugin. The plugin will create a second CGI object with unpredictable results.

12.2.2 Web Programming in Templates

The Template Toolkit gives you access to plugins and allows you to call out to subroutines and other objects from template code. This means that you can do a large amount of web programming entirely within the templates.

This is the style of popular web programming languages such as PHP and Microsoft's ASP. It is how HTML::Mason works, albeit by embedding Perl code rather than using a custom web programming language. It is a useful technique, particularly for small applications where you want to keep things together in one place, and don't want the overhead of a complex application framework that will only distract you from the task at hand.

The problem with this approach is that it often doesn't scale well to larger applications. HTML::Mason is perhaps the exception here, being very much based around a component architecture that naturally promotes modularity and scalability. However, it suffers from the same problem as PHP and ASP in binding the application code too tightly to presentation aspects, making it hard to change one without affecting the other.

The Template Toolkit approaches the problem from a different angle. Whereas PHP, ASP, and HTML::Mason are designed primarily for web programming, the Template Toolkit is more focused on web presentation. It deals mostly with making the generated pages look pretty (which may involve all manner of complex presentation logic) but doesn't worry itself too much about application programming issues. That is best left to a real programming language, namely Perl.

But as we have said, the technique is useful for smaller applications, and with a little careful organization can scale reasonably well. The Template Toolkit isn't fanatical about enforcing strict disciplines on anyone, and provides what you need to get the job done quickly, if that's what you want.

12.2.2.1 Dispatching CGI script

To illustrate this, we will take the Perl CGI script from Example 12-11 and implement the body of it in a template, making use of the CGI and DBI plugins. We still require a Perl CGI script to dispatch the template, as shown in Example 12-23.

Example 12-23. guide/cgi-bin/ttguide.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;

$| = 1;

my $ROOTDIR = '/home/dent/guide';
my $ROOTURL = '/~dent/guide';
my $ROOTCGI = '/cgi-bin/dent/ttguide.pl';
my $DBDSN   = 'DBI:mysql:guide';
my $DBUSER  = 'dent';
my $DBPASS  = 'ruhtra';

my $input = 'guide.html';
my $vars  = {
    rootdir => $ROOTDIR,
    rooturl => $ROOTURL,
    rootcgi => $ROOTCGI,
    dbdsn   => $DBDSN,
    dbuser  => $DBUSER,
    dbpass  => $DBPASS,
};

my $tt  = Template->new({
    INCLUDE_PATH => [
        "$ROOTDIR/templates/cgi",
```

```
        "$ROOTDIR/templates/lib",
    ],
    PRE_PROCESS => 'config',
    WRAPPER     => 'wrapper',
});

print "Content-type: text/html\n\n";

$tt->process($input, $vars)
    || die $tt->error( );
```

The script does little more than define some variables and create a Template object to process the *guide.html* file, located in the *templates/cgi* directory, relative to the $ROOTDIR, which in this example is /home/dent/guide.

12.2.2.2 Main control template

The application processing has now been moved into the *guide.html* template, shown in Example 12-24.

Example 12-24. guide/templates/cgi/guide.html

```
[%  USE cgi;
    USE dbi( dbdsn, dbuser, dbpass );

    # main control loop
    IF (param = cgi.param('name'));
        PROCESS entry/name;
    ELSIF (param = cgi.param('id'));
        PROCESS entry/id;
    ELSIF (param = cgi.param('search'));
        PROCESS entry/search;
    ELSE;
        PROCESS index.html;
    END;
%]
```

It first loads the CGI plugin, then the DBI plugin, passing the relevant configuration parameters for it to make a database connection. For both plugins, the lowercase names are used:

```
USE cgi;
USE dbi( dbdsn, dbuser, dbpass );
```

Then the control block follows. The request parameters are inspected and one of the relevant templates, *entry/name*, *entry/id*, or *<entry/search>*, is processed. If none of the parameters is provided, the *index.html* template is used.

```
IF (param = cgi.param('name'));
    PROCESS entry/name;
ELSIF (param = cgi.param('id'));
```

```
    PROCESS entry/id;
ELSIF (param = cgi.param('search'));
    PROCESS entry/search;
ELSE;
    PROCESS index.html;
END;
```

12.2.2.3 Additional control templates

The *entry/name* template, shown in Example 12-25, dispatches a database request to fetch an entry by name.

Example 12-25. guide/templates/cgi/entry/name

```
[%  entries = dbi.query(
        "SELECT id, name, author, about, date
         FROM entry WHERE name='$param'"
    );

    # entries is an iterator, so get first item
    entry = entries.get;

    IF entry;
        PROCESS entry.html;
    ELSE;
        PROCESS notfound.html;
    END;
%]
```

The query method of the DBI plugin returns a reference to an iterator object, which is assigned to entries. We're expecting only one item to be returned from this query, so we call the get method to fetch the first item from entries:

```
entry = entries.get;
```

If an entry is returned, the *entry.html* template is processed to present it. Otherwise, the *notfound.html* template is used to inform the user that the entry could not be found.

The *entry/id* template is very similar (see Example 12-26).

Example 12-26. guide/templates/cgi/entry/id

```
[%  entries = dbi.query(
        "SELECT id, name, author, about, date
         FROM entry WHERE id=$param"
    );
    entry = entries.get;
    IF entry;
        PROCESS entry.html;
```

```
    ELSE;
        PROCESS notfound.html;
    END;
%]
```

Example 12-27 shows the *entry/search* template.

Example 12-27. guide/templates/cgi/entry/search

```
[%  search = param.replace('*', '%');
    entries = dbi.query(
        "SELECT id, name, author, about, date
         FROM entry WHERE name LIKE '$search'"
    );
    PROCESS entries.html
        entries = entries.get_all;
%]
```

As before, we change any occurrences of * to % so that the user's idea of what constitutes a wildcard expression (e.g., ear*) matches the format that MySQL is expecting (e.g., ear%). This time, however, we do it using the replace virtual method:

```
search = param.replace('*', '%');
```

We are expecting a list of items to be returned from the search. The *entries.html* template generates an appropriate response even if no matches are found and the entries list is empty. We call the get_all method on the entries iterator to return a list of all matches found and then assign it back to entries. This effectively turns the iterator into a regular list so that the *entries.html* template can use the size list virtual method to determine whether there are any entries to display.

12.2.2.4 Perl or template?

We don't normally recommend putting too much application logic in templates as a general rule. But we do recognize that it can be useful from time to time, particularly when you have a small job to get done quickly and would rather have something basic working today than something elegant working next week.

In the example that we have just looked at, we created a CGI Perl script specifically to dispatch a single template. Given that we have gone to the effort of writing a Perl script, it would make more sense on this occasion to encode the application logic in Perl, leaving the templates to handle only presentation issues. This is the approach that we showed you in Example 12-7.

On the other hand, you may be using a generic template dispatcher such as Apache::Template. We saw an example in Chapter 11 where it was configured to process any *.tt2* that it finds before being returned to the client. It means you can simply drop a new *.tt2* file into your web directory to have Apache::Template automatically process it as a dynamically generated web page. There is no need to write a calling CGI script or custom mod_perl handler to cater for it. In cases such as this, the benefit of being able to perform some basic web programming tasks entirely within a template is more apparent.

So even though hardcore web programming in templates isn't usually encouraged, it certainly can be done. Furthermore, it is still possible to maintain a clear separation of concerns by using different templates for different parts of the system. In this example, we used one template for the main control loop and one for handling each query. All the presentation templates were borrowed without change from the previous example.

12.3 Apache and mod_perl

The biggest problem with CGI programming is that it is slow. Each request fires off a CGI script from scratch. Perl must first parse and compile the script and any modules you use (including the Template Toolkit, of course) before it can even start to generate content.

The mod_perl extension to Apache makes these problems go away. Rather than writing Perl CGI scripts, you write Perl handlers that sit "inside" the web server. The handlers and any modules they use are loaded and compiled when the server starts. Once Perl has compiled them into an internal "opcode" tree, they can be executed quickly, efficiently, and repeatedly with minimal overhead.

A second important benefit comes from using the Template Toolkit in a mod_perl-enabled Apache server. It allows you to create one Template object that is reused for all requests. When a template is first used, it is parsed by the Template Toolkit and converted to the equivalent Perl code. This is then passed to Perl, which compiles it into an opcode tree.

The Template Toolkit caches these compiled templates so that you can process them as many times as you like but only have to go through the relatively slow process of compiling them once. However, to get the benefit of this, you must use one Template object that remains persistent from one request to the next. The examples that follow all adopt this technique.

For a complete discussion of mod_perl and related topics, see *Practical mod_perl* by Stas Bekman and Eric Cholet (O'Reilly).

12.3.1 Apache::Template

Way back in Chapter 1, we looked at using Apache::Template to dispatch templates from a mod_perl-enabled Apache server. Example 12-28 shows an Apache/mod_perl configuration that uses Apache::Template to dispatch the web application template from Example 12-24.

Example 12-28. Apache::Template configuration

```
PerlModule       Apache::Template

TT2IncludePath     /home/dent/guide/templates/cgi
TT2IncludePath     /home/dent/guide/templates/lib
TT2PreProcess      config
TT2Process         process

TT2Variable        rooturl /~dent/guide
TT2Variable        rootcgi /ttguide
TT2Variable        dbdsn   DBI:mysql:guide
TT2Variable        dbuser  dent
TT2Variable        dbpass  ruhtra

Alias              /ttguide /home/dent/guide/templates/cgi

<Location /ttguide>
    SetHandler      perl-script
    PerlHandler     Apache::Template
</Location>
```

The Apache::Template module is loaded and then various TT2* parameters are set. At the time of this writing, *Apache::Template* is a version behind the Template Toolkit and doesn't yet support the TT2Wrapper (i.e., WRAPPER) configuration option. For now, we can emulate the behavior of TT2Wrapper with the TT2Process option. We tell Apache::Template to process the *process* template, shown in Example 12-29, in place of each main page template.

Example 12-29. templates/lib/process

```
[% PROCESS $template WRAPPER wrapper -%]
```

The *process* template processes the original page template.[2] The template variable contains a reference to the original page template (or rather, the Template::Document object used to represent it). The original template is processed and the output is wrapped in the *wrapper* template, thereby providing the equivalent functionality to the WRAPPER configuration option.

[2] The leading $ on $template indicates that it is the template variable we want processed, rather than a template with the literal name "template."

The rooturl, rootcgi, dbdsn, dbuser, and dbpass template variables are set to their appropriate values using the TT2Variable directive. We also define an Apache Alias that maps the */ttguide* URL to the appropriate template files in the */home/dent/guide/templates/cgi* directory.

```
Alias           /ttguide /home/dent/guide/templates/cgi
```

Finally, we indicate that all files in this location and corresponding directory should be processed by Apache::Template:

```
<Location /ttguide>
    SetHandler      perl-script
    PerlHandler     Apache::Template
</Location>
```

The *guide.html* page template can now be accessed via the URL */ttguide/guide.html*. No changes to the template are required.

12.3.2 Custom Apache Handler

The Apache::Template module is good for simple things. If you want to do anything that doesn't count as simple, you will probably need to write your own custom *mod_perl* handler.

Example 12-30 shows an example of a module that defines such a handler.

Example 12-30. lib/TTBook/Apache/Handler.pm

```
package TTBook::Apache::Handler;

use strict;
use warnings;

use Template;
use Apache;
use Apache::Constants qw(OK SERVER_ERROR DECLINED);

our $VERSION = 1.00;
our $TT;

sub handler {
    my $r = shift;
```

```
    my $output;

    my %params = $r->method( ) eq 'POST'
        ? $r->content( )
        : $r->args( );

    my $template = $r->path_info( )
        || 'index.html';

    $template =~ s[^/][  ]g;

    $TT ||= do {
        my $rootdir = $r->dir_config('rootdir')
            || return error($r, "'rootdir' not defined");

        Template->new({
            INCLUDE_PATH => [
                "$rootdir/templates/cgi",
                "$rootdir/templates/lib",
            ],
            PRE_PROCESS => 'config',
            WRAPPER     => 'wrapper',
            ERROR       => 'error.html',
        });
    };

    $r->content_type('text/html');
    $r->send_http_header( );

    $TT->process($template, \%params, $r)
        || return error($r, $TT->error( ));

    return OK;
}

sub error {
```

```
    my $r = shift;
    $r->log_error(@_);
    return SERVER_ERROR;
}

1;
```

The interesting part is the handler method. It is called by mod_perl and passed a reference to an Apache::Request object. Through this, we can fetch the request parameters by calling the content( ) method for POST requests, or the args( ) method for GET (and other) requests:

```
sub handler {
    my $r = shift;
    my $output;

    my %params = $r->method( ) eq 'POST'
        ? $r->content( )
        : $r->args( );
```

In this handler, we are using PATH_INFO to determine which template to process. If the handler is bound to a URL of */tthandler*, for example, calling it with a URL of */tthandler/help/index.html* would result in a value of /help/index.html for PATH_INFO. In this case, we would then process the *help/index.html* template in the *$rootdir/templates/cgi* directory, having removed the leading / from the path:

```
my $template = $r->path_info( )
    || 'index.html';

$template =~ s[^/][  ]g;
```

The next block of code creates a Template object and assigns it to the $TT package variable. If $TT already contains an object, it is reused instead. This ensures that the same Template object is used from one request to the next and thus benefits from the caching of compiled templates.

```
$TT ||= do {
    my $rootdir = $r->dir_config('rootdir')
        || return error($r, "'rootdir' not defined");

    Template->new({
        INCLUDE_PATH => [
            "$rootdir/templates/cgi",
            "$rootdir/templates/lib",
        ],
        PRE_PROCESS => 'config',
        WRAPPER     => 'wrapper',
        ERROR       => 'error.html',
    });
};
```

The root directory, $rootdir, from which the INCLUDE_PATH directories are built, is defined in the Apache configuration file that we will be looking at shortly. To fetch this value, the dir_config( ) method is called against the request object.

The content type is declared and the HTTP headers are sent to the client's browser:

```
$r->content_type('text/html');

$r->send_http_header( );
```

Then the page template, $template, is processed, passing the current request parameters as template variables. The request object, $r, is passed to the process( ) method as the third argument. Rather than printing the generated HTML page to standard out, the *process( )* method will pass it to the request object by calling its print( ) method:

```
$TT->process($template, \%params, $r)

    || return error($r, $TT->error( ));



return OK;

}
```

Example 12-31 shows the relevant directive for an Apache configuration file to use this handler.

Example 12-31. etc/tthandler.conf

```
<perl>

  use lib qw( /home/dent/guide/lib )

</perl>



PerlModule   TTBook::Apache::Handler

PerlSetVar   rootdir /home/dent/guide



<Location /myhandler>

    SetHandler      perl-script

    PerlHandler     TTBook::Apache::Handler

</Location>
```

The <perl> ... </perl> block allows Perl code to be embedded in the configuration. In this example, we are using it to add the location of our custom handler module to Perl's search path. The module is then loaded with the PerlModule directive. The PerlSetVar directive is used to set a value for the rootdir variable. Finally, a <Location> ... </Location> block is used to bind the handler to the URL */myhandler*.

12.4 A Complete Web Application

We are now going to build a complete mod_perl- and Template Toolkit-enabled, database-driven web application, based on our earlier examples. Although this is a relatively simple example as web applications go, we will nevertheless concentrate on making a clear separation between the different functional concerns.

Presentation will of course be handled by the Template Toolkit. The application-specific processing will be implemented in one module, using another separate module to manage the storage layer (i.e., the database). A third module will then provide the interface between Apache and the application.

12.4.1 Storage

To best understand how the complete application is built, it is perhaps easiest to start from the inside and work out. Or at the bottom and work up. Well, whatever direction it is, we're going to start with the storage module.

This provides a wrapper around a database to hide as much of the nitty-gritty detail as possible. This allows our different applications to use the same storage module, or for an application to use different storage modules as requirements change. In this example, we're using a MySQL database through the DBI module, but next week we might decide to use XML files instead.

In other words, it provides an abstraction that allows applications to work independently of any particular storage mechanism.

12.4.1.1 TTBook::H2G2::Database

This module begins in the usual way for any Perl module by declaring its package and then loading some external Perl modules:

```
package TTBook::H2G2::Database;



use strict;

use DBI;

use Class::Base;

use base qw( Class::Base );
```

The DBI module is of course required to access the MySQL database. We're also using Class::Base and defining it to be the base class of the TTBook::H2G2::Database module.

The three SQL queries that we will be using are defined in the $SQL package variable. They use ? placeholder characters to indicate positions where parameters to the query will be inserted.

```
our $SQL     = {

    get_entry_id    => 'SELECT id, name, author, about, date

                        FROM entry WHERE id=?',

    get_entry_name => 'SELECT id, name, author, about, date

                        FROM entry WHERE name=?',

    entry_search    => 'SELECT id, name, author, about, date

                        FROM entry WHERE name LIKE ?',

};
```

The Class::Base module defines a default new( ) constructor method. This calls the init( ) method to initialize the object using any configuration parameters passed.

```
sub init {

    my ($self, $config) = @_;
```

```
    @$self{ keys %$config } = values %$config;
    $self->{ sql } = $SQL;
    $self->connect( ) || return;
    return $self;
}
```

The contents of the $config hash array are copied into $self and the sql item is set to reference the $SQL package hash. The connect( ) method is then called to make a connection to the database.

Here is the connect( ) method. Notice how the database handle is cached internally in the object as the dbh item.

```
sub connect {
    my $self = shift;

    return $self->{ dbh } ||= do {
        my $dsn = $self->dsn( )
            || return $self->error("No DSN available");

        DBI->connect($dsn, $self->{ user }, $self->{ pass },
                    { RaiseError => 0, PrintError => 0 })
            || $self->error($DBI::errstr);
    };
}
```

The dsn( ) method returns a connection string (in Data Source Notation, hence DSN) for the connect( ) method. If a dsn is already defined, either by a configuration option or a previous call to dsn( ), it is returned as is. Otherwise, it is generated using some or all of the values for name, host, port, and driver, which should be provided as configuration options to the new( ) constructor.

```
sub dsn {
    my $self = shift;
    return $self->{ dsn } ||= do {
        my ($name, $host, $port) = @$self{ qw( name host port ) };
        $host .= ":$port" if $host && $port;
        $name .= "@$host" if $host;
        join(':', 'DBI', $self->{ driver }, $name);
    };
}
```

The prepare( ) method is used to fetch a named SQL query from the sql hash (e.g., get_entry_name, get_entry_id, etc.) and prepare it for execution. The prepared query is cached in the internal sql_query hash table for subsequent use.

```
sub prepare {
    my $self = shift;
    my $sql  = shift
        || return $self->error("no SQL");
    my $dbh  = $self->{ dbh }
        || return $self->error("DBI not connected");
```

```
    my $query;

    if ($query = $self->{ sql }->{ $sql }) {
        my $cache = $self->{ sql_query } ||= { };

        return $cache->{ $sql } ||= $dbh->prepare($query)
            || $self->error("DBI prepare failed: $DBI::errstr");
    }
    else {
        return $dbh->prepare($sql)
            || $self->error("DBI prepare failed: $DBI::errstr");
    }
}
```

The query( ) method calls prepare( ) to prepare a query, and then executes it:

```
sub query {
    my $self = shift;
    my $sql  = shift
        || return $self->error("no SQL");
    my $dbh  = $self->{ dbh }
        || return $self->error("DBI not connected");
    my $sth = $self->prepare($sql)
        || return;
    $sth->execute(@_)
        || return $self->error($sth->errstr( ));
    return $sth;
}
```

The item( ) method first calls query( ) to execute a query. It then calls fetchrow_hashref( ) on the returned DBI statement handle to fetch the first (or only) record returned.

```
sub item {
    my $self = shift;
    my $sth  = $self->query(@_) || return;
    return $sth->fetchrow_hashref( )
        || $self->error($DBI::errstr || "not found");
}
```

The list( ) method is similar, but calls fetchall_arrayref( ) to return a list of all records returned by the query:

```
sub list {

    my $self = shift;

    my $sth  = $self->query(@_) || return;

    return $sth->fetchall_arrayref({  })

        || $self->error($DBI::errstr || "not found");

}
```

The one other method that is worth mentioning is DESTROY. This calls the disconnect( ) method to ensure that the database connection is closed when the object is destroyed.

```
sub DESTROY {

    my $self = shift;

    $self->disconnect('object destroyed') if $self->{ dbh };

}
```

We haven't shown you disconnect( ) yet, but you can probably guess what it does. It is included in the complete listing of the TTBook::H2G2::Database module that follows in Example 12-32.

Example 12-32. lib/TTBook/H2G2/Database.pm

```
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

 = = == = = = = = = = = = = = = = = = = = = = = = = = = = = = =

  = = =

#

# TTBook::H2G2::Database

#

# DESCRIPTION

#   Backend database module for the H2G2 web application.

#

# AUTHOR

#   Andy Wardley <abw@wardley.org>

#

# COPYRIGHT

#   Copyright (C) 2003 Andy Wardley.  All Rights Reserved.

#

#   This module is free software; you can redistribute it and/or

#   modify it under the same terms as Perl itself.

#

# REVISION

#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

=

```
package TTBook::H2G2::Database;

use strict;
use DBI;
use Class::Base;
use base qw( Class::Base );

our $VERSION = sprintf("%d.%02d", q$Revision: 1.6 $ =~ /(\d+)\.(\d+)/);
our $ERROR   = '';
our $SQL     = {
    get_entry_id   => 'SELECT id, name, author, about, date
                       FROM entry WHERE id=?',
    get_entry_name => 'SELECT id, name, author, about, date
                       FROM entry WHERE name=?',
    entry_search   => 'SELECT id, name, author, about, date
                       FROM entry WHERE name LIKE ?',
};

#------------------------------------------------------------------------
# init(\%config)
#
# Initialization method called by Class::Base new( ) constructor.
#------------------------------------------------------------------------

sub init {
    my ($self, $config) = @_;
    @$self{ keys %$config } = values %$config;
    $self->{ sql } = $SQL;
    $self->connect( ) || return;
    return $self;
}

#------------------------------------------------------------------------
# dsn( )
#
# Generate a DSN string from the database
```

```
# connection parameters.
#------------------------------------------------------------------------

sub dsn {
    my $self = shift;
    return $self->{ dsn } ||= do {
        my ($name, $host, $port) = @$self{ qw( name host port ) };
        $host .= ":$port" if $host && $port;
        $name .= "@$host" if $host;
        join(':', 'DBI', $self->{ driver }, $name);
    };
}

#------------------------------------------------------------------------
# connect( )
#
# Connect to the backend database.
#------------------------------------------------------------------------

sub connect {
    my $self = shift;

    return $self->{ dbh } ||= do {
        my $dsn = $self->dsn( )
            || return $self->error("No DSN available");
        DBI->connect($dsn, $self->{ user }, $self->{ pass },
                     { RaiseError => 0, PrintError => 0 })
            || $self->error($DBI::errstr);
    };
}

#------------------------------------------------------------------------
# disconnect( )
#
# Disconnect the database.
#------------------------------------------------------------------------

sub disconnect {
```

```
    my $self = shift;
    my $msg = shift || '';
    $msg = " ($msg)" if length $msg;

    delete $self->{ sql_query };

    $self->{ dbh }->disconnect( )
        if $self->{ dbh };
    delete $self->{ dbh };

    return 1;
}

#------------------------------------------------------------------------
# prepare($sql)
#
# Prepare a query and store the live statement handle internally for
# subsequent execute( ) calls.
#------------------------------------------------------------------------

sub prepare {
    my $self = shift;
    my $sql  = shift
        || return $self->error("no SQL");
    my $dbh  = $self->{ dbh }
        || return $self->error("DBI not connected");
    my $query;

    if ($query = $self->{ sql }->{ $sql }) {
        my $cache = $self->{ sql_query } ||= { };

        return $cache->{ $sql } ||= $dbh->prepare($query)
            || $self->error("DBI prepare failed: $DBI::errstr");
    }
    else {
        return $dbh->prepare($sql)
            || $self->error("DBI prepare failed: $DBI::errstr");
```

```
    }
}

#-----------------------------------------------------------------------
# query($sql, @params)
#
# Prepares and executes an SQL query.
#-----------------------------------------------------------------------

sub query {
    my $self = shift;
    my $sql  = shift
        || return $self->error("no SQL");
    my $dbh  = $self->{ dbh }
        || return $self->error("DBI not connected");
    my $sth = $self->prepare($sql)
        || return;
    $sth->execute(@_)
        || return $self->error($sth->errstr( ));
    return $sth;
}

#-----------------------------------------------------------------------
# item($sql, @args)
#
# Executes the $sql query, passing @args and calls fetchrow_hashref( ) on
# the returned statement handle to fetch a single row as a hash.
#-----------------------------------------------------------------------

sub item {
    my $self = shift;
    my $sth  = $self->query(@_) || return;
    return $sth->fetchrow_hashref( )
        || $self->error($DBI::errstr || "not found");
```

```
}

#------------------------------------------------------------------------
# list($sql, @args)
#
# Executes the $sql query, passing @args and calls fetchall_arrayref( ) on
# the returned statement handle to fetch all rows as a list of hashes.
#------------------------------------------------------------------------

sub list {
    my $self = shift;
    my $sth  = $self->query(@_) || return;
    return $sth->fetchall_arrayref({  })
        || $self->error($DBI::errstr || "not found");
}

#------------------------------------------------------------------------
# insert_id( )
#
# Returns the identity of the record most recently inserted into the
# database.
#------------------------------------------------------------------------

sub insert_id {
    my $self = shift;
    return $self->{ dbh }->{ mysql_insertid };
}

#------------------------------------------------------------------------
# quote($value [, $data_type ])
#
# Returns a quoted string (correct for the connected database) from the
# value passed in.
#------------------------------------------------------------------------

sub quote {
    my $self = shift;
    my $dbh  = $self->{ dbh } || return $self->error("DBI not connected");
```

```
    return $dbh->quote(@_);
}

#------------------------------------------------------------------------
# dbh( )
#
# Internal method that retrieves the database handle belonging to the
# instance or attempts to create a new one using connect.
#------------------------------------------------------------------------

sub dbh {
    my $self = shift;
    return $self->{ dbh } || $self->connect( );
}

#------------------------------------------------------------------------
# DESTROY( )
#
# Destructor method called automatically when the object goes out of
# scope.   Disconnects any active database.
#------------------------------------------------------------------------

sub DESTROY {
    my $self = shift;
    $self->disconnect('object destroyed') if $self->{ dbh };
}

1;
```

12.4.2 Configuration

The database storage module expects to be provided with various configuration options to define the parameters for connecting to the database. Rather than littering this information around in several different places (something that makes it hard to find and change), we will create a single configuration module, as shown in Example 12-33.

Example 12-33. lib/TTBook/H2G2/Config.pm

```
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = =
```

```
#
# TTBook::H2G2::Config
#
# DESCRIPTION
#   Configuration module for the Hitch-Hiker's Guide to the Galaxy web
#   application.
#
# AUTHOR
#   Andy Wardley <abw@wardley.org>
#
# COPYRIGHT
#   Copyright (C) 2003 Andy Wardley.  All Rights Reserved.
#
#   This module is free software; you can redistribute it and/or
#   modify it under the same terms as Perl itself.
#
# REVISION
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = =

package TTBook::H2G2::Config;

use strict;
use warnings;

our $VERSION  = 1.00;
our $ROOTDIR  = '/home/dent/web/guide';
our $ROOTURL  = '/H2G2';
our $ROOTCGI  = '/H2G2/guide';
our $DATABASE = {
    driver  => 'mysql',
    name    => 'guide',
    user    => 'dent',
    pass    => 'ruhtra',
    host    => '',
    port    => '',
};
our $TEMPLATE = {
```

```
    INCLUDE_PATH => [
      "$ROOTDIR/templates/cgi",
      "$ROOTDIR/templates/lib",
    ],
    PRE_PROCESS  => 'config',
    WRAPPER      => 'wrapper',
    VARIABLES    => {
        rooturl  => $ROOTURL,
        rootcgi  => $ROOTCGI,
    }
};
our $TEMPLATES = {
    index   => 'index.html',
    entry   => 'entry.html',
    entries => 'entries.html',
    error   => 'error.html',
};

1;
```

It defines $ROOTDIR, $ROOTURL, and $ROOTCGI to indicate the root directory, the root URL for documents, and the URL to access the application handler, respectively. The $DATABASE hash array defines the connection parameters for the TTBook::H2G2::Database module. The $TEMPLATE hash provides the familiar set of options for the Template module. Finally, the $TEMPLATES hash (note the plural) maps application actions (e.g., fetch entry, fetch list of entries, etc.) to presentation templates for displaying the outcome of the operation.

12.4.3 Application

Now that we have a storage module and the means to configure it, we can start to build our main application module:

```
package TTBook::H2G2;


use strict;
use Template;
use TTBook::H2G2::Config;
use TTBook::H2G2::Database;
use Class::Base;
use base qw( Class::Base );
```

The TTBook::H2G2 module is also a subclass of Class::Base and uses the configuration and database modules that we have already defined. We will be making several references to the $ROOTURL and $TEMPLATES items in the TTBook::H2G2::Config module, so we create local package variables to alias them, to save us from typing them repeatedly, if nothing else:

```
our $ROOTURL   = $TTBook::H2G2::Config::ROOTURL;
our $TEMPLATES = $TTBook::H2G2::Config::TEMPLATES;
```

The init( ) method, called by the new( ) constructor method in Class::Base, looks for three different configuration options. The first, database, can be used to provide a reference to a storage object other than the default. The second, template, allows the default template processing engine to be replaced. We'll not be using either of these in this example, but they illustrate how easy it is to use different modules to handle storage or presentation issues. The third option, templates, allows a different set of template mapping to be provided. These are merged with the default set, $TEMPLATES.

```
sub init {
    my ($self, $config) = @_;

    # user can provide custom database object
    $self->{ database } = $config->{ database };

    # same for template object
    $self->{ template } = $config->{ template };

    # merge user-supplied templates with defaults
    my $templates = $config->{ templates } || { };
    $self->{ templates } = {
        map { defined $templates->{ $_ }
                    ? ($_ => $templates->{ $_ })
                    : ($_ => $TEMPLATES->{ $_ })
        } keys %$TEMPLATES
    };

    return $self;
}
```

The database( ) method creates a TTBook::H2G2::Database object using the $DATABASE connection parameters defined in TTBook::H2G2::Config and caches it internally as the database item. If an object is already defined for database, either by being passed to new( ) as a configuration option or by being created by a previous call to the database( ) method, it is instead returned.

```
sub database {
    my $self = shift;

    return $self->{ database } ||= do {
        my $params = @_ && UNIVERSAL::isa($_[0], 'HASH') ? shift : { @_ };
        my $config = $TTBook::H2G2::Config::DATABASE;
        $config = {
            %$config,
            %$params,
        };
        TTBook::H2G2::Database->new($config)
            || $self->error(TTBook::H2G2::Database->error( ));
    };
}
```

The template( ) method is a factory method similar to database( ). In this case, it creates a Template object for processing templates for the application.

```
sub template {
    my $self = shift;

    return $self->{ template } ||= do {
        my $params = @_ && UNIVERSAL::isa($_[0], 'HASH') ? shift : { @_ };
        my $config = $TTBook::H2G2::Config::TEMPLATE;
        $config = {
            %$config,
            %$params,
        };
        Template->new($config)
            || return $self->error(Template->error( ));
    };
}
```

Now we can define some application-processing methods. The first is entry( ). It expects either a name or id parameter and then makes a call to the database item method to fetch the entry in question.

```
sub entry {
    my $self = shift;
    my $args = @_ && ref $_[0] eq 'HASH' ? shift : { @_ };
    my $database = $self->database( ) || return;
    my $entry;

    if (defined $args->{ id }) {
        return $database->item( get_entry_id => $args->{ id } )
            || $self->error($database->error( ));
    }
    elsif (defined $args->{ name }) {
        return $database->item( get_entry_name => $args->{ name } )
            || $self->error($database->error( ));
    }
    else {
        return $self->error("entry( ) expects 'name' or 'id' parameter");
    }
}
```

The search( ) method expects a search term as an argument. It calls the database list method to fetch a list of items returned by the entry_search query, forwarding the search term (modified as before) as an argument.

```
sub search {
    my ($self, $search) = @_;
```

```
    my $database = $self->database( ) || return;

    # change '*' to '%'
    $search =~ s/*/\%/g;

    return $database->list( entry_search => $search )
        || $self->error($database->error( ));
}
```

The run( ) method ties it all together. It is passed a reference to a hash array of request parameters. It inspects the parameters and dispatches the appropriate method to handle it: entry( ) or search( ). The entry or entries returned are added to the $params hash as template variables. The $template variable is also set to indicate the correct page template for the action.

```
sub run {
    my ($self, $params) = @_;
    my $templates = $self->{ templates };
    my ($tt, $template, $output);

    if (defined $params->{ name } || defined $params->{ id }) {
        # fetch entry if 'name' or 'id' specified
        my $entry = $self->entry($params);
        if ($entry) {
            $params->{ entry } = $entry;
            $template = $templates->{ entry };
        }
        else {
            $params->{ error } = $self->error( );
            $template = $templates->{ error };
        }
    }
    elsif (defined $params->{ search }) {
        # search for entries if 'search' specified
        my $entries = $self->search($params->{ search });
        if ($entries) {
            $params->{ entries } = $entries;
            $template = $templates->{ entries };
        }
        else {
            $params->{ error } = $self->error( );
            $template = $templates->{ error };
        }
```

```
    }
    else {
        return [ redirect => "$ROOTURL/index.html" ];
    }
```

If none of the parameters is set, a reference to a list is returned, indicating that the application should redirect to the *index.html* page relative to the $ROOTURL. We will be looking at the meaning of these return values shortly.

The final section of the run( ) method uses the Template object returned by the template( ) method ($tt) to process the page template named in the $template variable. The $params hash defines the template variables and the output is saved to the $output variable.

```
$tt = $self->template( )
    || return [ error => $self->error( ) ];

$tt->process($template, $params, \$output)
    || return [ error => $tt->error( ) ];
```

Whatever happens the method returns a reference to a list. The first item in the list is a string indicating the required action to be undertaken. A value of redirect should trigger a redirect to the URL specified as the second item in the list. A value of error denotes an error, with the second item in the list being an appropriate error message.

A value of output indicates that the page was successfully processed and that it has generated output that should be sent back to the client's browser. In this case, the second item in the list is a *reference* to the variable containing the output.

```
return [ output => \$output ];
```

The complete TTBook::H2G2 module is shown in Example 12-34.

Example 12-34. lib/TTBook/H2G2.pm

```
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = =
#
# TTBook::H2G2
#
# DESCRIPTION
#   A web application for a guide such as the Hitch Hiker's Guide to the
#   Galaxy.
#
# AUTHOR
#   Andy Wardley <abw@wardley.org>
#
# COPYRIGHT
#   Copyright (C) 2003 Andy Wardley.  All Rights Reserved.
#
#   This module is free software; you can redistribute it and/or
#   modify it under the same terms as Perl itself.
```

```
#
# REVISION
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = =

package TTBook::H2G2;

use strict;
use Template;
use TTBook::H2G2::Config;
use TTBook::H2G2::Database;
use Class::Base;
use base qw( Class::Base );

our $VERSION = sprintf("%d.%02d", q$Revision: 1.6 $ =~ /(\d+)\.(\d+)/);
our $DEBUG   = 0 unless defined $DEBUG;
our $ERROR   = '';
our $ROOTURL   = $TTBook::H2G2::Config::ROOTURL;
our $TEMPLATES = $TTBook::H2G2::Config::TEMPLATES;

#------------------------------------------------------------------------
# init(\%config)
#
# Initializer method called by Class::Base new( ) method.
#------------------------------------------------------------------------

sub init {
    my ($self, $config) = @_;

    # user can provide custom database object
    $self->{ database } = $config->{ database };

    # same for template object
    $self->{ template } = $config->{ template };

    # merge user-supplied templates with defaults
    my $templates = $config->{ templates } || { };
    $self->{ templates } = {
```

```
        map { defined $templates->{ $_ }
                  ? ($_ => $templates->{ $_ })
                  : ($_ => $TEMPLATES->{ $_ })
        } keys %$TEMPLATES
    };

    return $self;
}

#------------------------------------------------------------------------
# database( )
#
# Create or reuse existing database object.
#------------------------------------------------------------------------

sub database {
    my $self = shift;

    return $self->{ database } ||= do {
        my $params = @_ && UNIVERSAL::isa($_[0], 'HASH') ? shift : { @_ };
        my $config = $TTBook::H2G2::Config::DATABASE;
        $config = {
            %$config,
            %$params,
        };
        TTBook::H2G2::Database->new($config)
            || $self->error(TTBook::H2G2::Database->error( ));
    };
}

#------------------------------------------------------------------------
# template( )
#
# Create or reuse existing template processing object.
#------------------------------------------------------------------------

sub template {
    my $self = shift;
```

```
    return $self->{ template } ||= do {
        my $params = @_ && UNIVERSAL::isa($_[0], 'HASH') ? shift : { @_ };
        my $config = $TTBook::H2G2::Config::TEMPLATE;
        $config = {
            %$config,
            %$params,
        };
        Template->new($config)
            || return $self->error(Template->error( ));
    };
}

#------------------------------------------------------------------------
# entry( id => 12345 )
# entry( name => 'Earth' )
#
# Fetch an entry from the database.
#------------------------------------------------------------------------

sub entry {
    my $self = shift;
    my $args = @_ && ref $_[0] eq 'HASH' ? shift : { @_ };
    my $database = $self->database( ) || return;
    my $entry;

    if (defined $args->{ id }) {
        return $database->item( get_entry_id => $args->{ id } )
            || $self->error($database->error( ));
    }
    elsif (defined $args->{ name }) {
        return $database->item( get_entry_name => $args->{ name } )
            || $self->error($database->error( ));
    }
    else {
        return $self->error("entry( ) expects 'name' or 'id' parameter");
```

```
    }
}

#------------------------------------------------------------------------
# search($term)
#
# Search for items in the database based on a search term.
#------------------------------------------------------------------------

sub search {
    my ($self, $search) = @_;
    my $database = $self->database( ) || return;

    # change '*' to '%'
    $search =~ s/*/\%/g;

    return $database->list( entry_search => $search )
        || $self->error($database->error( ));
}

#------------------------------------------------------------------------
# run(\%params)
#
# Run web application.
#------------------------------------------------------------------------

sub run {
    my ($self, $params) = @_;
    my $templates = $self->{ templates };
    my ($tt, $template, $output);

    if (defined $params->{ name } || defined $params->{ id }) {
        # fetch entry if 'name' or 'id' specified
        my $entry = $self->entry($params);
        if ($entry) {
            $params->{ entry } = $entry;
            $template = $templates->{ entry };
        }
        else {
```

```
            $params->{ error } = $self->error( );
            $template = $templates->{ error };
        }
    }
    elsif (defined $params->{ search }) {
        # search for entries if 'search' specified
        my $entries = $self->search($params->{ search });
        if ($entries) {
            $params->{ entries } = $entries;
            $template = $templates->{ entries };
        }
        else {
            $params->{ error } = $self->error( );
            $template = $templates->{ error };
        }
    }
    else {
        return [ redirect => "$ROOTURL/index.html" ];
    }

    # process template and return output or error
    $tt = $self->template( )
        || return [ error => $self->error( ) ];

    $tt->process($template, $params, \$output)
        || return [ error => $tt->error( ) ];

    return [ output => \$output ];
}

1;
```

12.4.4 Apache mod_perl Interface Module

Finally we can add a module to provide the Apache-specific interface to the web application. This is shown in Example 12-35.

Example 12-35. lib/TTBook/H2G2/Apache.pm

```
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

```
 = =
#
# TTBook::H2G2::Apache
#
# DESCRIPTION
#   Apache/mod_perl handler for the H2G2 web application.
#
# AUTHOR
#   Andy Wardley <abw@wardley.org>
#
# COPYRIGHT
#   Copyright (C) 2003 Andy Wardley.  All Rights Reserved.
#
#   This module is free software; you can redistribute it and/or
#   modify it under the same terms as Perl itself.
#
# REVISION
#= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
 = =

package TTBook::H2G2::Apache;

use strict;
use Apache;
use Apache::Constants qw(OK SERVER_ERROR);
use TTBook::H2G2;

our $VERSION = 1.00;
our $H2G2APP;

sub handler {
    my $r = shift;
    my %params   = $r->method( ) eq 'POST'
                ? $r->content( ) : $r->args( );

    # create or reuse existing application object
    $H2G2APP ||= TTBook::H2G2->new( )
        ||  return error($r, "Can't create webapp instance: ",
```

```
                    TTBook::H2G2->error( ));

    # run the application
    my $result = $H2G2APP->run(\%params)
        || return error($r, "Can't run webapp",
                        $H2G2APP->error( ));

    # handle the result
    my $action = shift @$result;

    if ($action eq 'output') {
        my $content = shift @$result;
        $r->content_type('text/html');
        $r->headers_out->add('Content-Length', length($$content));
        $r->send_http_header( );
        $r->print($$content);
        return OK;
    }
    elsif ($action eq 'redirect') {
        my $url = shift @$result;
        $r->internal_redirect($url);
    }
    elsif ($action eq 'error') {
        return error($r, @$result);
    }
    else {
        return error($r, "cannot handle action: $action");
    }
}

sub error {
    my $r = shift;
    $r->log_error(@_);
    return SERVER_ERROR;
}

1;
```

The $H2G2APP package variable is used to store a persistent reference to a TTBook::H2G2 application object. Inside the handler( ) method, we call the application run( ) method, passing the current set of request parameters as arguments. The result returned in stored in the $result variables.

```
my $result = $H2G2APP->run(\%params)
    || return error($r, "Can't run webapp",
                    $H2G2APP->error( ));
```

Then all that is left to do is to examine the first item in the $result list reference and perform the appropriate action: return content to the client, perform a redirect, or log an error.

```
my $action = shift @$result;


if ($action eq 'output') {
    my $content = shift @$result;
    $r->content_type('text/html');
    $r->headers_out->add('Content-Length', length($$content));
    $r->send_http_header( );
    $r->print($$content);
    return OK;
}
elsif ($action eq 'redirect') {
    my $url = shift @$result;
    $r->internal_redirect($url);
}
elsif ($action eq 'error') {
    return error($r, @$result);
}
else {
    return error($r, "cannot handle action: $action");
}
```

12.4.5 Apache Configuration

All that remains to deploy our web application under mod_perl is to write an Apache configuration file and restart the web server. Example 12-36 shows a typical configuration that should be copied into the main *httpd.conf* file or loaded through an Include directive.

Example 12-36. etc/ttguide.conf

```
Alias /H2G2/images/   /home/dent/guide/images/
Alias /H2G2/          /home/dent/guide/html/


<perl>
  use lib qw( /home/dent/guide/lib )
```

```
</perl>


PerlModule   TTBook::H2G2::Apache


<Location /H2G2/guide>
    SetHandler       perl-script
    PerlHandler     TTBook::H2G2::Apache
</Location>
```

Chapter 12. Dynamic Web Content and Web Applications

In Chapter 2 and Chapter 11, we looked at some basic, and then some more advanced techniques for generating static web content. The fundamental limitation of static web pages is, rather obviously, that they are static. The Template Toolkit allows you to incorporate any kind of dynamic data into a template as it is being processed. But once the page has been generated, the data is fixed. If you want to use different data, you must process the template again.

Most web content is *static*. The page is generated offline from a template, using a page design tool, or perhaps just typed in at a text editor. It is then uploaded to the web server where it is delivered time and time again without changing. Simple, fast, and efficient.

Some web content is *dynamic*. The results from a search engine are a perfect example of a dynamically generated page. There's no way of generating the page in advance because you don't know what search terms the user is going to enter. There are many other examples of dynamically generated web content to be found at news sites, in bulletin boards and chat rooms, and of course in e-commerce applications, where pages showing the latest offers or the contents of a user's shopping cart must be generated dynamically to incorporate the latest live data.

In this chapter, we will look at generating dynamic web pages using the Template Toolkit. We will start with some simple CGI scripts to illustrate the basic principles, and then move up to Apache and mod_perl. We'll be working toward a complete (but minimal) web application, concentrating particularly on achieving a clear separation of concerns between different functional aspects of the system: presentation, application, and storage.

2.1 Getting Started

Every big web site is made up of individual pages. Let's start with a small and simple page, showing how to eliminate basic repetition using templates. In later sections, we can build on this to generate more pages and add more complex elements.

2.1.1 A Single Page

Example 2-1 shows the HTML markup of a page that displays the customary "Hello World" message, complete with a title, footer, and various other bits of HTML paraphernalia.

Example 2-1. hello.html

```
<html>
 <head>
  <title>Arthur Dent: Greet the Planet</title>
 </head>

 <body bgcolor="#FF6600">
  <h1>Greet the Planet</h1>

  <p>
   Hello World!
  </p>

  <hr />

  <div align="middle">
   &copy; Copyright 2003 Arthur Dent
  </div>
 </body>
</html>
```

HTML is relatively straightforward in terms of syntax and semantics. We'll assume that you've got at least a passing aquaintance with the basics of HTML. If you don't, *HTML & XML* by Chuck Musciano and Bill Kennedy (O'Reilly) provides a definitive guide to the subject.

Although HTML is simple, it does tend to be rather verbose. It's all too easy for the core content of the page to be obscured by the extra markup required around it. There's also some repetition that we would like to avoid. The page title and author's name both appear twice in the same page, for example. We can also assume that other pages in the site will be using similar pieces of data, repeated over and over again in numerous different places.

The author's name, background color, and copyright message are a few examples of items that we would really rather define in just one place in case we ever decide to change them. We don't want to have to edit every page in the site when we need to change the copyright message (at the start of a new year, for example), or decide that blue is the new orange and want to use it as the background color for every page.

2.1.2 A "Hello World" HTML Template

We can address these issues by applying the basic principles of template processing. Rather than creating the HTML page directly, we write a template for generating the HTML page. In this document, we use template variables to store these values instead of hardcoding them.

Example 2-2 shows a source template for the HTML page in Example 2-1. The author's name, page title, background color, and year have been replaced by the variables author, title, bgcol, and year, respectively.

Example 2-2. hello.tt

```
<html>
  <head>
    <title>[% author %]: [% title %]</title>
  </head>

  <body bgcolor="[% bgcol %]">
    <h1>[% title %]</h1>

    <p>
      Hello World!
    </p>

    <hr />

    <div align="middle">
      &copy; Copyright [% year %] [% author %]
    </div>
  </body>
</html>
```

2.1.3 Processing Templates with tpage

Of course, a template isn't something a browser can make sense of. We need to process the template to generate HTML to send to the browser. Let's use the *tpage* command we met in Chapter 1:

```
$ tpage --define author="Arthur Dent" \
>       --define title="Greet the Planet" \
>       --define bgcol="#FF6600" \
>       --define year=2003 \
>       hello.tt > hello.html
```

The *hello.html* now contains the same HTML that we saw in Example 2-1. This time, however, it has been generated from a template. The benefit of this approach is that we easily change any of these variable values and generate a new HTML page, simply by invoking *tpage* with a different set of parameters.

2.10 Assessment

This brings us nicely back to where we started, looking at the basic principle of template processing: separating your *data* from the way it is *presented*. It's not always clear where your data belongs: in a configuration template; defined in a Perl script; or perhaps stored in a SQL database or XML file. Sometimes you'll want to begin by defining some simple variables in a configuration template so that you can start designing the layout and look and feel of the site. Later on, you might choose to define that data somewhere else, passing it in from a Perl script or making it available through a plugin.

The beauty of the Template Toolkit is that it really doesn't matter. It abstracts the details of the underlying implementation behind the uniform dotted notation for accessing data so that your templates keep working when your storage requirements change, as they inevitably will for many web sites.

It also makes it easy to include things such as loops, conditional statements, and other templates as easy as possible so that you can concentrate on presentation, rather than getting bogged down in the more precise details of full-blown programming language syntax. This is what we mean when we describe the Template Toolkit as a *presentation language* rather than a *programming language*.

It is an example of a *domain-specific language* that in many ways is similar to SQL, which is a domain-specific language for formulating database queries. As such, it should generally be used for what it is good at, rather than being contorted into doing something that might be a lot easier in another language. That doesn't mean that you can't use the Template Toolkit to do CGI programming, embed Perl, or even write Vogon poetry, if that's your thing, but that's not necessarily where its particular strengths lie.[3]

[3] Although the jury is still grooping hooptiously at the implorations of generating Vogon Poetry using the Template Toolkit.

And that's where Perl comes in. The Template Toolkit is designed to integrate with Perl code as cleanly and as easily as possible. When you want to do something more than the Template Toolkit provides, it is easy to append your own additions using a real programming language such as Perl. The plugin mechanism makes it easy to load external Perl code into templates so that you're not always writing Perl wrapper scripts just to add something of your own.

However, this total separation is not something that the Template Toolkit enforces, although the default settings for various configuration options such as EVAL_PERL do tend to encourage it. Sometimes you just want to define a simple Perl subroutine in a template, for example, and don't want to bother with a separate Perl script or plugin module. The Template Toolkit gives you the freedom to do things such as this when you really want to.

For example, by enabling the EVAL_PERL option (see Chapter 4 and the Appendix for details), we can quickly define a Perl subroutine and bind it to a template variable, using a PERL block such as the following:

```
[% PERL %]
$stash->set( help => sub {
    my $entry = shift;
    return "$entry: mostly harmless";
} );
[% END %]
```

The `$stash->set( var => $value )` code, shown here binding the `help` variable to the Perl subroutine, is the Perl equivalent of writing `[% var = value %]` in a template—except, of course, that you can't usually define a subroutine directly in a template, only by using Perl code with EVAL_PERL set (which we think is a sensible restriction). This block can easily be defined in a preprocessed configuration template to keep it out of harm's way, leaving the template authors to use the simple variable:

```
[% help('Earth') %]
```

The important thing is to achieve an *appropriate* separation of concerns, rather than a *total* separation of concerns. Sometimes it's easier to define everything in one template or Perl program and to use a clear layout to separate the different parts. Splitting a small and self-contained document into several different pieces, each comprising just one part of the jigsaw puzzle, can make it hard to see the big picture. On the other hand, a more complex web site may have bigger pieces that absolutely need to be maintained in isolation from the other parts. Remember, there is no golden rule, so the Template Toolkit doesn't try and enforce one on you.

The techniques that we've taught you in this chapter will allow you to address most, if not all, of the simple but common problems that you'll typically face when building and maintaining a web site. We'll be coming back to the Web in Chapter 11 to look at some further ways in which the Template Toolkit can be used to enhance your site and make your life easier. In Chapter 12, we'll be showing how it can be used to handle the presentation layer to simplify the process of building and customizing web applications.

2.2 Template Components

Example 2-2 shows a template for generating a complete HTML page. We refer to this kind of template as a *page template* to distinguish it from the other kind of template that we're now going to introduce: the *template component*.

We use the term "template component" to help us identify those smaller templates that contain a reusable chunk of text, markup, or other content, but don't constitute complete pages in their own right. Template components are no different from page templates as far as the Template Toolkit is concerned—they're all just text files with embedded directives that need processing and get treated equally. Examples of typical template components include headers, footers, menus, and other user interface elements that you will typically want to use and reuse in different page templates across the site.

When we start using *ttree* a little later in this chapter, we will need to be more careful about storing our page templates separately from any template components. For now, however, we can keep them all in the same directory, simplifying matters for the purpose of our examples. As a general naming convention, we use a .tt or .html file extension for page templates (e.g., *hello.tt*), and no extension for component templates (e.g., *header*), but this is entirely arbitrary. If you want to give them an extension (e.g., *header.ttc*), that's fine.

2.2.1 Headers and Footers

Our first components can be created easily. Extract the header and footer blocks from Example 2-2 and save them in their own *header* and *footer* template files, as in Examples Example 2-3 and Example 2-4.

Example 2-3. header

```
<html>

  <head>

    <title>[% author %]: [% title %]</title>

  </head>


  <body bgcolor="[% bgcol %]">

    <h1>[% title %]</h1>
```

Example 2-4. footer

```
    <hr />


    <div align="middle">

      &copy; Copyright [% year %] [% author %]

    </div>

  </body>

</html>
```

2.2.1.1 The PROCESS directive

We can now load these template components into a page template using the PROCESS directive. Example 2-5 shows this in action.

Example 2-5. goodbye.tt

```
[% PROCESS header %]


  <p>

    Goodbye World.

  </p>


[% PROCESS footer %]
```

When the Template Toolkit encounters a PROCESS directive, it loads the template from the file named immediately after the PROCESS keyword (*header* and *footer* are the two templates in this example), processes it to resolve any embedded directives, and then inserts the generated output into the calling template in place of the original directive.

We can use *tpage* to process the *goodbye.tt* template and save the generated output to *goodbye.html*:

```
$ tpage --define author="Arthur Dent" \

>       --define title="We'll Meet Again" \

>       --define bgcol="#FF6600" \

>       --define year=2003 \

>       goodbye.tt > goodbye.html
```

The output generated, shown in Example 2-6, shows how the header and footer have been processed into place and the variable references within them correctly resolved.

Example 2-6. goodbye.html

```
<html>

  <head>

    <title>Arthur Dent: We'll Meet Again</title>

  </head>


  <body bgcolor="#FF6600">

    <h1>We'll Meet Again</h1>




  <p>

    Goodbye World.

  </p>


    <hr />


    <div align="middle">

      &copy; Copyright 2003 Arthur Dent

    </div>

  </body>

</html>
```

2.2.1.2 The INSERT directive

The Template Toolkit provides a number of different directives for loading external template components. The INSERT directive, for example, inserts the contents of a template, but *without* processing any directives that may be embedded in it:

```
[% INSERT footer %]
```

INSERT is faster than PROCESS because there's much less work involved in inserting a file than there is in processing it as a template. It's not going to work for us in our current example because of the year and author variables in the footer that need resolving. If we INSERT the footer as it is, we'll see the [% year %] and [% author %] directives passed through as literal text.

However, we can hardcode the variables in the footer to make it a fixed block of text that we can then load using INSERT. For example:

```
    <hr />



    <div align="middle">

      &copy; Copyright 2003 Arthur Dent

    </div>

  </body>

</html>
```

Although we've no longer got the benefit of using variables or other template directives, we are still defining the footer in one place where we can easily make changes, should we ever need to.

In most day-to-day applications, the difference in speed between INSERT and PROCESS isn't going to be noticeable unless you really go looking for it. You're generally better off using whatever is most convenient for you, the template author. Worry about performance only if and when it ever becomes an issue. With this in mind, we'll leave our variables in the footer and continue to use PROCESS.

The other directives for loading templates are INCLUDE and WRAPPER, which we'll be looking at shortly.

2.2.2 Benefits of Modularity

Separating commonly used blocks of markup into reusable template component files in this way allows you to take a modular approach to building your web content. This brings a number of important benefits.

The first is that the page templates become easier to write, edit, and maintain. You can quickly and easily add new pages by reusing existing template components to do the repetitive work, leaving the template author to concentrate on adding the core content. When it comes to updating the content, it becomes a lot easier to find what you're looking for because you don't have to pore through great chunks of HTML markup that define header, footers, menus, and other user interface elements.

In other words, we're achieving a *clear separation of concerns* between the core content of the pages and the parts that deal mainly with presentation. Content authors can concentrate on writing content without worrying about what kind of fancy user interface the web designers have dreamt up to fit around it

The second benefit is that the headers, footers, and other template components can easily be updated at any time, and need to be modified only in one place. Changing the copyright messages, the background color, or perhaps the layout of the footer, for *every* page on the site, becomes as easy as editing the one template component file and then processing the page templates to rebuild the site content.

So the clear separation of concerns also works the other way around. Web designers can concentrate on building a nice user interface for the entire site without having to worry too much about the content of individual pages.

Even if you're the all-in-one web designer, content author, and webmaster for your site, it is still useful to maintain a clear separation between these different aspects. You may have many hats to wear, but you'll be most comfortable wearing just one at a time.

2.3 Defining Variables

Our current use of *tpage* for processing templates is hardly streamlined. We're spending a lot of time typing variable values on the command line, something that can only get worse as we add more pages that require processing to the site.

It would be easy to mistype the value for a variable, for example, or perhaps supply the wrong value altogether. You wouldn't see any complaint from the Template Toolkit. It would just go right ahead and process the template with whatever values you supplied, possibly leading to an error on an HTML page that could go unnoticed.

2.3.1 Configuration Template

A better approach is to create a template component that defines any commonly used variables in one place. Example 2-7 shows our *config* template.

Example 2-7. config

```
[%  author = 'Arthur Dent'

    bgcol  = '#FF6600'    # orange

    year   = 2003

    copyr  = "Copyright $year $author"

-%]
```

You can define any number of variables in a single directive, as Example 2-7 illustrates. The Template Toolkit is very flexible in terms of the syntax it supports inside its tags, allowing you to spread your directives over several lines, adding as little or as much whitespace as you like for formatting purposes. You don't need to put each on a separate line as we have here—they can all go on the same line as long as some kind of whitespace is separating them. In the end, it's your choice. The Template Tooolkit isn't fussy about how you lay out your directives, as long as you follow the basic rules of syntax, which we'll be introducing throughout this chapter and describing in greater detail in Chapter 3.

2.3.1.1 Comments

You can add comments to annotate your code, as shown in the second line of Example 2-7: # orange. A comment starts with the # character and continues to the end of the current line. The comment is ignored by the Template Toolkit, and processing continues as normal on the next line.

If # is used as the first character immediately following the opening [% tag, the Template Toolkit ignores the entire directive up to the closing %]:

```
[%# this is a comment

    this line is also part of the comment

%]
```

2.3.1.2 Variable values

In Example 2-7, the four variables set are author, bgcol, year, and copyr. The first two are defined as the literal strings 'Arthur Dent' and '#FF6600'. The ' single quotation marks surrounding the values indicate that the contents should be used as provided. This makes it clear to the Template Toolkit that the # character in the definition for bgcol, for example, is part of the value and not the start of a comment. The third variable, year, is defined as the integer value 2003. Numbers such as these (and also floating-point numbers such as 2.718) don't need to be quoted, but can be if you prefer.

The last variable, copyr, shows an example of a double-quoted string, in which the value is enclosed by " characters. Here the Template Toolkit looks for any references to variables embedded in the string, denoted by the $ character, and replaces (*interpolates*) them for the corresponding values. In this example, the values for year and author will be interpolated into the string, resulting in the copyr variable being set to "Copyright 2003 Arthur Dent".

2.3.2 Loading the Configuration Template

The *config* template can now be loaded using the PROCESS directive to gain access to these variable definitions. This is shown in Example 2-8, which also defines the title variable specific to this page. This is really no different from the way you might define a constant or global variable at the start of a program in Perl or some other programming language. It's good practice to do this at the top of the file, where any future changes can easily be made.

Example 2-8. earth.tt

```
[% title = 'Earth' -%]
[% PROCESS config -%]
[% PROCESS header %]

  <p>
    Mostly Harmless.
  </p>

[% PROCESS footer %]
```

Notice the - character placed immediately before the closing %] tags at the end of the directives on the first two lines. This tells the Template Toolkit to remove, or *chomp*, the newline and any other whitespace following the directive. Some older web browsers don't like to see whitespace appearing before the opening <html> element, so this ensures that the *header* file is inserted right at the top of the output. In effect, it is as if we had written the template like so:

```
[% title = 'Earth' %][% PROCESS config %][% PROCESS header %]
...
```

Now the template can be processed using *tpage* without the need to provide variable values as command-line arguments:

```
$ tpage earth.tt > earth.html
```

2.3.2.1 Merging directives

The start of each page template can be simplified by defining the title variable and the PROCESS directives within a single directive tag. Each command is separated from the next by a ; (semicolon) character.

For example, we can write:

```
[% title = 'Earth';
   PROCESS config;
   PROCESS header
%]
```

instead of the more verbose:

```
[% title = 'Earth' -%]
[% PROCESS config -%]
[% PROCESS header %]
```

There's no need for a semicolon at the end of the last directive, but the Template Toolkit won't complain if it finds one there. As we saw earlier, semicolons aren't required between variable definitions that appear one after another. However, a semicolon is required if you switch from setting variables (which is technically the SET directive, although the explicit keyword is rarely used) to another kind of directive (e.g., PROCESS) in the same tag:

```
[% pi  = 3.142      # semicolon optional
   e   = 2.718      # "     "     "     "
   i   = 1.414;     # semicolon mandatory
   PROCESS config;  # "     "     "     "
   phi = 1.618      # semicolon optional
%]
```

The distinction becomes a little more obvious when we use the SET keyword explicitly and add some whitespace to format the directives more clearly:

```
[% SET pi  = 3.142
       e   = 2.718
       i   = 1.414;

   PROCESS config;

   SET phi = 1.618
%]
```

There's one final improvement we can make to the block at the start of our page templates. The two PROCESS directives can be merged into one, with the names of the templates separated by a + character:

```
[% title = 'Earth';

   PROCESS config
         + header
%]
```

The general rule of whitespace being insignificant inside directives applies equally well to the PROCESS directive, allowing us to list all the files on the same line, or across a number of lines, as we've done here. This flexibility allows us to lay out this header block in such a way that it's clear from a glance what's going on, and with the bare minimum of extra syntax cluttering up this high-level view.

Example 2-9 shows this in the context of a complete page template.

Example 2-9. magrethea.tt

```
[% title = 'Magrethea';

   PROCESS config
         + header
-%]

<p>
  Home of the custom-made
  luxury-planet building industry.
</p>

[% PROCESS footer %]
```

2.4 Generating Many Pages

The *tpage* program is fine for processing single templates, but isn't really designed to handle the many pages that comprise a typical web site. For this, *ttree* is much more appropriate. It works by drilling down through a source directory of your choosing, looking for templates to process. The output generated is saved in a corresponding file in a separate destination directory.

In addition to working well with a large number of template files, *ttree* also provides a much greater range of configuration options that allow you to modify the behavior of the Template Toolkit when processing templates. This allows you to further simplify the process of generating and maintaining web content in a number of interesting ways that we'll explore throughout this section.

Our templates will need to be organized a little more carefully when using *ttree*. In particular, we need to separate those page templates that represent complete HTML pages (*hello.tt*, *goodbye.tt*, *earth.tt*, and *magrethea.tt* in our previous examples) from those that are reusable template components (*config*, *header*, and *footer*).

2.4.1 Creating a Project Directory

We'll start by creating a directory for our web site, complete with subdirectories for the source templates for HTML pages (*src*), a library of reusable template components (*lib*), and the generated HTML pages (*html*). We'll also create a directory for miscellaneous files (*etc*), including a configuration file for *ttree*, and another (*bin*) for any scripts we accrue to assist in building the site and performing maintenance tasks.

```
$ cd /home/dent

$ mkdir web

$ cd web

$ mkdir src lib html etc bin
```

2.4.2 ttree Configuration File

Now we need to define a configuration file for *ttree*. Example 2-10 shows an example of a typical *etc/ttree.cfg* file.

Example 2-10. etc/ttree.cfg

```
# directories

src  = /home/dent/web/src

lib  = /home/dent/web/lib

dest = /home/dent/web/html


# copy images and other binary files

copy = \.(png|gif|jpg)$


# ignore CVS, RCS, and Emacs temporary files

ignore = \b(CVS|RCS)\b

ignore = ^#


# misc options

verbose

recurse
```

Options can appear in any order in the configuration file. In certain cases (such as lib, copy, and ignore), an option can be repeated any number of times.

The first section defines the three important template directories:

```
# directories

src  = /home/dent/web/src

lib  = /home/dent/web/lib

dest = /home/dent/web/html
```

The src option tells *ttree* where to look for HTML page templates. The lib option (of which there can be many) tells it where the library of additional template components can be found. Finally, the dest option specifies the destination directory for the generated HTML pages.

The next two sections provide regular expressions that *ttree* uses to identify files that should be copied rather than processed through the Template Toolkit (copy), and to identify files that should be ignored altogether (ignore):

```
# copy images and other binary files

copy = \.(png|gif|jpg)$



# ignore CVS, RCS, and Emacs temporary files

ignore = \b(CVS|RCS)\b

ignore = ^#
```

In this example, we're setting the options so that any images with png, gif, or jpg file extensions are copied, and any CVS or temporary files left lying around by our favorite text editor are ignored.

The next section sets two *ttree* flags:

```
# misc options

verbose

recurse
```

The verbose flag causes *ttree* to print additional information to STDERR about what it's doing, while it's doing it. The recurse flag tells it to recurse down into any sub-directories under the src directory.

2.4.3 Running ttree for the First Time

When you run *ttree* for the first time, it will display the following prompt, which asks if you'd like it to create a default *.ttreerc* file:

```
Do you want me to create a sample '.ttreerc' file for you?

(file: /home/dent/.ttreerc)   [y/n]:
```

Answer y to have it create the file in your home directory.

This file is used to provide a default configuration for *ttree*. If you've got only one web site to maintain, you can copy the contents of the *etc/ttree.cfg* file into it and run *ttree* without any command-line options:

```
$ ttree
```

If you've got more than one site to maintain, you'll probably want to keep separate configuration files for each. In that case, you can use the -f command-line option to provide the name of the configuration file when you invoke *ttree*:

```
$ ttree -f /home/dent/web/etc/ttree.cfg
```

2.4.4 Using a Build Script

Rather than providing a command-line configuration option for *ttree* each time you use it, you may prefer to write a simple build script that does it for you (as in Example 2-11).

Example 2-11. bin/build

```
ttree -f /home/dent/web/etc/ttree.cfg $@
```

The $@ at the end of the line passes any command-line arguments on to the *ttree* program, in addition to the -f option that is provided explicitly.

2.4.5 ttree Configuration Directory

Another alternative is to set the cfg option in the *.ttreerc* file to denote a default directory for *ttree* configuration files. You could set this to point to the project directory:

```
cfg = /home/dent/web/etc
```

and then invoke *ttree* with the short name of the configuration file:

```
$ tpage -f ttree.cfg
```

If you have many different web sites to maintain, another option is to create one general directory for *ttree* configuration files and use symbolic links from this directory to the project-specific files. The *.ttree* directory in your home directory is a common choice. In the *.ttreerc* file, we specify it like so:

```
cfg = /home/dent/.ttree
```

Then we prepare the directory, creating a symbolic link to our project-specific configuration file. We give it a memorable name (e.g., dentweb) to distinguish it from the various other *ttree.cfg* files that we may create links to from this directory:

```
$ cd /home/dent

$ mkdir .ttree

$ cd .ttree

$ ln -s /home/dent/web/etc/ttree.cfg dentweb
```

With these changes in place, *ttree* can then be invoked using the -f option to specify the *dentweb* configuration file:

```
$ tpage -f dentweb
```

The settings in the *.ttreerc* file and the magic of symbolic links result in *ttree* ending up with the right configuration file without us having to specify the full path to it every time. The other benefit of this approach is that *ttree* can be invoked from any directory and the correct configuration file will still be located.

2.4.6 Calling ttree Through the Build Script

From now on we'll assume that the *bin/build* script invokes *ttree* with the appropriate option to locate the configuration file. For the sake of clarity, we'll use it in the examples that follow whenever we want to build the site content, rather than calling *ttree* directly. Any other commands that you want performed when the site is built (e.g., copying files, restarting the web server or database) can also be added here.

As we saw in Example 2-11, any command-line options that we provide to the script are forwarded to *ttree*. One particularly useful option is -h, which provides a helpful summary of all the different *ttree* options:

```
$ bin/build -h

ttree 2.63 (Template Toolkit version 2.10)


usage: ttree [options] [files]


Options:

   -a      (--all)          Process all files, regardless of modification

   -r      (--recurse)      Recurse into sub-directories

   -p      (--preserve)     Preserve file ownership and permission

   -n      (--nothing)      Do nothing, just print summary (enables -v)

   -v      (--verbose)      Verbose mode
```

```
  -h      (--help)         This help
  -dbg    (--debug)        Debug mode
  -s DIR  (--src=DIR)      Source directory
  -d DIR  (--dest=DIR)     Destination directory
  -c DIR  (--cfg=DIR)      Location of configuration files
  -l DIR  (--lib=DIR)      Library directory (INCLUDE_PATH)  (multiple)
  -f FILE (--file=FILE)    Read named configuration file     (multiple)

File search specifications (all may appear multiple times):
  --ignore=REGEX           Ignore files matching REGEX
  --copy=REGEX             Copy files matching REGEX
  --accept=REGEX           Process only files matching REGEX

Additional options to set Template Toolkit configuration items:
  --define var=value       Define template variable
  --interpolate            Interpolate '$var' references in text
  --anycase                Accept directive keywords in any case.
  --pre_chomp              Chomp leading whitespace
  --post_chomp             Chomp trailing whitespace
  --trim                   Trim blank lines around template blocks
  --eval_perl              Evaluate [% PERL %] ... [% END %] code blocks
  --load_perl              Load regular Perl modules via USE directive
  --pre_process=TEMPLATE   Process TEMPLATE before each main template
  --post_process=TEMPLATE  Process TEMPLATE after each main template
  --process=TEMPLATE       Process TEMPLATE instead of main template
  --wrapper=TEMPLATE       Process TEMPLATE wrapper around main template
  --default=TEMPLATE       Use TEMPLATE as default
  --error=TEMPLATE         Use TEMPLATE to handle errors
  --start_tag=STRING       STRING defines start of directive tag
  --end_tag=STRING         STRING defined end of directive tag
  --tag_style=STYLE        Use pre-defined tag STYLE
  --plugin_base=PACKAGE    Base PACKAGE for plugins
  --compile_ext=STRING     File extension for compiled template files
  --compile_dir=DIR        Directory for compiled template files
  --perl5lib=DIR           Specify additional Perl library directories
```

2.4.7 A Place for Everything, and Everything in Its Place

Before we can run the build script to generate the site content, we will need to move our page and library template files into place.

The source templates for the HTML pages should now be moved into the *src* directory where *ttree* can find them. The HTML files that *ttree* generates in the *html* output directory will be given the same filename as the *src* template from which they are generated. For this reason, we'll be using a *.html* file extension on our page templates from now on.

Also, move the template components *config*, *header*, and *footer* into the *lib* directory. These are (for now) also identical to those shown in the earlier examples.

2.4.8 Running the Build Script

Now we can run the *bin/build* script to invoke *ttree* to build the site content:

```
$ bin/build

ttree 2.63 (Template Toolkit version 2.10)


      Source: /home/dent/web/src

 Destination: /home/dent/web/html

Include Path: [ /home/dent/web/lib ]

      Ignore: [ \b(CVS|RCS)\b, ^# ]

        Copy: [ \.(png|gif|jpg)$ ]

      Accept: [ * ]


  + earth.html

  + magrethea.html
```

The sample output from *ttree* shown here indicates that two page templates, *earth.html* and *magrethea.html*, were found in the *src* directory. The + character to the left of the filenames indicates that the templates were processed successfully. Corresponding *earth.html* and *magrethea.html* files will have been created in the *html* directory containing the output generated by processing the templates.

Now that we've set up *ttree* and told it where our page templates are located, we can add new pages to the site by simply adding them to the *src* directory. When you next run the build script, *ttree* will locate the new page templates, even if they're located deep in a subdirectory (thanks to the recurse option), and process them into the corresponding place in the *html* directory.

You can now build all the static web pages in your site using a single, simple command.

2.4.9 Skipping Unmodified Templates

When *ttree* is run it tries to be smart in working out which templates need to be processed and which don't. It does this by comparing the file modification time of the page template with the corresponding output file (if any) that it previously generated.

Run the *bin/build* script again, and the + characters to the left of the filename change to the - character:

```
$ bin/build

ttree 2.63 (Template Toolkit version 2.10)


      Source: /home/dent/web/src

 Destination: /home/dent/web/html

Include Path: [ /home/dent/web/lib ]

      Ignore: [ \b(CVS|RCS)\b, ^# ]

        Copy: [ \.(png|gif|jpg)$ ]
```

```
Accept: [ * ]

 - earth.html                  (not modified)

 - magrethea.html              (not modified)
```

This indicates that the templates weren't processed the second time around, with the message to the right of the filenames explaining why. In this case, *ttree* has recognized that the source templates, *src/earth.html* and *src/magrethea.html*, haven't been modified since the corresponding output files, *html/earth.html* and *html/magrethea.html*, were created. Given that nothing has changed, there's no need to reprocess the templates.

There may be times when you want to force *ttree* to build a particular page or even all the pages on the site, regardless of any file modification times. You can process one or more pages by naming them explicitly on the command line:

```
$ bin/build earth.html magrethea.html
```

One time that you might want to force all pages to be rebuilt is when you modify a header, footer, or some other template component that is used by all the pages. Unfortunately, *ttree* isn't smart enough to figure out which library templates are used by which page templates.[1] The -a option tells *ttree* to ignore file modification times and process all page templates, regardless:

[1] This occurs not because *ttree* is being lazy. It's actually very difficult, if not impossible, to do it accurately without processing the templates in their entirety. By this time, the Template Toolkit has already done the hard work, so there's nothing to be gained by discovering that the template didn't need processing after all.

```
$ bin/build -a
```

2.5 Adding Headers and Footers Automatically

In addition to the fact that *ttree* works well with large collections of page templates, it also has the benefit of providing a large number of configuration options that allow you to change the way it works and how it uses the underlying Template Toolkit processor. Two of the most convenient and frequently used options are pre_process and post_process. These allow you to specify one or more templates that should be *automatically* added to the top or bottom of each page template, respectively. This can be used to add standard headers and footers to a generated page, but pre- and postprocessed templates may not generate any visible output at all. For example, we can use a preprocessed template to configure some variables that we might want defined for use in the page template or other template components.

The following can be added to the bottom of the *etc/ttree.cfg* file to have the *config* and *header* templates preprocessed (in that order so that we can use variables defined in *config* in the *header*) and the *footer* template postprocessed:

```
pre_process  = config
pre_process  = header
post_process = footer
```

Now the page templates can be made even simpler, as Example 2-12 shows.

Example 2-12. src/magrethea.html

```
[% title = 'Magrethea' -%]


<p>
  Home of the custom-made
  luxury-planet building industry.
</p>
```

Remember that you'll need to use the -a option to force *ttree* to rebuild all pages in the site to have the changes take effect:

```
$ bin/build -a
```

2.5.1 Defining META Tags

There is one problem with this approach. The *header* template is processed in its entirety before the main page template gets a look in. This means that the title variable isn't set to any value when the *header* is processed. It doesn't get set until the page template is processed, by which time it's too late for the *header* to use it.

The Template Toolkit won't complain if it encounters a variable for which it doesn't have a value defined. Instead, it will quietly use an empty string (i.e., nothing at all) for the value of the variable and continue to process the remainder of the template. The DEBUG option (described in the Appendix) can be set to have it raise an error in these cases, and can be useful to help track down mistyped variable names and those that have somehow eluded definition.

We can use the META directive to solve our immediate problem. It works by allowing us to define values within the page template that *are* accessible for use in the *header* and any other preprocessed templates, *before* the main page template is itself processed.

Example 2-13 shows how this is done. Instead of defining the title in a SET directive (which technically we were, even if we had omitted the SET keyword for convenience), we use the META directive, but otherwise leave the definition of the variable unmodified.

Example 2-13. src/milliways.html

```
[% META title = 'Milliways' %]


<p>
  The Restaurant at the
  End of the Universe.
</p>
```

Variables defined like this are made available as soon as the template is loaded. This happens *before* any of the preprocessed templates are processed so that these META variables are defined and ready for use.

There are some subtle differences between META variables and normal SET variables. The first is that you can't use double-quoted strings to interpolate other variables into the values for META variables. You *can* use double-quoted strings, but you can't embed variables in them and expect them to get resolved. The simple reason for this is that META variables are defined before the template is processed with any live data. At this time, there aren't any variables defined, so there's no point trying to use them.

The second difference is that the variables must be accessed using the template. prefix:

```
[% template.title %] not [% title %]
```

The template variable is a special variable provided by the Template Toolkit containing information about the current page template being processed. It defines a number of items, including the name of the template file (template.name) and the modification time (template.modtime), as well as any META variables defined in the template (template.title).

The dot operator, ., is the Template Toolkit's standard notation for accessing a variable such as title that is one small part of a larger, more complex data structure such as template. It doesn't matter for now (or generally at all) how this is implemented behind the scenes because the dot operator hides or *abstracts* that detail from you so that you don't need to worry about it.

We'll be coming back to the dot operator later on in this chapter when we look at defining and using complex data structures. For now, it is sufficient to know that template.title is how we access the title META variable defined in the main page template.

We can easily modify our *header* template to accommodate these requirements and restore the page title to the generated header (see Example 2-14).

Example 2-14. lib/header

```
<html>
  <head>
    <title>[% author %]: [% template.title %]</title>
  </head>


  <body bgcolor="[% bgcol %]">
    <h1>[% template.title %]</h1>
```

2.6 More Template Components

You can create any number of different reusable template components to help you generate the content for your web site. Whenever you find yourself repeating the same, or a similar, block of markup in more than one place, you might want to consider moving it into a separate template file that you can then use and reuse whenever you need it. This not only saves you a lot of typing, but also ensures that the HTML generated in each place you use it is identical, or as near to identical as you would like it to be, accounting for any variables that might change from one use to the next.

Example 2-15 shows a template component for displaying an entry from Arthur's favorite reference book.

Example 2-15. lib/entry

```
<p>
 The Hitch Hiker's Guide to the Galaxy
 has this to say on the subject of
 "[% title %]".
</p>

<table border="0">
 <tr valign="top">
   <td>
     <b>[% title %]:</b>
   </td>
   <td>
     [% content %]
   </td>
 </tr>
</table>
```

The template uses two variables, title and content. The value for title can in this case be copied from template.title, thereby providing the title set in the META directive for the page. A value for content will be set explicitly for the sake of simplicity. These variables can be set either before the PROCESS directive:

```
[% title   = template.title
   content = 'Mostly harmless'
%]

[% PROCESS entry %]
```

or as part of the PROCESS directive, following the template name as additional arguments:

```
[% PROCESS entry
     title   = template.title
     content = 'Mostly harmless'
%]
```

The end result is the same. The Template Toolkit treats all variables as global by default so that you can define a variable in one template and use it later in another without having to explicitly pass it as an argument every time. In both of the preceding examples, the title and content variables are defined globally and can subsequently be used in both the called template (*entry*) and the calling template (*earth.tt*) after the point of definition.

In the following fragment, for example, the reference to the content variable at the end of the template will generate the value "Mostly harmless" as set in the earlier PROCESS directive:

```
[% PROCESS entry
     title   = template.title
     content = 'Mostly harmless'
%]

[% content %]   # Mostly harmless
```

2.6.1 The INCLUDE Directive

There may be times when you would rather keep the definition of certain variables local to a particular template. The INCLUDE directive provides a way of doing this. In terms of syntax, it is used in exactly the same way as the PROCESS directive in all except the keyword.

The key difference between INCLUDE and PROCESS is that INCLUDE *localizes* any variables that are passed to the template as arguments in the directive. The variables passed have local values for the template component being processed by INCLUDE, but then revert to their previous values or undefined states.

In the following fragment, we define two variables at the start of the template whose values we would like to preserve to be used in the sentence at the end:

```
[% name  = 'Zaphod Beeblebrox'
   title = 'President of the Galaxy'
%]

[% INCLUDE entry
     title   = 'Earth'
     content = 'Mostly harmless'
%]

Hi!  I'm [% name %], [% title %].
```

The INCLUDE directive provides local definitions for the title and content variables for the *entry* template to display. However, the original value for the title variable will be left untouched, and there will be no trace of the content variable outside of the *entry* template.

The final line of the template generates the output that we're expecting:

```
Hi!  I'm Zaphod Beeblebrox, President of the Galaxy.
```

Had we used PROCESS instead of INCLUDE, the value for title would have been overwritten and the output generated by the final line would incorrectly read:

```
Hi!  I'm Zaphod Beeblebrox, Earth.
```

There is one important caveat to be aware of. The INCLUDE directive only localizes simple variables. Any complex variables containing dot operators are effectively global regardless of whether you use INCLUDE, PROCESS, or any other directive.

Dotted variables are a little like Perl's package variables. In Perl, you can refer to a variable as, for example, $My::Dog::Spot. This tells Perl the precise location for the variable $Spot in the My::Dog package. In the Template Toolkit, the equivalent variable would be something like my.dog.spot.

On the other hand, a Perl variable written as just $Spot could be either a "global" (for these purposes) variable defined in the current package, or a lexically scoped variable in the current subroutine, for example. Similarly, in the Template Toolkit, the equivalent variable spot could also be a global variable or a local copy created by invoking a template using INCLUDE.

The explanation isn't important as long as you remember the simple rule: the INCLUDE localizes only simple variables that don't contain any "." dots.

2.6.2 Setting Default Values

When you define a reusable template component, you may want to provide default values for any variables used in the template. For example, the following template component might want to ensure that sensible values are provided for the <title> element and bgcolor attribute in the <body>, even if the respective title and bgcol variables aren't set:

```
<html>

  <head>

    <title>[% title %]</title>

  </head>

  <body bgcolor="[% bgcol %]">

    ...
```

2.6.2.1 The DEFAULT directive

One way to achieve this is by using the DEFAULT directive. The syntax is the same as SET in everything but the keyword, allowing you to provide default values for one or more variables:

```
[% DEFAULT

     title = "Arthur Dent's Web Site"

     bgcol = '#FF6600'

-%]

<html>

  <head>

    <title>[% title %]</title>

  </head>

  <body bgcolor="[% bgcol %]">

    ...
```

The key difference between DEFAULT and SET is that DEFAULT will set the variable to the value prescribed only if it is currently undefined, if it is set to an empty string, or if it contains the number zero. (Perl programmers will recognize the similarity with Perl's idea about what is *true* and *false* when it comes to the value of a variable.) The component will use any existing values for title and bgcol, either defined globally or passed as explicit arguments when the template is used. Otherwise, it will use the values provided in the DEFAULT directive.

2.6.2.2 Expressions

Another approach is to use Template Toolkit *expressions* instead of just variables. Expressions allow you to make logical statements including the and and or operators, both of which can be written in either upper- or lowercase. For example, we can write:

```
[% bgcol or '#FF6600' %]
```

instead of just:

```
[% bgcol %]
```

The tertiary ?: operator is another option. It provides the equivalent of an IF...THEN...ELSE construct, in which the expression to the left of the ? is evaluated to determine whether it is true or false. If true, whatever comes after the ? and before the : is used. Otherwise, it returns whatever follows the :.

Here's an example showing how the ?: operator can be used to generate an appropriate title for the page:

```
[% title ? "Arthur Dent: $title"

        : "Arthur Dent's Web Site"

%]
```

If the title variable is set, the string "Arthur Dent: $title" is used. This uses variable interpolation to insert the current value for the title variable into the string, following Arthur's name. If title isn't set to anything that the Template Toolkit considers meaningfully true, the string "Arthur Dent's Web Site" is instead used. The expression doesn't need to be split across two lines as we've shown here, but in this case it helps to make the code clearer and easier to read.

So if title is set to Earth, the directive will generate the following output:

```
Arthur Dent: Earth
```

If the title isn't set, it will instead generate this output:

```
Arthur Dent's Web Site
```

Expressions can also contain comparison operators, as shown in the following example. These are discussed in detail in Chapter 3.

```
[% age > 18 ? 'Welcome to my site...'

          : "Sorry, but you're not old enough..."

%]
```

2.6.2.2.1 = versus = =

One important distinction worth mentioning now is the difference between = and = =. The first performs an assignment, setting the variable named on the left to the value (or expression) on the right:

```
[% foo = bar %]
```

The second is the equality comparison operator, which tests to see whether the string values of the items on either side are identical:

```
[% foo =  = bar ? 'equal' : 'not equal' %]
```

2.6.2.2.2 Setting variables using expressions

Expressions can also be used to set the value of a variable. For example, the pagetitle variable can be set to either of the values previously shown, depending on the setting of title, using the following code:

```
[% pagetitle = title ? "Arthur Dent: $title"

                 : "Arthur Dent's Web Site"

%]
```

It's perfectly valid to use a variable in an expression to update the same variable. Everything to the right of the = is evaluated first, and the resulting value is then used to set the variable specified to the left of the =:

```
[% title = title ? "Arthur Dent: $title"

              : "Arthur Dent's Web Site"

%]
```

2.6.2.2.3 Setting variables using directives

You can also assign the output of a directive to a variable. In the following example, the *header* template is processed using the PROCESS directive and the generated output is stored in the headtext variable:

```
[% headtext = PROCESS header %]
```

2.6.3 The IF Directive

The IF directive can be used to encode more complex conditional logic in templates. It evaluates the expression following the IF keyword, which in these examples will be a simple variable. If the expression is true, the following block, up to the matching END directive, is processed. Otherwise, it is ignored.

Here's a simple example:

```
<body

[%- IF bgcol -%]

 bgcolor="[% bgcol %]"

[%- END -%]

>
```

This example uses an IF block to add the bgcolor attribute to the HTML <body> element, but only if the bgcol variable is defined and contains a true value. By careful placement of - characters at the start and end of the IF and END directives, we're enabling the Template Toolkit's prechomping and postchomping facility. This removes the newline characters before the [% tags and after the %] tags so that the output lines up in the correct place in the <body> element.

So, for a bgcol value of #FF6600, the following output would be generated:

```
<body bgcolor="#FF6600">
```

For an undefined bgcol, we would instead see the following:

```
<body>
```

Like many of the Template Toolkit directives that expect a block to follow, the IF directive can be used in *side-effect* notation.

For example, you can write:

```
[% INCLUDE header IF title %]
```

instead of the more laborious:

```
[% IF title; INCLUDE header; END %]
```

This works only when you've got a single directive or variable as the content for the block—in this example, it's the INCLUDE header directive. Our earlier example, which constructed the <body> tag, included both text and a reference to the bgcol variable in the block. However, we can write this using a double-quoted string to interpolate the value for bgcol:

```
<body [%- " bgcolor=\"$bgcol\"" IF bgcol %]>
```

Matters are complicated a little by the need to escape the double quotes inside the double quotes. The \ character tells the Template Toolkit that the following " is part of the string, and not the quote that terminates it. Overall it's an improvement over the more explicit IF...END form and illustrates a useful principle.

You can add an ELSE block after the IF block, which will be processed if the variable (or more generally, the expression) is false. For example:

```
[% IF bgcol -%]

<body bgcolor="[% bgcol %]">

[%- ELSE -%]

<body>

[%- END -%]
```

There is also the ELSIF directive, which allows you to define different blocks for different conditions:

```
[% IF name =  = 'Arthur Dent'

   OR name =  = 'Ford Prefect' %]

Hello [% name %]!

[% ELSIF name.match('(?i:vogon)') %]

I'm sorry, but there's no one at home.

Please don't bother calling again.

[% ELSE %]

Hello World!

[% END %]
```

In this example, the ELSIF expression uses the match virtual method to test whether the name contains anything looking remotely Vogon. The argument passed to the match method is a Perl regular expression, allowing us to use the (?i:...)

grouping to construct a case-insensitive match. An ELSE block is also provided in case neither the IF nor ELSIF conditions match.

The SWITCH directive, described in detail in Chapter 4, provides an alternative for more complicated multiway matching.

2.7 Wrapper and Layout Templates

Now it's time to bring out some of the bigger guns of the Template Toolkit. The WRAPPER directive and layout templates let you define a common look for web pages in a single file, rather than scattering the components over *header* and *footer* files.

2.7.1 The WRAPPER Directive

The *entry* template from Example 2-15 works well when the content to be displayed is relatively simple. However, it quickly becomes cumbersome for longer entries such as the one shown here:

```
[% INCLUDE entry
   title   = 'Vogon Poetry'
   content = 'Vogon poetry is of course the
           third worst in the Universe.
           The second worst is that of...

           ...etc...

           ...in the destruction of the
           planet Earth'
%]
```

Special care must be taken when quoting content that contains quote characters. Consider the following extract that illustrates this problem:

```
Grunthos is reported to have been "disappointed"
by the poem's reception.
```

If this is enclosed in single-quote characters, the apostrophe in "poem's" must be escaped by preceding it with a backslash \ character (the apostrophe and single-quote characters are one and the same for these purposes):

```
[% INCLUDE entry
   title   = 'Grunthos the Flatulent'
   content = 'Grunthos is reported to have
           been "disappointed" by the
           poem\'s reception.'
%]
```

Another alternative is to use double quotes to define the variable, allowing single quotes to remain as they are. But in this case, any occurrences of double quotes will then need to be escaped:

```
[% INCLUDE entry
   title   = 'Grunthos the Flatulent'
   content = "Grunthos is reported to have
           been \"disappointed\" by the
           poem's reception."
%]
```

A better solution is to use the WRAPPER directive. It works in a similar way to INCLUDE, but uses an additional END directive to enclose a block of template content. The WRAPPER directive uses this block as the value for the content variable:

```
[% WRAPPER entry
     title = 'Grunthos the Flatulent'
%]
   Grunthos is reported to have
   been "disappointed" by the
   poem's reception.
[% END %]
```

The immediate benefit in this example is that the extract is now a block of plain text rather than a quoted string. There is no longer any need to escape the quote characters within it.

The WRAPPER block can contain any combination of text and template directives, even including other nested WRAPPER blocks. The following fragment shows a simple example in which the reaction variable is used to report Grunthos' reaction:

```
[% reaction = 'disappointed' %]


[% WRAPPER entry
     title = 'Grunthos the Flatulent'
%]
   Grunthos is reported to have
   been "[% reaction %]" by the
   poem's reception.
[% END %]
```

The WRAPPER block is processed first to resolve any directives within it. Then the complete block, including any output generated dynamically by embedded directives, is passed to the *entry* template as the value for the content variable.

It's no coincidence that we chose content as a variable name in the *entry* template in Example 2-15, knowing full well that we would later use it in this example for WRAPPER. The WRAPPER directive always assigns the block content to the content variable, and in that sense it's one of the Template Toolkit's "special" variables, like the template variable that we used earlier. However, there's nothing to stop you from using it as a regular variable, and indeed it makes a good choice in any template for a variable that you might one day want to define as a block in a WRAPPER directive.

The end result is that the *entry* template works as expected, whether we call it using INCLUDE and pass the content explicitly as a variable, or call it using WRAPPER and define the content implicitly in the enclosed block.

2.7.2 Using an Automatic Wrapper Template

In Examples Example 2-4 and Example 2-14, we created separate *header* and *footer* files to add to the start and end of each HTML page generated. One problem with this approach is that neither file contains valid HTML markup. The *header* provides the opening tag of the html element, for example, but the corresponding closing tag is located at the end of the *footer* file.

Having HTML elements split across separate files makes them harder to maintain, and increases the likelihood of them being accidentally mismatched or incorrectly nested. It is also likely to confuse or infuriate any HTML-aware text editors or validation tools that you may be using.

A better approach is to use a *wrapper* template to combine the *header* and *footer* into one template. The content variable is used to denote the position for the page content. This is shown in Example 2-16.

Example 2-16. lib/wrapper

```
<html>
  <head>
    <title>[% author %]: [% template.title %]</title>
```

```
 </head>

 <body bgcolor="[% bgcol %]">
  <h1>[% template.title %]</h1>

  [% content %]

  <hr />

  <div align="middle">
   &copy; [% copyr %]
  </div>
 </body>
</html>
```

We need to modify the *etc/ttree.cfg* file to specify the new *wrapper* template using the wrapper option. The fact that our *wrapper* template happens to be called *wrapper* is entirely coincidental (but intentional). We could have named the file *tom*, *dick*, *larry*, or something else if we wanted to, but it wouldn't be as succinct or descriptive as *wrapper*.

We're still using the pre_process option to load the *config* template, but we can now remove the references to the *header* and *footer* (or comment them out as shown here), replacing them with a single wrapper option:

```
pre_process  = config
wrapper      = wrapper
# pre_process  = header
# post_process = footer
```

With the wrapper option in place, the Template Toolkit processes the main page template (after preprocessing the *config* template) and then calls the *wrapper* template, passing the generated page content as the content variable. It has the same effect as if there were an explicit WRAPPER directive around the entire page content:

```
[% WRAPPER wrapper %]
   The entire page content goes here...
[% END %]
```

Of course, the benefit of having the Template Toolkit apply a *wrapper* automatically is that you don't need to edit any of your page templates to add it explicitly. You can switch from using pre_process and post_process to wrapper, or you can change the name of any of the *header*, *footer*, or *wrapper* templates, without having to make any changes to your core content.

To put the change into effect, run the *bin/build* script with the -a option to have it rebuild all pages in the site:

```
$ bin/build -a
```

2.7.3 Using Layout Templates

Most real web sites will require far more complex layout templates than the simple *wrapper* we saw in Example 2-16. A common practice is to use HTML tables to place different elements such as headers, footers, and menus in a consistent position and formatting style. These elements may themselves be built using tables and other HTML elements, perhaps nested several times over. This can quickly lead to confusing markup that is hard to read and even harder to update.

Consider the following example, which illustrates how difficult nested tables can be to write and maintain:

```
<table border="0" cellpadding="0" cellspacing="0">
  <tr valign="top">
    <td>
      <table border="0">
        <tr>
          <td>
            Oh Dear!
          </td>
          <td>
            This is not a good example
            of a layout template...
          </td>
          <td>
            <table>
              ...etc...
            </table>
        </tr>
      </table>
    </td>
    <td>
      <table>
        ...etc...
      </table>
    </td>
    .
    .
    .
```

The sensible formatting helps to make the structure clearer through use of indenting. However, it is still difficult to match rows and cells with their corresponding tables, and there is little indication of what the different tables contribute to the overall layout.

A better approach is to build the layout using several different templates. For example, we can simplify the preceding template by moving the inner tables to separate templates:

```
<table border="0" cellpadding="0" cellspacing="0">
  <tr valign="top">
    <td>
      [% PROCESS sidebar %]
    </td>
    <td>
      [% PROCESS topmenu %]
    </td>
```

```
.

.

.
```

Now we can easily see the high-level structure without getting bogged down in the detail of the nested tables. Furthermore, by giving our templates names that reflect their purpose (e.g., sidebar and topmenu), we effectively have a self-documenting template that shows at a glance what it does. Another benefit is that the individual elements, the sidebar and topmenu in this example, will themselves be much easier to write and maintain in isolation. They also become reusable, allowing you to incorporate them into another part of the site (or perhaps another site) with a PROCESS or similar directive.

2.7.4 Layout Example

Let's work through a complete example now, applying this principle to the presentation framework for our web site. Example 2-17 shows an alternate version of the *wrapper* template that delegates the task to two further templates, *html* and *layout*.

Example 2-17. lib/wrapper2

```
[% WRAPPER html + layout;

     content;

   END

-%]
```

The two wrapper templates, *html* and *layout*, are both specified in the one WRAPPER directive, separated using the + character in the same way that we used it with the PROCESS directive in Example 2-9. In this case, the page content will be processed first, then the *layout* template, and finally the *html* template. Remember that the WRAPPER directive works "inside out" by processing the wrapped content first, and then the wrapping templates.

If we unwrap the preceding directive into two separate WRAPPER calls, it should become more obvious why the WRAPPER directive processes the templates in the *reverse* order to how they're specified:

```
[% WRAPPER html;

     WRAPPER layout;

       content;

     END;

   END

%]
```

The end result is that it does what you would expect, regardless of the slightly counterintuitive order in which it does it. The *html* template ends up wrapping the *layout* template, which in turn wraps the value of the content variable, which in this case is the output from processing the main page template.

2.7.4.1 Side-effect wrappers

The WRAPPER directive can also be used in side-effect notation. Consider the following fragment:

```
[% WRAPPER layout;

     content;

   END

%]
```

You can simplify this by writing it as follows:

```
[% content WRAPPER layout %]
```

The *wrapper* template shown in Example 2-17 can be rewritten in the same way, as shown in Example 2-18.

Example 2-18. lib/wrapper3

```
[% content WRAPPER html + layout -%]
```

2.7.4.2 Separating layout concerns

Using two separate layout templates, *html* and *layout*, allows us to make a clear separation between the different kinds of markup that we're adding to each page. The *html* template adds the <head> and <body> elements required to make each page valid HTML. The *layout* template deals with the overall presentation of the visible page content, adding a header, footer, menu, and other user interface components.

Example 2-19 shows the *html* template.

Example 2-19. lib/html

```
<html>
  <head>
    <title>[% author %]: [% template.title %]</title>
  </head>

  <body bgcolor="[% bgcol %]">
    [% content %]
  </body>
</html>
```

Example 2-20 shows the *layout* template.

Example 2-20. lib/layout

```
<table border="0" width="100%">
  <tr>
    <td colspan="2">
      [% PROCESS pagehead %]
    </td>
  </tr>
  <tr valign="top">
    <td width="150">
      [% PROCESS menu %]
    </td>
    <td>
      [% content %]
    </td>
  </tr>
  <tr>
    <td colspan="2" align="center">
```

```
    [% PROCESS pageinfo %]

   </td>

  </tr>

</table>
```

We've created a new header template, *pagehead*, shown in Example 2-21, which generates a headline for the page. It's simple for now, but we can easily change it to something more complicated at a later date.

Example 2-21. lib/pagehead

```
<h1>[% template.title %]</h1>
```

We're also using another template, *menu*, to handle the generation of a menu for the site. We'll be looking at this shortly.

Example 2-22 shows the final template used in the layout, *pageinfo*. This incorporates the copyright message and some information about the page template being processed.

Example 2-22. lib/pageinfo

```
[% USE Date %]



&copy; [% copyr %]



<br />



[% template.name -%]

last modified

[%- Date.format(template.modtime) %]
```

Notice how we're using the template.name and template.modtime variables to access the filename and modification time of the current page template. The template.modtime value is returned as a large number that means something to computers[2] but not a great deal to humans. To turn this into something more meaningful, we're using the Date plugin to format the number as a human-readable string.

[2] It's the number of seconds that have elapsed since January 1, 1970, known as the the *Unix epoch*.

2.7.4.3 Plugins and the USE directive

Plugins are a powerful feature of the Template Toolkit that allow you to load and use complex functionality in your templates, but without having to worry about any of the underlying implementation detail. Plugins are covered in detail in Chapter 6, but there's not much you need to know to start using them.

In Example 2-22, we first load the Date plugin with the USE directive:

```
[% USE Date %]
```

This creates a Date template variable that contains a reference to a plugin object (of the Template::Plugin::Date class, but you don't need to know that). We can then call the format method against the Date object using the dot operator, passing the value for template.modtime as an argument:

```
[%- Date.format(template.modtime) %]
```

The output generated would look something like this:

```
17:43:35 14-Jul-2003
```

That's all we need to do to load and use the Date plugin. Dozens of plugins are available for doing all kinds of different tasks, described in detail in Chapter 6.

2.8 Menu Components

In the *layout* template in Example 2-20, we delegate the task of generating a menu for the web site to the *menu* template. Before we look at how the template does this, let's see an example of the kind of HTML that we would like it to generate.

```
<table border="0">
  <tr>
    <td>
      <img src="/images/icon.png" width="4" height="4" />
    </td>
    <td>
      <a href="earth.html">Earth</a>
    </td>
  </tr>
  <tr>
    <td>
      <img src="/images/icon.png" width="4" height="4" />
    </td>
    <td>
      <a href="magrethea.html">Magrethea</a>
    </td>
  </tr>
</table>
```

The entire menu is defined as a <table> element, containing one <tr> row for each item, each of which holds two <td> cells, one to display an icon, the other a link to a particular page. Only two items are in this simple example, but already we can see how it gets repetitive very quickly. This suggests that we can modularize the markup into separate template components.

2.8.1 Simple Menu Template

Example 2-23 shows a *menu* template that defines the outer <table> elements and uses a second template, *menuitem*, to generate each item.

Example 2-23. lib/menu

```
<table border="0">
[%
 PROCESS menuitem
  text = 'Earth'
  link = 'earth.html';
```

```
 PROCESS menuitem
  text = 'Magrethea'
  link = 'magrethea.html';
%]
</table>

[% BLOCK menuitem %]
<tr>
  <td>
    <img src="/images/icon.png" width="4" height="4" />
  </td>
  <td>
    <a href="[% link %]">[% text %]</a>
  </td>
</tr>
[% END %]
```

2.8.1.1 The BLOCK directive

We could easily define the *menuitem* template in a separate file as we have with other components, but it would require us to split the HTML <table> markup into different files. This would make it harder to maintain and possibly lead to tag mismatch or other formatting errors.

Instead, we define the *menuitem* template inside the *menu* template using the BLOCK directive. The argument following the BLOCK keyword is a name for the template component, which can then be used in any PROCESS, INCLUDE, or WRAPPER directives. The content of the component follows, and can contain any kind of Template Toolkit directives up to the corresponding END directive.

```
[% BLOCK menuitem %]
   <tr>
     <td>
       <img src="/images/icon.png" width="4" height="4" />
     </td>
     <td>
       <a href="[% link %]">[% text %]</a>
     </td>
   </tr>
[% END %]
```

The *menuitem* template block is defined at the bottom of the *menu* template, but that doesn't stop us from using it earlier in the same template, before it is defined.

The *menuitem* block will remain defined while the *menu* template is being processed. Any other templates that are called from within the *menu* template (e.g., by a PROCESS or INCLUDE directive) will also be able to use the *menuitem* block.

2.8.2 Component Libraries

When a template is loaded using the PROCESS directive, any BLOCK definitions within it will be imported and available for use in the calling template. Templates loaded using the INCLUDE directive keep to themselves and don't export their BLOCK definitions (or any of their local variables, as described in the earlier discussion of the INCLUDE directive).

This feature allows you to create single template files that contain libraries of smaller template components, defined using the BLOCK directive. This is illustrated in Example 2-24.

Example 2-24. lib/mylib

```
[% BLOCK image -%]
  <img src="[% src %]" alt="[% alt %]"
  width="[% width %]" height="[% height %]" />
[%- END %]


[% BLOCK link -%]
  <a href="[% link %]">[% text %]</a>
[%- END %]


[% BLOCK icon;
   INCLUDE image
     src    = '/images/icon.png'
     alt    = 'dot icon'
     width  = 4
     height = 4 ;
END
-%]
```

Notice how the *icon* BLOCK definition is defined within a single directive, and consists of nothing more than a call to the *image* template component, defined earlier in the same file. This illustrates how easy it is to reuse existing components to quickly adapt them for more specific, or alternate purposes.

The BLOCK definitions can be loaded from the *mylib* template with a PROCESS directive. Then they can be used just like any other template component. Example 2-25 shows a variation of the *menu* template from Example 2-23 in which the *icon* and *link* components are used to generate the menu items.

Example 2-25. lib/menu2

```
[% PROCESS mylib %]


<table border="0">
[%
  PROCESS menuitem
    text = 'Earth'
    link = 'earth.html';


  PROCESS menuitem
```

```
    text = 'Magrethea'
    link = 'magrethea.html';
%]
</table>

[% BLOCK menuitem %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link %]
  </td>
</tr>
[% END %]
```

2.8.2.1 The EXPOSE_BLOCKS option

You can also set an option that allows you to use BLOCK directives without having to first PROCESS the template in which they're defined. The expose_blocks option for *ttree* and the corresponding EXPOSE_BLOCKS option for the Template module can be set to make this possible.

For example, by adding the following to the *etc/ttree.cfg* file:

```
expose_blocks
```

we can then access a BLOCK in the *mylib* template like so:

```
[% PROCESS mylib/icon %]
```

The template name, *mylib*, is followed by the BLOCK name, *icon*, separated by a / (slash) character. The notation is intentionally identical to how you would specify the *icon* file in the *mylib* directory. This is another example of how the Template Toolkit abstracts certain underlying implementation details so that you don't tie yourself down to one particular way of doing something.

At a later date, for example, you might decide to split the *mylib* template into separate files, stored in the *mylib* directory. The same directive will continue to work because the syntax is exactly the same for blocks in files as it is for files in directories:

```
[% PROCESS mylib/icon %]
```

This gives you more flexibility in allowing you to change the way you organize your template components, without having to worry about how that might affect the templates that use them.

2.8.3 The FOREACH Directive

The menu component from Example 2-25 can be simplified further by first defining a list of menu items and then iterating over them using the FOREACH directive. Example 2-26 demonstrates this.

Example 2-26. lib/menu3

```
[% PROCESS mylib %]

[% menu = [
    { text = 'Earth'
```

```
    link = 'earth.html' }
  { text = 'Magrethea'
    link = 'magrethea.html' }
  ]
%]

<table border="0">
[% FOREACH item IN menu %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link
         text = item.text
         link = item.link
    %]
  </td>
</tr>
[% END %]
</table>
```

The menu variable is defined as a list of hash arrays, each containing a text and link item:

```
[% menu = [
     { text = 'Earth'
       link = 'earth.html' }
     { text = 'Magrethea'
       link = 'magrethea.html' }
  ]
%]
```

The main body of the template defines an HTML <table> element. Within the table, the FOREACH directive iterates through the menu list, setting the item variable to each element in turn.

```
<table border="0">
[% FOREACH item IN menu %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link
```

```
        text = item.text
        link = item.link
    %]
  </td>
</tr>
[% END %]
</table>
```

The block following the FOREACH directive, up to the corresponding END, can contain text and other directives, even including nested FOREACH blocks. To make the code easier to read, we might prefer to define the *menuitem* BLOCK, as shown in Example 2-25. This allows us to simplify the FOREACH directive, merging it into a single tag.

```
<table border="0">
[% FOREACH item IN menu;
     PROCESS menuitem
       text = item.text
       link = item.link;
   END
%]
</table>
```

The FOREACH block now contains just one directive to PROCESS the *menuitem* component. The text and link variables are set to the item.text and item.link values, respectively.

When the items in a FOREACH list are hash arrays, as they are in Example 2-26, you can omit the name of the item variable:

```
<table border="0">
[% FOREACH menu;
     PROCESS menuitem;
   END
%]
</table>
```

In this case, the values in each hash array will be made available as local variables inside the FOREACH block. So item.text becomes the text variable, and item.link becomes link, but only within the scope of the FOREACH block. This conveniently allows us to process the *menuitem* template without needing to explicitly dereference the item variables.

There's one more improvement we can make by taking advantage of the Template Toolkit's side-effect notation. Instead of writing the PROCESS menuitem directive in the FOREACH block all by itself, we can put it *before* the FOREACH and do away with the semicolons and END keyword:

```
<table border="0">
[% PROCESS menuitem FOREACH menu %]
</table>
```

All these enhancements to the *menu* template are shown in Example 2-27.

Example 2-27. lib/menu4

```
[% PROCESS mylib %]

[% menu = [
    { text = 'Earth'
      link = 'earth.html' }
    { text = 'Magrethea'
      link = 'magrethea.html' }
   ]
%]

<table border="0">
[% PROCESS menuitem FOREACH menu %]
</table>

[% BLOCK menuitem %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link %]
  </td>
</tr>
[% END %]
```

2.9 Defining and Using Complex Data

The variables that we have used so far have mostly been simple *scalar* variables that contain just one value. The few exceptions include the tantalizing glimpses of the template variable, and the Date plugin in Example 2-22. As we saw in Chapter 1, the Template Toolkit also supports lists and hash arrays for complex data, and allows you to access Perl subroutines and objects.

In this section, we will look more closely at defining and using complex data structures, and describe the different Template Toolkit directives for inspecting, presenting, and manipulating them.

2.9.1 Structured Configuration Templates

Larger sites will typically use dozens of different global site variables to represent colors, titles, URLs, copyright messages, and various other parameters. The Template Toolkit places no restriction on the number of different variables you use, but you and your template authors may soon lose track of them if you have too many.

Another problem with having lots of global variables lying around is that you might accidentally overwrite one of them. We saw in Example 2-7 how the author variable was used to store the name of the site author, Arthur Dent, for use in the *header* and *footer* templates. At some later date, we might decide to add a *quote* template component that also uses the author variable. This is shown in Example 2-28.

Example 2-28. lib/quote

```
<blockquote>
  [% quote %]
</blockquote>


  -- [% author %]
```

There's no problem if we use INCLUDE to load the template, providing a local variable value for author:

```
[% INCLUDE quote
     author = 'Douglas Adams'
     quote  = 'I love deadlines. I like the
               whooshing sound they make as
               they fly by.'
%]
```

The value for author supplied as a parameter to the INCLUDE directive (Douglas Adams) remains set as a local variable within the *quote* template. It doesn't affect the global author variable that is defined in the config (Arthur Dent).

However, it is all too easy to forget that the author variable is "reserved"—especially if it's just one of a large number of such variables—and to use PROCESS instead of INCLUDE:

```
[% PROCESS quote
     author = 'Douglas Adams'
     quote  = 'I love deadlines. I like the
               whooshing sound they make as
               they fly by.'
%]
```

The PROCESS directive doesn't localize any variables. As a result, our global author variable now is incorrectly set to Douglas Adams instead of Arthur Dent. One solution is to religiously use INCLUDE instead of PROCESS at every opportunity. However, that's just working around the problem rather than addressing the real issue. Furthermore, the INCLUDE directive is quite a bit slower than PROCESS, and if performance is a concern for you, you should be looking to use PROCESS wherever possible.

Variables are localized for the INCLUDE directive in a part of the Template Toolkit called the *Stash*. It saves a copy of all the current variables in use before the template is processed, and then restores them to these original values when processing is complete. Understandably, this process takes a certain amount of time (not much in human terms, but still a finite amount), and the more variables you have, the longer it takes.

It is worth stressing that for most users of the Template Toolkit, these performance issues will be of no concern whatsoever. If you're using the Template Toolkit to generate static web content offline, it makes little difference if a template takes a few hundredths or thousandths of a second longer to process. Even for generating dynamic content online, performance issues such as these probably aren't going to concern you unless you have particularly complicated templates or your site is heavily loaded and continually generating lots of dynamic content.

The more important issue is one of human efficiency. We would like to make it easier for template authors to keep track of the variables in use, make it harder for them to accidentally trample on them in a template component, and ideally, allow them to use PROCESS or INCLUDE, whichever is most appropriate to the task at hand.

The answer is to use a nested data structure to define all the sitewide variables under one global variable. Example 2-29 shows how numerous configuration variables can be defined as part of the site data structure, in this case implemented using a hash array.

Example 2-29. lib/site

```
[%  site = {
    author = 'Arthur Dent'
    bgcol  = '#FF6600'    # orange
    year   = 2003
  }


  site.copyr = "Copyright $site.year $site.author"
%]
```

To interpolate the values for the year and author to generate the copyright string, we must now give them their full names, site.year and site.author. We need to set the site.copyr variable *after* the initial site data structure is defined so that we can use these variables. In effect, the site variable doesn't exist until the closing brace, so any references to it before that point will return empty values (unless the site has previously been set to contain these items at some earlier point).

```
[% site = {
     author = 'Arthur Dent'
     bgcol  = '#FF6600'    # orange
     year   = 2003

     # this doesn't work because site.year
     # and site.author are undefined at
     # this point
     copyr  = "Copyright $site.year $site.author"
   }
%]
```

Sitewide values can now be accessed through the site hash in all templates, leaving author, bgcol, year, and all the other variables (except site, of course) free to be used, modified, and updated as "temporary" variables by page templates and template components. Now there's just one variable to keep track of, so there's much less chance of accidentally overwriting an important piece of data because you forgot it was there. It also means that the INCLUDE directive works faster because it has only one variable to localize instead of many. The Stash copies only the top-level variables in the process of localizing them and doesn't drill down through any of the nested data structures it finds.

2.9.2 Layered Configuration Templates

As your site data structure becomes more complicated, you might find it easier to build it in layers using several templates. Example 2-30 shows a preprocessed configuration template that loads the *site*, *col*, and *url* templates using PROCESS.

Example 2-30. lib/configs

```
[% PROCESS site
     + col
     + url
-%]
```

We have already seen the *site* template in Example 2-29. Example 2-31 shows the *col* and *url* configuration templates.

Example 2-31. lib/col

```
[%  site.rgb = {
      white  = '#FFFFFF'
      black  = '#000000'
      orange = '#FF6600'
    }

    site.col = {
      back = site.rgb.orange
      text = site.rgb.white
    }
-%]
```

Example 2-31 shows the definition of a site.rgb hash and then another, site.col, which references values in the first. Template authors can use explicit colors, by referencing site.rgb.orange, for example, to fetch the correct RGB value, #FF6600. Or they can code their templates to use colors defined in the site.col structure—for example, referencing site.col.back in the *html* template to set the bgcolor attribute of the HTML <body> element. Either way, the colors are defined in one place, and the symbolic names allow us to see at a glance that the background color for the pages in the site is currently orange.

The *url* template is a little simpler, but also illustrates how variables can be built in stages (see Example 2-32).

Example 2-32. lib/url

```
[%  url = 'http://tt2.org/ttbook'

    site.url = {
      root   = url
      home   = "$url/index.html"
      help   = "$url/help.html"
      images = "$url/images"
    }

-%]
```

The benefits of this approach are twofold. The first is that you can save yourself a great deal of typing by replacing a long-winded URL with a shorter variable name. The second benefit is that you can easily change all the URL values in a single stroke by changing the root url from which they are constructed.

One advantage of building a complex data structure from several templates is that you can easily replace one of the templates without affecting the others. For example, you might want to use a different set of URL values at some point. Rather than edit the *url* template, you can copy the contents to a new file (e.g., *url2*), make the changes there, and then update the *configs* template accordingly:

```
[% PROCESS site

        + col

        + url2

-%]
```

If you must revert to the old URLs at a later date, you need to change only the *configs* template to load *url* instead of *url2*. You can also use this approach to load different configuration templates based on a conditional expression. For example:

```
[% PROCESS site

        + col;



  IF developing;

    PROCESS url2;

  ELSE;

    PROCESS url;

  END

-%]
```

2.9.3 Choosing Global Variables Wisely

Fewer global variables are better, but don't try to cram everything into the one site variable if more would do the job better. Try and separate your variables into structures according to their general purpose and relevance to different aspects of the site. For example, you can define one structure containing everything related to the site as a whole (e.g., site), and another related to the individual page being processed (e.g., page):

```
[% site = {

    title  = "Arthur Dent's Web Site"

    author = 'Arthur Dent'

    # ...etc...

  }



  page = {

    title  = template.title

    author = template.author or site.author

  }

%]
```

You may also want to define others to represent a user, server, application, or request depending on how you're using the Template Toolkit and what you're using it for.

The Template Toolkit allows you to use upper- or lowercase, or some combination of the two, to specify variable names. It's not recommended that you use all uppercase variable names, as they might clash with current (or future) Template Toolkit directives. However, you might like to capitalize your global variables to help you remember that they're special in some way (e.g., Site versus site):

```
[% Site = {
     # ...etc...
   }
   Page = {
     # ...etc...
   }
   User = {
     # ...etc...
   }
%]
```

2.9.4 Passing Around Data Structures

You can pass a complex data structure around the Template Toolkit as easily as you would a scalar variable. Example 2-33 shows a configuration template that defines the site.menu data structure to contain the menu items that we used earlier in Example 2-26.

Example 2-33. lib/menudef

```
[% site.menu = [
    { text = 'Earth'
      link = 'earth.html' }
    { text = 'Magrethea'
      link = 'magrethea.html' }
   ]
%]
```

We've moved the definition of the sitewide menu into a central configuration file and will need to add it to the list of templates loaded by the PROCESS directive in the pre-processed *configs* template shown in Example 2-30:

```
[% PROCESS site
        + col
        + url
        + menudef
-%]
```

Now we can remove the definition of the menu structure from the component (or components) that generate the menu in a particular style, as shown in Example 2-34.

Example 2-34. lib/menu5

```
[% PROCESS mylib %]

<table border="0">
[%- FOREACH item IN menu;
      PROCESS menuitem
        text = item.text
```

```
        link = item.link;
END
-%]
</table>

[% BLOCK menuitem %]
<tr>
  <td>
    [% PROCESS icon %]
  </td>
  <td>
    [% PROCESS link %]
  </td>
</tr>
[% END %]
```

The value for menu (site.menu in this case) is passed to the *menu5* template as an argument in an INCLUDE directive:

```
[% INCLUDE menu5
      menu = site.menu
%]
```

The benefit of this approach is that the component that generates the menu is now generic, and will work with any menu data you care to define. Wherever you need a menu in the same style, simply call the component and pass in a different definition of menu data:

```
[% INCLUDE menu5
      menu = [
        { text = 'Milliways'
          link = 'milliways.html' }
        { text = 'Hotblack Desiato'
          link = 'desiato.html' }
      ]
%]
```

Separating the definition of a menu from its presentation also makes it easier to change the menu style at a later date. There's only one generic menu component to update or replace, regardless of how many times it is used in various places around the site. If you want two or more different menu styles, simply create additional menu components with different names or in different locations. For example, you may have *site_menu* and *page_menu*, or *site/menu* and *page/menu*, or perhaps something such as *slick/graphical/menu* and *plain/text/menu*.

Chapter 2. Building a Complete Web Site Using the Template Toolkit

This chapter puts the Template Toolkit into context. We show several different ways of using the Template Toolkit to simplify the process of building and managing web site content. We start with some simple examples showing the use of template variables and template components that allow web content to be constructed in a modular fashion. As we progress further into the chapter, we look at more advanced techniques that address the issues of managing the site structure, generating menus and other navigation components, and defining and using complex data.

Although the focus of this chapter is on generating web content, it also serves as a general introduction to the Template Toolkit. It demonstrates techniques that can be adapted to different application areas. This chapter will quickly get you up to speed using the Template Toolkit, but without bogging you down in too much gory detail (we're saving that for the rest of the book). We come back to the Web to look at more advanced examples of static and dynamic web content in Chapter 11 and Chapter 12.

Although we may touch briefly on some more advanced issues, we try not to bore you with too much detail, except where it is absolutely necessary to illustrate a key point or explain an important concept. Chapter 3 discusses the syntax and structure of templates and the use of variables, while Chapter 4 covers the various template directives. More information relating to filters and plugins can be found in Chapter 5 and Chapter 6, respectively. More advanced topics concerning the use of the Template Toolkit for generating web content and interfacing to web applications can be found in Chapter 11 and Chapter 12.

We assume a Unix system in the examples in this chapter, but the principles apply equally well to other operating systems. On a Microsoft Windows machine, for example, the File Explorer can be used to create folders (directories) and shortcuts (symbolic links) using the familiar point-and-click interface. Another option we can highly recommend is to install *Cygwin*. *Cygwin* is freely available from http://www.cygwin.com and provides you with a Unix-like environment on Win32.

3.1 Template Syntax

The Template Toolkit has many configuration options to change the appearance and meaning of the directives in a template. This section looks at the different types of directives, shows how to change the directive tags, and describes the various ways you can control the processing of whitespace around directives.

3.1.1 Text and Directives

A template contains a mixture of fixed text and directive tags, denoted by the [% and %] markers. Everything coming after the [% and before the following %] is part of the directive tag. Everything else in the document is fixed text that is passed through intact.

Well, that's the default behavior, anyway. There are certain occasions when the text surrounding directives *will* be modified. For example, the whitespace *chomping* options (PRE_CHOMP and POST_CHOMP) and related flags (which we'll be looking at shortly) tell the Template Toolkit to remove any extraneous whitespace in the text on either side of (i.e., before or after) a directive. The INTERPOLATE option is another example that, when set (which it isn't by default), causes the text part of the template to be passed through a second scanning process to look for any embedded variables, denoted by a $ prefix—e.g., Hello $planet. More on that later.

You can also change the characters used to denote tags with the TAG_STYLE, START_TAG, and END_TAG configuration options, and with the TAGS directive. We'll also be looking at this shortly.

3.1.1.1 Template parser

All of this happens inside a part of the Template Toolkit called the *parser* (implemented in the Template::Parser module, and assisted by various others including Template::Grammar and Template::Directives). The job of the parser is to scan the source template to figure out which parts are text and which are directives, taking all the relevant configuration options and any values set by the TAGS directive into account. Having worked out where the directive tags are, it then parses the statements within them, checking that their syntax and structure are correct. If they aren't, the parser returns a parse error along with a short message explaining the problem.

3.1.1.2 Parse errors

We can demonstrate a parse error by having *tpage* process the template in Example 3-1, which contains an erroneous directive. The mandatory template filename after the PROCESS keyword is missing.

Example 3-1. badfile

```
[% # this is an invalid directive
   # and will raise a parse error
   PROCESS
%]
```

This is what happens when we run *tpage*:

```
$ tpage badfile
file error - parse error - badfile line 1-4:
  unexpected end of directive
  [% # this is an invalid directive
     # and will raise a parse error
     PROCESS
  %]
  at /usr/bin/tpage line 60.
```

We've edited the output a little for the sake of clarity, but all the important parts are there. The message tells us what kinds of errors occurred (in this case, a general file error and a parse error), what the error was (unexpected end of directive), and where it occurred (badfile line 1-4). It also shows the offending directive and reports the line number in the *tpage* program where the error was raised (at /usr/bin/tpage line 60).

3.1.1.3 Caching templates

If the template content is valid, the parser *compiles* it into a Perl subroutine that faithfully reproduces its exact functionality. Although the subroutine takes a little time to parse and compile the template into the equivalent Perl code, it is more than paid back by the speed at which it then runs. The great benefit of this approach is that the compiled template (i.e., the Perl subroutine) can be *cached* internally by the Template Toolkit for subsequent reuse. It keeps hold of the subroutine for each template that gets compiled so that it doesn't have to do the hard job of parsing and compiling it again the next time you want to use it.

This caching lasts for the lifetime of the Perl Template object being used. When you run *ttree* to build all the pages in a web site, for example, one Template object is used throughout. Every page can call the *menuitem* template a dozen times, for instance, but it will only be parsed and compiled the first time it is used. This is also ideal when you're using the Template Toolkit to serve dynamic pages from a persistent web server process (i.e., Apache and mod_perl). In contrast to a CGI script, which is restarted each time it is used and must create a new Template object each time, an Apache mod_perl handler can reuse a shared Template object, allowing the compiled templates to remain cached and ready to be used over and over again.

3.1.1.4 Flexible syntax

The job of parsing a template document is not an easy one. The Template Toolkit parser tries to be as flexible as possible with regard to the syntax and structure of directive tags. It doesn't complain if you forget (or choose not) to put a comma between items in a list, for example. As long as there's some kind of whitespace to separate them and the meaning isn't ambiguous, it will work around you so that you don't have to work around it.

Understandably, there are some basic rules that you'll need to follow, as well as some general guidelines that can help to make your templates easier to read and write. This section covers them in detail and shows the various ways in which the default behavior can be modified through the use of configuration options and other means.

As long as you follow the basic rules, the matter of how you lay out your directives, incorporating whitespace, formatting, and comments, is very much one of personal taste. You don't have to lay out your templates (or Perl code) nicely at all if you don't want to, but you will appreciate it when you come back to them after an absence and have to try and figure out what is going on. Anyone else who has to maintain your templates will also appreciate your efforts in making them as simple and clear as possible.

3.1.2 Template Tags

The default characters that the Template Toolkit uses to denote the position of directive tags are [% and %].

We saw an example in Chapter 2 showing how the TAGS directive can be used to set a different tag style for a single template file:

```
[% TAGS star %]

People of [* planet *], your attention please.
```

The tag style can be changed any number of times within a template and will revert to the current default at the end.

Figure 3-1 shows a list of the different tag styles available.

Figure 3-1. Tag styles

Tag style	Start tag	End tag
template	[%	%]
template1	[% *or* %%	%] *or* %%
metatext	%%	%%
html	<!--	-->
mason	<%	>
asp	<%	%>
php	<?	?>
star	[*	*]

Custom start and end tags can be set using the two-argument form of the TAGS directive:

```
[% TAGS { } %]

People of {planet}, your attention please.
```

The TAGS directive should always be specified in a tag by itself. It is something of a special case for the parser and doesn't obey the usual rule for directives of allowing a semicolon to separate one statement from the next.

```
[% TAGS star;

   # don't do this... it doesn't work

   PROCESS header

%]
```

However, you can use the whitespace chomping flags in a TAGS directive:

```
[% TAGS star -%]

[* PROCESS header -*]
```

The Template Toolkit provides the TAG_STYLE configuration option for setting a named tag style from Perl:

```
my $tt = Template->new({

    TAG_STYLE => 'star',

});
```

If you can't find an existing style you like, you can define custom start and end tags using the START_TAG and END_TAG options:

```
my $tt = Template->new({

    START_TAG => quotemeta('[*'),

    END_TAG   => quotemeta('*]'),

});
```

The START_TAG and END_TAG options support Perl regular expressions, giving you precise control over exactly what you want to match. One side effect of this is that any regular expression metacharacters (such as [and *) will need to be explicitly escaped with a \ prefix (e.g., '\[*') or passed through Perl's quotemeta function, as shown in the previous example.

The next example shows how regular expressions can be used for the START_TAG and END_TAG options:

```
my $tt = Template->new({

    START_TAG => '<(?i:tt):',

    END_TAG   => '/?>',

});
```

Here we allow the <tt: prefix to be specified in uppercase, lowercase, or mixed case (the (?i:...) part of the START_TAG regular expression), and the END_TAG to permit an optional / before the closing >. The following fragment shows four tags in slightly different styles, all of which will be matched by the START_TAG and END_TAG regular expressions:

```
<tt:pi=3.142/>

<tt:e=2.718>

pi: <TT:pi>

 e: <TT:e/>
```

The TAG_STYLE option takes priority over any values for START_TAG and END_TAG, so it makes no sense to mix them in the same configuration. Use either TAG_STYLE or START_TAG and END_TAG.

3.1.3 Interpolated Variables

The INTERPOLATE option allows you to embed variables in plain text using a simple $variable or ${variable} syntax. It is disabled by default, but can be set to any true value as a configuration option to enable this behavior.

```
my $tt = Template->new({

    INTERPOLATE => 1,

});
```

With the INTERPOLATE option enabled, the following template fragments have the same effect:

```
# using explicit directives

<a href="[% path %]">[% page.title %]</a>



# using interpolated variables

<a href="$path">$page.title</a>
```

Variable names can contain dotted elements, as shown by $page.title in the preceding example. The explicit braces can be used to delimit a variable name where necessary.

For example:

```
<img src="/images/${icon.file}.png">
```

Without the explicit scoping, the parser would treat icon.file.png as the variable name:

```
# incorrect usage

<img src="/images/$icon.file.png">
```

You must also use braces to explicitly scope embedded variables if you want to pass arguments to any of the dotted elements:

```
<img src="/images/${icon.file(extension='png')}">
```

If you've got the INTERPOLATE mode set and want to use a $ character in your document *without* it triggering a variable lookup, escape it with a \ prefix to nullify its special meaning.

For example:

```
...costing less than one

   Altairian dollar (\$1.00 ALD)

   per day...
```

The backslash tells the parser to treat the $ that follows it as just that, a literal $ character, rather than trying to interpret it as the start of a reference to a nonexistant $1.00 variable. Rather surprisingly, 1.00 is a perfectly valid variable name, given that variables can be dotted, with each part being composed of any combination of letters, numbers, or underscores. You'll have a difficult job trying to use a variable called 1.00 because the Template Toolkit will assume that you really mean the floating-point number 1.00 whenever you try and use it. Nevertheless, it's enough to confuse the parser in this case, so the preceeding \ is used to clarify our meaning.

3.1.4 Comments

Comments can be added to directives, either to provide explanations of what's going on for future maintainers (i.e., you, in six months time, when you've forgotten what you did and why you did it), or to temporarily disable all or part of the directive for testing or debugging purposes.

The # character introduces a comment in a directive. Everything from the # to the end of the current line is ignored. Here's an example that would be cryptic (at best) without the liberal use of comments that we've afforded it:

```
[%  # Calculate whether year is a leap year

    # if it's evenly divisible by 4...

    IF (year % 4) =  = 0;

        # if it is not a century year...

        IF (year % 100) =  = 0;

            is_leap = 1;    # it's a leap year

        # if it is a century year and divisible by 400 ...
```

```
    ELSIF (year % 400) =  = 0;
      is_leap = 1;    # it's a leap year
    END;
  END;
 %]
```

Comments can begin at the start of a line or part of the way through it. In either case, once you've started a comment on a line, there's no turning back. The rest of the line is a comment, and there's no character that will put you back into "uncommenting" mode.

If the # comment character immediately follows the [% start tag (or the appropriate value for the start tag if you're using something other than the default), with no intervening whitespace, the whole directive is treated as one big comment and is totally ignored. This can be used to temporarily disable an entire directive tag.

```
[%# this is broken, so disable it...
   IF skateboarding;
     kickflip(
       rotation  = 180,
       direction = 'backside'
     );
   END
%]
```

The first # character in the preceding directive temporarily disables the entire block of code. When and if we want to use it again, we can simply remove the leading comment line, or add a space between the [% and # to make it a single-line comment:

```
[% # this is working again!
   IF skateboarding;
     kickflip(
       rotation  = 180,
       direction = 'backside'
     );
   END
%]
```

There's not a lot to distinguish between these two examples, so be aware of the big difference that a single space can make.

3.1.5 Whitespace Chomping

Anything outside a directive tag is considered fixed text and is passed through unaltered. This includes all whitespace and newline characters surrounding directive tags. Directives such as SET and BLOCK that don't generate any output by themselves will leave gaps in the output document.

For example:

```
Foo
[% a = 10 %]
Bar
```

The newline following the directive is left intact, resulting in the following output:

```
Foo



Bar
```

This generally isn't a problem when you're generating HTML, which treats whitespace as (mostly) irrelevant. However, it will be of greater concern when generating plain-text documents or other formats in which whitespace is significant.

3.1.5.1 Chomping flags

The `-` chomping flag can be placed immediately after an opening directive tag (e.g., `[%` or the current value for the start tag) to have the Template Toolkit remove the newline and any other whitespace immediately preceding the directive tag. This is called *prechomping*.

Here is a trivial example to illustrate:

```
Foo

[%- 'Bar' %]

Baz
```

The template is parsed as if written:

```
Foo[% 'Bar' %]

Baz
```

and therefore generates the following output:

```
FooBar

Baz
```

As you might expect, you can also place a `-` immediately before the closing directive tag (e.g., `%]` or the current value for the end tag) to enable *postchomping*.

The following example:

```
Foo

[% 'Bar' -%]

Baz
```

is parsed as if written:

```
Foo

[% 'Bar' %]Baz
```

and generates the following output:

```
Foo

BarBaz
```

Both prechomping and postchomping flags can be set for a directive, as shown in the following example, which generates the output `FooBarBaz`:

```
Foo

[%- 'Bar' -%]

Baz
```

3.1.5.2 Chomping options

You can set the `PRE_CHOMP` and `POST_CHOMP` options to enable prechomping and postchomping as the default for all directives:

```
my $tt = Template->new({
    PRE_CHOMP  => 1,
    POST_CHOMP => 1,
});
```

With these options set, the following example:

```
Foo
[% 'Bar' %]
Baz
```

is equivalent to explicitly adding a - at the start and end of the tag:

```
Foo
[%- 'Bar' -%]
Baz
```

You can then use + in place of where the - would usually go if you want to *disable* the default prechomping or postchomping behavior on a per-directive basis. In other words, the + tells the Template Toolkit to not chomp the whitespace coming before or after a directive, regardless of the current settings of the PRE_CHOMP and POST_CHOMP options.

```
Foo
[%+ 'Bar' +%]
Baz
```

To summarize, the PRE_CHOMP and POST_CHOMP options define the default behavior, but the - and + options take priority on an individual directive basis.

The PRE_CHOMP and POST_CHOMP options also support a different style of chomping that you can enable by setting their values to 2 instead of 1. Instead of removing the whitespace entirely, it is *collapsed* into a single space.

3.1.5.3 Chomping constants

The Template::Constants module defines an exportable set of constants, CHOMP_NONE (0), CHOMP_ALL (1), and CHOMP_COLLAPSE (2), that you can use to make your code more readable. They are loaded into a Perl program when you use the Template::Constants module, providing the quoted name :chomp as an argument. The following example demonstrates this, and shows how the CHOMP_COLLAPSE constants can then be used:

```
use Template;
use Template::Constants qw( :chomp );


my $tt = Template->new({
    PRE_CHOMP  => CHOMP_COLLAPSE,
    POST_CHOMP => CHOMP_COLLAPSE,
});
```

When the following template is processed:

```
Foo
[% 'Bar' %]
Baz
```

it is parsed as if written:

```
Foo [% 'Bar' %] Baz
```

and therefore generates the following output:

```
Foo Bar Baz
```

The + flags have the same effect of protecting whitespace around a directive regardless of the PRE_CHOMP or POST_CHOMP option being set to CHOMP_ALL or CHOMP_COLLAPSE.

3.1.6 Multiple Directive Tags

When you start to use more complex directives, you may find your templates start to look a little cluttered, as Example 3-2 shows.

Example 3-2. printer1

```
[% IF title %]
  [% IF printer_friendly %]
    [% INCLUDE headers/printer_friendly %]
  [% ELSE %]
    [% INCLUDE headers/standard %]
  [% END %]
[% END %]
```

The default tag style is designed to make the directives stand out from the rest of the document. However, the [% and %] characters overwhelm the important part of this example, the content of the various directives, making the template harder to both read and write.

Fortunately, the Template Toolkit has been around long enough for people to get bored of typing [% and %] and demand a better solution. The answer is to merge the directives into one tag, using the ; (semicolon) character to delimit one directive statement from the next.

Example 3-3 demonstrates this, showing how much simpler Example 3-2 can be written.

Example 3-3. printer2

```
[% IF title;
  IF printer_friendly;
    INCLUDE headers/printer_friendly;
  ELSE;
    INCLUDE headers/standard;
  END;
END
%]
```

When you merge directives together, you lose any whitespace that might previously have been nestling between the directives. That may be what you want. If it isn't, you can easily add it back where you need it by adding literal strings, including any text and whitespace required, as part of the directive block. This is shown in Example 3-4.

Example 3-4. person1

```
[% FOREACH person IN company.employees;
  "* ";
  person.name;
  "\n  ";
  person.email;
  "\n\n";
END
```

```
END
%]
```

With a "double-quoted" string, the \n sequence introduces a newline character. So given the following definition for company:

```
[% company = {
     employees = [
       { name = 'Tom' email = 'tom@tt2.org' },
       { name = 'Dick' email = 'dick@tt2.org' },
       { name = 'Larry' email = 'larry@tt2.org' },
     ]
   }
%]
```

the output generated by Example 3-4 would be:

```
* Tom
  tom@tt2.org


* Dick
  dick@tt2.org


* Larry
  larry@tt2.org
```

3.1.7 Side-Effect Notation

The IF, UNLESS, FOREACH, WHILE, WRAPPER, and FILTER directives expect a template block to follow them, up to the relevant END directive (or ELSIF or ELSE in the case of IF and UNLESS). They can also be used in a "side-effect" notation. This is a concept borrowed from Perl in which looping or conditional logic can be placed *after* the statement that it controls. Here is an example:

```
[% PROCESS config IF something %]
```

The equivalent code, writing the directive in full, would look like this:

```
[% IF something;
     PROCESS config;
   END
%]
```

It works only when you've got one variable, directive, or piece of text that you want to use in the block. This isn't the case in Example 3-4, which we looked at in the previous section. However, Example 3-5 shows how it can be rewritten to define the block as one double-quoted string, using variable interpolation to insert the values for person.name and person.email in the right place.

Example 3-5. person2

```
[% FOREACH person IN company.employees;
  "* $person.name\n  $person.email\n\n";
END
%]
```

With a single string as the content for the block, FOREACH can now be used in side-effect notation, as shown in Example

3-6.

Example 3-6. person3

```
[% "* $person.name\n  $person.email\n\n"
  FOREACH person IN company.employees
%]
```

More complex content can be moved into a separate template file or BLOCK definition that is then called using a single PROCESS or INCLUDE directive, as shown in Example 3-7.

Example 3-7. person4

```
[% PROCESS info
  FOREACH person IN company.employees
%]

[% BLOCK info %]
  * [% person.name %]
  [% person.email %]
[% END %]
```

3.1.8 Capturing Directive Output

The output of a directive can be captured by assigning it to a variable. The following example shows this in action:

```
[% headtext = PROCESS header
              title = "Hello World"
%]
```

In the next example, it is used to capture the output of a side-effect block:

```
[% people = PROCESS userinfo
          FOREACH user = userlist
%]
```

It can also be used in conjunction with the BLOCK directive for defining large blocks of text or other content:

```
[% quote = BLOCK %]
   'Where,' said Ford Prefect quietly,
   'does it say teleport?'

   'Well, just over here in fact,'
   said Arthur, pointing at a dark
   control box in the rear of the cabin.

   'Just under the word "emergency",
   above the word "system" and beside
   the sign saying "out of order".'
```

```
[% END %]
```

Note one important caveat of using this syntax in conjunction with side-effect notation. The following directive does not behave as might be expected:

```
[% # WRONG
  description =  'Mostly Harmless'
    IF planet =  = 'Earth'
%]
```

Our intention is to set the description variable (using the single equals assignment operator, =) to the value Mostly Harmless if the planet variable contains the value Earth (tested using the double equals comparison operator, = =):

```
[% # RIGHT
  IF planet =  = 'Earth';
     description = 'Mostly Harmless';
  END
%]
```

Unfortunately, that's not how the Template Toolkit parser sees things. The directive is interpreted as if written:

```
[% # WRONG
  description = BLOCK;
    IF planet =  = 'Earth';
      'Mostly Harmless';
    END;
  END
%]
```

The variable is assigned the output of the IF block. This returns Mostly Harmless correctly for planet Earth, but nothing in all other cases, resulting in the description variable being unintentionally cleared.

To achieve the expected behavior, the directive should use the SET keyword explicitly:

```
[% # RIGHT
  SET description =  'Mostly Harmless'
        IF planet =  = 'Earth'
%]
```

3.1.9 Template Filenames

Like Perl, the Template Toolkit treats data differently depending on whether it is quoted. For example, foo.bar accesses the value in a variable, but 'foo.bar' is a literal string.

The INSERT, INCLUDE, PROCESS, and WRAPPER directives expect a filename to be pro vided as the first argument:

```
[% PROCESS header %]
```

You can use single or double quotes around the filename, but they're generally not required:

```
[% PROCESS 'header' %]

[% PROCESS "header" %]
```

The Template Toolkit assumes that the first argument is a filename, even if it includes dot characters:

```
[% PROCESS header.tt %]
```

If you do use double quotes around the string, any variable references within it will be interpolated. For example:

```
[% file = 'header'

   ext  = 'tt'

%]

[% PROCESS "${file}.$ext" %]   # header.tt
```

You'll also need to explicitly quote the filename if it contains any characters other than alphanumerics, underscores, dots, and slashes:

```
[% PROCESS no/need_2_quote/this.txt %]

[% PROCESS 'My Documents/q&a.txt' %]
```

If you want to use a variable value to denote the name of a file, you can interpolate it into a double-quote string:

```
[% file = 'header' %]

[% PROCESS "$file" %]    # header
```

As a convenience, you can do away with the double quotes and simply use the $ prefix to tell the parser that a variable name follows:

```
[% PROCESS $file %]      # header
```

3.2 Template Variables

The Template Toolkit's simple-to-access variables are one of its strengths. In this section, we describe the syntax and semantics of variables—what names are allowed, the different types of data that can be stored in a variable, the predefined Template Toolkit variables, and so on.

3.2.1 What's in a Name?

Variable names can contain alphanumeric characters or underscores. They can be lowercase, uppercase, or mixed case, although the usual convention is to use lowercase to avoid confusion with uppercase directives. The case *is* significant, however, so foo, Foo, and FOO are all different variables. Here are some examples of valid variable names:

```
foo
foo123
foo_bar
foo_bar_123
FooBar123
Foo_Bar_123
```

The kind of data you can store in a variable depends on its *type*. The Template Toolkit is written in Perl and provides template authors with access to the full range of underlying Perl variable types. Although there are different variable types for different purposes, you can change a Template Toolkit variable from one to the other at any time. Both Perl and the Template Toolkit are examples of *dynamic languages* that don't require the type of variable to be set in stone.

The basic data types are *scalars*, which store a single value, *arrays* (or *lists*), which store multiple values in order, and *hash arrays* (or *hashes*), which store multiple values indexed by a name. In addition to these *static* data types, the Template Toolkit provides *dynamic* data types that can reference Perl *subroutines*, and *objects* that can implement any kind of functionality you require to fetch or compute a variable value on demand.

Unlike Perl, the Template Toolkit does not require you to use a different leading character, or *sigil*, on a variable name to denote its type—e.g., $item, @list, %hash. In fact, it requires you not to do it. The only time you ever use a leading $ on a variable in a template is to tell the parser that a variable for interpolation follows where it otherwise wouldn't be expecting one—for example, in a double-quoted string such as "Hello $planet", or following a directive keyword that usually expects a filename, such as [% PROCESS $myfile %].

The $ prefix should *always* be used for variable interpolation, *regardless* of the underlying data type. For example, the string "$msg.greeting $planet.0" shows how $ is used to access a hypothetical hash value, msg.greeting, and also a list item, planet.0. In both cases, $ is used as the prefix.

3.2.2 Simple Data Types

The simplest variables are scalars that hold just one value:

```
[% answer = 42 %]
[% author = 'Douglas Adams' %]
```

The values are referenced in a template by embedding the variable name in a tag:

```
The answer to the Ultimate Question of Life, the
Universe and Everything is [% answer %].

                    -- [% author %]
```

The optional SET and GET directive keywords can be used when defining and subsequently retrieving variable values:

```
[% SET author = 'Douglas Adams' %]
[% GET author %]
```

However, you'll rarely see the GET and SET keywords used because the Template Toolkit allows you to omit them. The common use is to update and access variables directly, as shown here:

```
[% author = 'Douglas Adams' %]

[% author %]
```

Scalar variables can contain numbers or text strings that both the Template Toolkit and Perl treat as interchangeable. Strings are automatically converted into numbers and numbers into strings whenever one or the other is required.

```
The answer to the Ultimate Question of Life, the

Universe and Everything is 42.



                     -- Douglas Adams
```

You can set any number of variables in the same directive:

```
[% answer = 42

   author = 'Douglas Adams'

%]
```

You don't need a semicolon between each item in a SET list, but you will need one after the last item if other directives follow. Semicolons are always required to separate GET directives in the same tag:

```
[% answer = 42               # implicit SET..

   author = 'Douglas Adams';   #    ...continued

   answer;                   # implicit GET

   author;                   # implicit GET

%]
```

Numbers can be specified as integers or in floating-point format:

```
[% answer = 42

   pi     = 3.14

%]
```

String values can be enclosed in single quotes or double quotes and can span several lines:

```
[% author = 'Douglas Adams'

   book   = "The Hitch Hiker's Guide to the Galaxy"

   advice = "Don't Panic"

   about  = "On thursday lunchtime the Earth gets

            unexpectedly demolished to make way

            for a new hyperspace bypass..."

%]
```

Using single or double quotes can be a matter of convenience, such as in this example in which the values for book and advice contain apostrophes that would otherwise be mistaken for the closing single-quote character. However, the main reason for choosing double quotes over single quotes is to allow variable values to be embedded in the string.

In single quotes, the $ character is treated as a literal and has no special meaning:

```
[% price = '$4.20' %]
```

In double quotes, on the other hand, the $ is used to mark the start of a variable name:

```
[% summary = "$book by $author" %]

Summary: [% summary %]
```

The values of the $book and $author variables will be *interpolated* into the relevant places in the string:

```
Summary: The Hitch Hiker's Guide to the Galaxy by Douglas Adams
```

You can also embed dotted variables in double-quoted strings:

```
[% summary = "$book.title by $book.author" %]
```

The ${...} delimiters can be used to explicitly scope a variable name. You'll need this whenever you have a variable nestling up tight against a dot (.) or other characters that could be mistaken for part of the name.

```
[% webpage = "h2hg/chapter_${chapter.number}.html" %]
```

Watch out in particular for periods used to mark the end of a sentence. Without the ${ and } in place to scope the your.name variable in the next example, the template fails to compile and raises a parse error:

```
[% greeting = "Hello ${your.name}." %]   # GOOD

[% greeting = "Hello $your.name." %]     # BAD - parse error!
```

If you want to include a literal $ character in a double-quoted string, precede it with a \ (backslash) character to *escape* it from any special meaning:

```
[% language = 'Perl'

   pledge   = "Will hack $language for \$\$\$"

%]

I pledge: [% pledge %]
```

The backslash characters are removed, leaving the dollar signs ringing:

```
I pledge: Will hack Perl for $$$
```

You can also use the backslash character to escape any occurrences in the string of the quote character you're using, ' or ":

```
[% advice  = 'Don\'t Panic'

   suggest = "Read \"$book\" by ${author}."

%]

1) [% advice  %]

2) [% suggest %]
```

This is the output generated:

```
1) Don't Panic

2) Read "The Hitch Hiker's Guide to the Galaxy" by Douglas Adams.
```

One final use of the backslash is to embed special metacharacters in a double-quoted string. For example, the \n sequence indicates a newline, \r a carriage return, and \t a tab character:

```
[% blockquote = "$advice\n\t-- $author" %]
```

When the value of blockquote is displayed, a newline and tab character are printed in the correct place:

```
Don't Panic

        -- Douglas Adams
```

If you want a literal backslash character in either a single- or double-quote string, you'll need to escape it with another backslash:

```
[% dospath = "C:\\dos\\path" %]
```

It's ugly, but it works. The backslash is a relatively uncommon character (except in DOS filenames, as in this example), so it's not something you normally need to worry about.

3.2.3 Complex Data Types

In contrast to simple data types that hold only a single value, the Template Toolkit supports two complex data types for storing multiple values: the *list* and *hash*. A list is an ordered array of other variables, indexed numerically and starting at element 0. A hash is an unordered collection of other variables that are indexed and accessible by a unique name or *key*.

If you're using the Template Toolkit from Perl, you can define template variables that reference any existing hash and array data structures in your Perl program that you want to make accessible in the templates:

```
my $vars = {
    primes => [ 2, 3, 5, 7, 11, 13 ],
    terms  => {
        sass  => 'know, be aware of, meet, have sex with',
        hoopy => 'really together guy',
        frood => 'really, amazingly together guy',
    },
};

$tt->process($input, $vars)
    || die $tt->error( );
```

List and hash data structures can also be defined within templates using a syntax similar to the Perl equivalents shown earlier. The default syntax is actually a little simpler than in Perl, allowing = to be used in place of => and treating commas between items as optional. However, the Template Toolkit is also comfortable with data structures laid out "Perl-style" using => and commas. This is particularly useful if you're coming from a Perl background or trying to merge existing Perl data definitions into template code, or vice versa.

Let's look at the syntax for lists and hashes in more detail.

3.2.3.1 Lists

A list variable is defined in a template using the [...] construct. Individual elements can be separated with whitespace, commas, or any combination of the two. The following all create equivalent lists:

```
[% primes = [2,3,5,7,11,13] %]
[% primes = [ 2 3 5 7 11 13 ] %]
[% primes = [ 2, 3, 5, 7, 11, 13 ] %]
[% primes = [ 2, 3 5, 7 11, 13 ] %]
```

The elements can be literal number or string values, or can reference other variables:

```
[% two    = 2
   three  = 3
   primes = [ two, three, 5, 7, 11, 13 ]
%]
```

You can also use the .. operator to create a range of values. Whitespace is optional on either side of it.

```
[% one_to_four = [ 1..4 ] %]
```

The values in a range can also be specified using variables:

```
[% start = 1
   end   = 4
   items = [ start .. end ]
%]
```

List elements are accessed using the *dot operator*. The list name is followed by the . character and then the element number.

```
[% primes.0 %]     # 2
[% primes.3 %]     # 7
```

Like Perl, the first element of a list is element 0, not element 1, meaning that primes.3 is the *fourth* element in the list, not the third. If this is confusing, it might help if you think of this number as an offset from the beginning of the list, rather than as the element number.

3.2.3.2 Hashes

A hash variable is defined in a template using the {...} construct:

```
[%  terms = {

       sass  = 'know, be aware of, meet, have sex with'

       hoopy = 'really together guy'

       frood = 'really, amazingly together guy'

   }

%]
```

Each entry in a hash is composed of a pair of values. The first is the key through which the second, the value, will be indexed in the hash. You can use either = or => to separate the key from the value. As with lists, commas can be used to delimit each pair but are not required.

```
[%  terms = {

       sass  => 'know, be aware of, meet, have sex with',

       hoopy => 'really together guy',

       frood => 'really, amazingly together guy',

   }

%]
```

Hash items are also accessed using the dot operator. In this case, the key for the required item is specified after the . character:

```
[% terms.hoopy %]    # really together guy
```

If you assign a value to an element in a hash that doesn't yet exist, it will *autovivify* the parent hash and any intermediate hashes so that the variable just springs into life when you first use it:

```
[% foo.bar.baz = 'hello world' %]
```

In this example, the foo hash and nested bar hash will be created automatically (assuming they didn't already exist), and bar will contain a baz item assigned the value hello world.

3.2.4 Dot Operator

We've already seen some simple examples of using the dot operator to access elements of complex variables. In the case of a list, an integer follows the dot operator to reference a particular item in the list. Remember that lists start counting their elements at 0, not 1, so the following directive fetches the *fourth* item in the primes list—in this case, the number 7:

```
[% primes = [2, 3, 5, 7, 11, 13] %]

[% primes.3 %]  # 7
```

For hash arrays, the dot operator is followed by the key for the item required:

```
[% terms = {

     sass  = 'know, be aware of, meet, have sex with'

     hoopy = 'really together guy'

     frood = 'really, amazingly together guy'

  }

%]

[% terms.hoopy %]
```

3.2.4.1 Compound dot operations

A variable reference can include many dot operators chained together to access data nested deeply in a complex data structure.

Here's an example of some nested data:

```
[% arthur = {
    name    = 'Arthur Dent',
    planet  = 'Earth',
    friends = {
      ford  = {
        name = 'Ford Prefect'
        home = 'Betelgeuse'
      }
      slarti = {
        name = 'Slartibartfast'
        home = 'Magrethea'
      }
    }
  }
%]
```

The following compound variables access different parts of the data structure, returning the values shown as comments to the right:

```
[% arthur.friends.ford.name %]   # Ford Prefect
[% arthur.friends.slarti.home %] # Magrethea
```

3.2.4.2 Interpolated variables names

The Template Toolkit uses the $ character to indicate that a variable should be interpolated in position. Most frequently, you see this in double-quoted strings:

```
[% fullname = "$honorific $firstname $surname" %]
```

or embedded in plain text when the INTERPOLATE option is set:

```
Dear $honorific $firstname $surname,
```

The same rules apply within directives. If a variable or part of a variable is prefixed with a $, it is replaced with its value before being used. The most common use is to retrieve an element from a hash where the key is stored in a variable.

We saw an example of this in Chapter 2:

```
[%  terms = {
       sass  = 'know, be aware of, meet, have sex with'
       hoopy = 'really together guy'
       frood = 'really, amazingly together guy'
    }
%]
[% key = 'frood' %]
[% terms.$key %]     # really, amazingly together guy
```

The value for key is interpolated into the terms.$key expression, resulting in the correct value being displayed for terms.frood.

Curly braces can be used to delimit interpolated variable names where necessary. For example:

```
[% ford = {

     name = 'Ford Prefect'

     type = 'frood'

  }

%]


[% ford.name %] is a [% terms.${ford.type} %]
```

3.2.4.3 Private variables

In Perl, it is common practice to use a leading underscore before the names of variables in an object hash to indicate those that should be considered "private" and not for use outside of the object methods. The Template Toolkit honors this and will not return any item from a hash array or object whose name begins with _ or . (which could be confused with the dot operator).

```
[% stuff = {

  _private = "You won't see me"

   public  = "You will see me"

  }

%]

[% stuff.public   %] # You will see me

[% stuff._private %] # [nothing]
```

Any attempts to retrieve these values, even indirectly by use of a variable key, will return the empty string, indicated in these examples as [nothing]:

```
[% var = "_private";

  stuff.$var       # [nothing]

%]
```

3.2.5 Dynamic Data Types

The common feature of scalars, lists, and hash arrays is that they contain *static* values. What this means in the context of template processing is that they contain pre-defined values that don't change from one minute to the next *unless* you specifically update the variable. In other words, the value is "there for the taking" once set, and can be inserted directly into a template without requiring any additional computation.

A *dynamic* value, on the other hand, is one that is computed *each* time it is used. The Template Toolkit allows template variables to be bound to Perl subroutines and objects. When the variable is accessed, the subroutine or appropriate object method is called and can perform whatever operation or calculation is required to return a value. The value returned can be different each time and may depend on any number of different factors. Hence the name *dynamic*.

Static and dynamic variables are accessed using exactly the same dotted notation. You don't need to change your templates if you decide to one day switch from using a static hash array to a dynamic subroutine that fetches some data from a database and returns a generated hash, for example. These are the kinds of implementation details that the Template Toolkit hides from you so that your templates can remain simple and portable.

Using dynamic variables when calling the Template Toolkit from Perl is as simple as passing references to subroutines or objects:

```
use CGI;


my $vars = {

   prime_number => sub {
```

```
        # return a random prime number from first 6
        my @primes = (2, 3, 5, 7, 11, 13);
        return $primes[ rand @primes ];
    },

    cgi => CGI->new( ),
};

$tt->process($input, $vars)
    || die $tt->error( );
```

There is no way to define new subroutines or objects directly in a template without resorting to embedding Perl code using the PERL or RAWPERL directives (and enabling the EVAL_PERL option, of course). However, the Template Toolkit plugin architecture allows you to define plugins that *can* be loaded directly into a template to define new subroutine and object variables. This will be covered in detail in Chapter 6.

3.2.5.1 Subroutines

The subroutine bound to a template variable will be invoked each time the value is required, in a GET directive, for example, or perhaps for interpolating into a string:

```
[% prime %]                      # calls subroutine
[% more = "$prime $prime $prime" %]  # three calls
```

The subroutine returns a value for the template variable, in this case returning a random choice of one of the first six prime numbers. Each time the variable is used, the subroutine is called and a different value returned. Of course, the nature of random numbers is such that the same value could actually be returned any number of times in the example. However, the important fact is that the value is computed each time, and any similarity between the values returned for any particular invocations is coincidental.

3.2.5.2 Objects

A variable can also be bound to a Perl object whose methods can be invoked using the same dotted notation as for accessing elements in a hash array:

```
This CGI script is running on [% cgi.server_name %]
```

The use of identical syntax for accessing hash items and object methods is an intentional and powerful feature of the Template Toolkit language. The *Uniform Access Principle* hides the implementation details behind an abstract notation that effectively "does the right thing" for whatever kind of data you're using. It provides a *clear separation of concerns* between the *representation* and *presentation* of the data, allowing one to change without affecting the other.

3.2.5.3 Passing arguments

Arguments can be passed to subroutines or object methods called from a template by adding them in parentheses immediately after the variable name. The following example shows how the literal string value docid is passed to the param( ) method of the CGI object bound to the cgi variable:

```
[% cgi.param('docid') %]
```

Here's an example of a subroutine that takes a list of arguments and returns them joined together in a single string, delimited by a comma and space:

```
my $vars = {
    join => sub {
        return join(', ', @_);
```

```
    },
};

$tt->process($input, $vars)
    || die $tt->error( );
```

Any number of arguments can be passed to the subroutine, either as numbers, as literal strings, or by referencing other variables. This is shown in Example 3-8.

Example 3-8. join

```
[% ten    = 10
   thirty = 30;
   join(ten, 20, thirty, '40')
%]
```

The output generated by Example 3-8 is as follows:

```
10, 20, 30, 40
```

3.2.5.4 Pointless arguments

Strictly speaking, you can pass arguments to any template variable, even if the variables aren't defined as references to subroutines or objects:

```
[% arthur = {
     name    = 'Arthur Dent',
     planet  = {
        name = 'Earth',
        info = 'Mostly Harmless'
     }
   }
%]

[% arthur(6).planet(7).name(42) %]  # Earth
```

In this example, the data structure is entirely static. There are no subroutines or objects lurking around that might make use of the arguments, so they are silently ignored. However, it illustrates the basic principle that any variable component can be provided with parenthesized parameters.

Providing arguments for variables that ignore them is not entirely pointless. When you're designing the look and feel of a web site, for example, you can define some simple, static data to use as "dummy" values for the page content. If you plan to implement some of these data items using subroutines or objects, you can go ahead and add any relevant parameters now so that you don't have to update your templates when the data model changes.

3.2.5.5 Named parameters

Named parameters can also be passed to subroutines and object methods. These are automatically collated into a hash reference and passed as the *last* argument to the subroutine or method.

```
my $vars = {
    join => sub {
        # look for hash ref as last argument
```

```
        my $params = ref $_[-1] eq 'HASH' ? pop : { };

        my $joint  = $params->{ joint };

        $joint = ', ' unless defined $joint;

        return join($joint, @_);

    },

};


$tt->process($input, $vars)

    || die $tt->error( );
```

Example 3-9 shows a named parameter, joint, provided in addition to the positional arguments, ten, 20, thirty, and '40'.

Example 3-9. joint

```
[% ten    = 10

   thirty = 30;

   join(ten, 20, thirty, '40', joint = '+')

%]
```

The output generated by Example 3-9 is as follows:

```
10+20+30+40
```

Named parameters can be specified anywhere in the argument list:

```
[% join(joint='+', ten, 20, thirty, '40') %]

[% join(ten, 20, joint='+', thirty, '40') %]
```

They are automatically removed from the list of positional arguments and passed to the subroutine or object method as the last argument, bound together in a single hash array reference. For this reason, and for the sake of clarity, we recommend that you always specify named parameters at the end of the list:

```
[% join(ten, 20, thirty, '40', joint='+') %]
```

In all these examples, the subroutine bound to the join variable would be called with the following list of arguments:

```
(10, 20, 30, 40, { joint => '+' })
```

In this subroutine, we look to see whether the last argument is a reference to a hash array. If it is, we pop it from the list. Otherwise, we create an empty Perl hash reference for $params.

```
# look for hash ref as last argument

my $params = ref $_[-1] eq 'HASH' ? pop : { };
```

We then look for the joint item in the named parameter hash and provide a sensible default if it isn't defined:

```
my $joint  = $params->{ joint };

$joint = ', ' unless defined $joint;
```

The subroutine calls Perl's join function, passing the $joint value along with the rest of the argument list. The resulting string is then returned:

```
return join($joint, @_);
```

Arguments can be passed to any variable, even those that are set to static values and have no use for an argument. In this case, they are simply ignored. As such, the following code:

```
[% meaning_of_life = 42 %]

[% meaning_of_life("Monday") %]
```

produces:

```
42
```

The argument "Monday" is ignored when the value of meaning_of_life is evaluated. The static value, 42, is simply inserted in its place.

3.2.5.6 Mixing dynamic and static data

Static and dynamic data structures can be freely intermixed. Static lists and hash arrays can contain references to dynamic subroutines and object methods. These can return complex data structures, including any combination of scalars, hash arrays, lists, subroutines, and object references.

```
my $vars = {
    zero => sub {
        return {
            one => sub {
                return [ $obj1, $obj2, $obj3 ],
            },
        };
    },
};
```

Compound dot operations work with dynamic data items exactly as they do for static ones. A series of dot operations can be chained together into a single expression to fetch an item from deep within a data structure, some or all of which might be computed on demand.

```
[% zero.one.2.three %]
```

In this example, zero is bound to a subroutine that returns a reference to a hash array. This contains another subroutine, one, which returns a list of objects. We take the third object, $obj3 (yes, the third, don't forget they start at 0), and call the three( ) method against it. Other than knowing that one returns a list (and so requires an index number—e.g., 2) and the others are hashes or objects (requiring index keys—e.g., one and three), we can remain blissfully ignorant of any of the underlying implementation details.

Furthermore, there's nothing to stop you from changing the one subroutine to return a hash array (or object) that contains the items (or methods) 0, 1, 2, and so on:

```
my $vars = {
    zero => sub {
        return {
            one => sub {
                return {
                    0 => $obj1,
                    1 => $obj2,
                    2 => $obj3,
                },
            },
        };
    },
};
```

It probably isn't something that you would want to do that often, but it does illustrate the point that all data types are equal as far as the dot operator is concerned. The following fragment continues to work unmodified, with 2 now being treated as a hash key instead of a list index:

```
[% zero.one.2.three %]
```

3.2.5.7 Returning values

A subroutine or object method can return any kind of value when called. Hash arrays and lists should be returned using references rather than a list of multiple items.

```
my $vars = {
    moregood => sub {
        return [ 3.14, 2.718 ];
    },
    lessgood => sub {
        return ( 3.14, 2.718 );
    },
};
```

If your subroutine does return multiple values, the Template Toolkit will automatically combine them into a list reference. This isn't the recommended usage, but it provides some level of support for existing Perl code that wasn't written with the Template Toolkit in mind.

```
# both work as expected
[% moregood.0 %] [% moregood.1 %]
[% lessgood.0 %] [% lessgood.1  %]
```

If you're writing new subroutines and methods from scratch, we suggest that you return a reference to a list rather than a list of items whenever possible. Be warned that if you do return a list of items, the first of which is undefined, the Template Toolkit will assume an error has occurred and raise it as such:

```
return (undef, ...);   # NOT OK: undef indicates error!
```

If you want to return a list of items that contains an undefined value as the first element, you should always return it as a reference to a list:

```
return [undef, ...];   # OK, returns list reference
```

3.2.5.8 Error handling

Errors can be reported from subroutines and object methods by calling die( ). This example shows a subroutine that dies as soon as it is called:

```
my $vars = {
    barf => sub {
        die "a sick error has occurred\n";
    },
};
```

If we process a template containing a reference to the barf variable, like so:

```
I think I'm going to [% barf %]
```

the Template process( ) method will return a false value and the error( ) method will report:

```
undef error - a sick error has occurred
```

Errors raised by calling die are caught by the Template Toolkit and converted to a Template::Exception object that includes the error message (a sick error has occurred) and an error type (undef). To throw an exception of a type other than the default undef, Perl code should die( ) with a reference to a Template::Exception object.

```
use Template::Exception;


my $vars = {
    barf => sub {
        die Template::Exception->new( sick => 'feel ill' );
```

```
    },
};
```

Now when the variable is accessed and the subroutine invoked, the error reported will be:

```
sick error - feel ill
```

Exceptions can be caught within templates using the TRY / CATCH directive construct:

```
[% TRY;
     barf;
   CATCH sick;
     "Eeew!  We just caught a sick error ($error.info)";
   END
%]
```

In this example, the sick error will be caught by the CATCH block, generating the following output:

```
Eeew!  We just caught a sick error (feel ill)
```

In this case, the process( ) method will return a true value. The error has been caught and dealt with, and as far as we're concerned, the template was processed successfully. Any exceptions of other types will still be passed through unless we add other CATCH blocks to catch them. This ensures that anything besides a sick exception will not be caught here.

The exception types 'stop' and 'return' are used to implement the STOP and RETURN directives. Throwing an exception as:

```
die (Template::Exception->new('stop'));
```

has the same effect as the directive:

```
[% STOP %]
```

See Chapter 4 for further information on error handling and flow control directives.

3.2.6 Special Variables

The Template Toolkit defines a number of special variables. Some, such as template and component, are universally defined and can be accessed from anywhere. Others, such as loop and content, are available only in a particular context, such as inside a FOREACH block (loop) and in a template loaded into another using the WRAPPER directive (content).

There's nothing to stop you from creating your own variables with the same name. In that case, they will simply mask the special variables provided by the Template Toolkit. However, if you define your own variable called loop, for example, it will be masked by the special variable provided in a FOREACH loop. However, the original value for your loop variable will be restored at the end of the FOREACH block.

The special variables defined by the Template Toolkit are covered in the sections that follow.

3.2.6.1 template

The template variable contains a reference to the main template being processed. It is implemented as a Template::Document object, described in detail in Chapter 8. The template variable is correctly defined within templates that are processed via the PRE_PROCESS, PROCESS, WRAPPER, and POST_PROCESS configuration options. This allows standard headers, footers, and other user interface templates to access metadata about the main page template being processed, even before it is processed.

The name and modtime metadata items are automatically defined, providing the template name and modification time in seconds since January 1, 1970 (the Unix *Epoch*), respectively. Any other items defined in META tags in the template will also be available via the appropriately named method.

For example, if the main page template defines the following:

```
[% META title  = 'My Test Page'
        author = 'Arthur Dent'
%]
```

a *header* template, defined as a PRE_PROCESS option, can access the template.title and template.author variables:

```
<html>
  <head>
    <title>[% template.title %]</title>
  </head>

  <body>
   <h1>[% template.title %]</h1>
   <h2>by [% template.author %]</h2>
```

Note that the template variable always references the main page template, regardless of any additional template components that may be processed.

3.2.6.2 component

The component variable is like template but always contains a reference to the current template component being processed.

This example demonstrates the difference:

```
$tt->process('foo')
   || die $tt->error( ), "\n";
```

A F<foo> template:

```
[% template.name %]           # foo
[% component.name %]          # foo
[% PROCESS footer %]
```

A F<footer> template:

```
[% template.name %]           # foo
[% component.name %]          # footer
```

In the main page template, *foo*, the template and component variables both reference the same Template::Document object, returning a value of foo for both template.name and component.name. In the *footer* template, the template variable remains unchanged, but the component now references the Template::Document object for the *footer* and returns the value of footer for component.footer accordingly.

3.2.6.3 loop

Inside the block of a FOREACH directive, the loop variable references a special object called an *iterator*, which is responsible for controlling and monitoring the execution of the loop. The following example shows it in use:

```
[% FOREACH item IN items %]
   [% IF loop.first %]
     <ul>
   [% END %]
      <li>[% item %] ([% loop.count %] of [% loop.size %])</li>
   [% IF loop.last %]
     </ul>
   [% END %]
[% END %]
```

The loop variable is implemented by a Template::Iterator object. It provides methods such as first and last, shown in the previous example, which return true only on the first and last iteration of the loop. The count method returns the current iteration count, starting at one (use index to get the real index number, starting at zero). The size method returns the size of the list.

The loop iterator is covered in detail in the discussion of the FOREACH directive in Chapter 4.

3.2.6.4 error

The Template Toolkit provides the TRY...CATCH construct to allow you to catch (and throw) runtime errors in your templates. Within a CATCH block, the error variable contains a reference to the Template::Exception object thrown from within the TRY block. The type and info methods can be called against it to determine what kind of error occurred and what (hopefully) informative error message was reported.

```
[% TRY %]
   ...some template code that
      may throw an error...
[% CATCH %]
   An error occurred:
      [% error.type %] - [% error.info %]
[% END %]
```

For convenience, the error variable can be referenced by itself and it will automatically be presented as a string of the form $type error - $info:

```
[% TRY;
     THROW food 'cheese roll';
   CATCH;
     error;    # food error - cheese roll
   END
%]
```

The TRY, CATCH, and other related directives are covered in detail in Chapter 4. For further information about the Template::Exception object, see Chapter 8 and the Template::Exception manpage.

3.2.6.5 content

The content variable is used by the WRAPPER directive to pass the output generated by processing the WRAPPER content block to the wrapping template. Example 3-10 shows it in action.

Example 3-10. content

```
[% scared = 'afeared'
  beats  = 'noises'
  vibes  = 'sweet airs'
  chill  = 'give delight'
-%]

[% WRAPPER box border=1 %]
  Be not [% scared %]; the isle is full of [% beats %],
  Sounds and [% vibes %], that [% chill %] and hurt not.
```

```
[% END -%]


[% BLOCK box -%]
<table border="[% border %]">
  <tr>
    <td>
      [%- content -%]
    </td>
  </tr>
</table>
[% END -%]
```

In the first section, we define some simple variables:

```
[% scared = 'afeared'
   beats  = 'noises'
   vibes  = 'sweet airs'
   chill  = 'give delight'
-%]
```

This is a rather contrived way of illustrating how the WRAPPER directive first processes the block following it, and up to the corresponding END directive, to resolve any directives embedded within. In this case, the values for the scared, beats, vibes, and chill variables are substituted into their correct places.

```
[% WRAPPER box border=1 %]
    Be not [% scared %]; the isle is full of [% beats %],
    Sounds and [% vibes %], that [% chill %] and hurt not.
[% END -%]
```

The WRAPPER directive then calls the *box* template as if it were an INCLUDE directive. In addition to any local variables specified with the WRAPPER (border in this example), it also sets the content variable to contain the processed block output. Here content contains the completed quote from "Be not afeard..." through "...give delight and hurt not".

In the BLOCK box defined at the end of the example, the content variable is referenced like any other, along with the border variable passed in as an explicit argument to the WRAPPER directive:

```
[% BLOCK box %]
<table border="[% border %]">
  <tr>
    <td>
      [%- content -%]
    </td>
  </tr>
</table>
[% END %]
```

This example generates the following output:

```
<table border="1">
  <tr>
    <td>
    Be not afeared; the isle is full of noises,
    Sounds and sweet airs, that give delight and hurt not.
    </td>
  </tr>
</table>
```

3.2.6.6 global

The global variable references a predefined hash array, which is initially empty. It can be used to store any global data that you want shared between templates, regardless of how they are processed, using PROCESS, INCLUDE, etc.

```
[% global.copyright = '&copy; 2003 Arthur Dent' %]
```

3.2.6.7 view, item

The Template Toolkit provides an experimental VIEW directive. It simplifies the process of displaying complex data structures by automatically mapping different data types onto templates designed specifically to deal with them.

In Example 3-11, a VIEW called people_view is defined that contains three BLOCK definitions, for hash, list, and text data items.

Example 3-11. view

```
[% VIEW people_view;
  BLOCK hash;
    "$item.name is from $item.home\n";
  END;

  BLOCK list;
    view.print(person)
      FOREACH person IN item;
  END;

  BLOCK text;
    item;
  END;
END;
-%]

[% people = [
  { name = 'Arthur Dent',
    home = 'Earth' }
```

```
  { name = 'Ford Prefect',
    home = 'Betelgeuse' }
    'Slartibartfast from Magrethea'
  ]
-%]

[% people_view.print(people) %]
```

The BLOCK definitions within the scope of the VIEW...END directives effectively remain local to the VIEW. Each can access the view and item variables that respectively reference the current view object, implemented by the Template::View module, and the current item of data being presented by the view.

The hash block, for example, will be called whenever the view has a hash array that needs presenting. The item variable references the hash array in question, allowing the block to access the item.name and item.home values.

```
BLOCK hash;
  "$item.name is from $item.home\n";
END;
```

The list block is called whenever the view has a list to present. In this case, we use a FOREACH directive to iterate through the items in the list that item now references. For each list element, we call back to the print method of the current view object so that it can correctly select the appropriate template for displaying it.

```
BLOCK list;
  view.print(person)
    FOREACH person IN item;
END;
```

The final block, text, is called whenever the view has a piece of plain text to present. All we need to do is output the value of item. If you want to pass all your text through a filter—to escape any HTML entities, for example—this is where you would do it.

```
BLOCK text;
  item;
END;
```

Having defined some sample data in people, we can then call the print method against the people_view view, passing the people data as an argument:

```
[% people_view.print(people) %]
```

The view will recognize that the argument is a reference to a list, and will call the list block to handle it. This will call the print method for each item in the list. For the first two items, this will result in the hash block being processed. For the last, it will call instead to the text block. The end result is that the right template gets called to handle the right kind of data.

Example 3-11, therefore, outputs the following:

```
Arthur Dent is from Earth
Ford Prefect is from Betelgeuse
Slartibartfast is from Magrethea
```

3.2.7 Variable Scope

Any simple variables that you create, or any changes you make to existing variables, will persist only while the template is being processed. The top-level variable hash is copied before processing begins, and any changes to variables are made in this copy, leaving the original intact. The same thing happens when you INCLUDE another template. The current namespace hash is cloned to prevent any variable changes made in the included template from interfering with existing variables. The PROCESS option bypasses the localization step altogether, making it slightly faster but requiring greater attention to the possibility of side effects caused by creating or changing any variables within the processed template.

Here is an example showing the difference between INCLUDE and PROCESS:

```
[% BLOCK change_name %]
   [% name = 'bar' %]
[% END %]

[% name = 'foo' %]
[% INCLUDE change_name %]
[% name %]                        # foo
[% PROCESS change_name %]
[% name %]                        # bar
```

Dotted compound variables behave slightly differently because the localization process is only skin-deep. The current variable namespace hash is copied, but no attempt is made to perform a deep-copy of other structures within it (hashes, arrays, objects, and so on). A variable referencing a hash, for example, will be copied to create a new reference, but one that points to the same hash. Thus, the general rule is that simple variables (undotted variables) are localized, but existing complex structures (dotted variables) are not.

This examples demonstrates this subtle effect:

```
[% BLOCK all_change %]
   [% x = 20 %]                # changes copy
   [% y.z = 'zulu' %]          # changes original
[% END %]

[% x = 10
   y = { z => 'zebra' }
%]
[% INCLUDE all_change %]
[% x %]                        # still '10'
[% y.z %]                      # now 'zulu'
```

If you create a complex structure such as a hash or list reference within a local template context, it will cease to exist when the template is finished processing:

```
[% BLOCK new_stuff %]
   [% # define a new 'y' hash array in local context
      y = { z => 'zulu' }
   %]
[% END %]

[% x = 10 %]
[% INCLUDE new_stuff %]
[% x %]                        # outputs '10'
[% y %]                        # outputs nothing, y is undefined
```

Similarly, if you update an element of a compound variable that *doesn't* already exist, a hash will be created automatically and deleted again at the end of the block:

```
[% BLOCK new_stuff %]

   [% y.z = 'zulu' %]

[% END %]
```

However, if the hash *does* already exist, you will modify the original with permanent effect. To avoid potential confusion, it is recommended that you don't update elements of complex variables from within blocks or templates included by another block or template.

If you want to create or update truly global variables, use the global namespace, described earlier.

3.2.8 Compile-Time Constant Folding

The default behavior for the Template Toolkit is to look up the value for a variable each and every time it is used in a template. This is what you want most of the time, but it can also be a little wasteful if you have variables that never or rarely change.

For example, you might want to define a set of variables to specify a particular color scheme for your web site. You want to use variables so that you can change the colors quickly and easily at some point in the future. However, you don't expect any of the values to change from one page, template, or web server request to the next. In fact, you would probably prefer it if they couldn't be changed, to protect them from being accidentally overwritten by a careless template author.

The solution is to use the CONSTANTS configuration option to provide a reference to a hash array of variables whose values are constant. The hash array can contain any kind of complex, nested, or dynamic data structures that you would normally define as a regular variable.

```
my $tt = Template->new({

    CONSTANTS => {

        version => 3.14,

        release => 'skyrocket',

        col     => {

            back => '#ffffff',

            fore => '#000000',

        },

        myobj => My::Object->new( ),

        mysub => sub { ... },

        joint => ', ',

    },

});
```

Within a template, these variables are accessed using the constants namespace prefix:

```
Version [% constants.version %] ([% constants.release %])



Background: [% constants.col.back %]
```

When the template is compiled, these variable references are replaced with the corresponding value. No further variable lookup is then performed when the template is processed. This results in templates that can be processed significantly faster by virtue of the fact that they have less work to do in looking up variable values. This can be an important optimization if you're using the Template Toolkit to generate dynamic pages behind an online web server.

Subroutines and objects can be provided as CONSTANTS items. You can even call virtual methods on constant variables:

```
[% constants.mysub(10, 20) %]

[% constants.myobj(30, 40) %]

[% constants.col.keys.sort.join(', ') %]
```

One important proviso is that any arguments you pass to subroutines or methods must also be literal values or compile-time constants.

For example, these are both fine:

```
# literal argument

[% constants.col.keys.sort.join(', ') %]



# constant argument

[% constants.col.keys.sort.join(constants.joint) %]
```

But this next example will raise an error at parse time, complaining that joint is a runtime variable that cannot be determined at compile time:

```
# ERROR: runtime variable argument!

[% constants.col.keys.sort.join(joint) %]
```

The CONSTANTS_NAMESPACE option can be used to provide a different namespace prefix for constant variables. For example:

```
my $tt = Template->new({

    CONSTANTS => {

        version => 3.14,

        # ...etc...

    },

    CONSTANTS_NAMESPACE => 'const',

});
```

Constants would then be referenced in templates as:

```
[% const.version %]
```

3.3 Virtual Methods

The Template Toolkit provides a number of virtual methods, or *vmethods*, that allow you to perform common operations on the three main types of data: scalars, lists, and hash arrays. In many cases, they are analogous to the Perl functions of the same name. The length scalar virtual method, for example, is implemented using Perl's length function.

Some virtual methods are interchangeable between data types. For example, you can call any list virtual method on a single scalar item and it will be treated as if it were a single element list. In other cases, the same virtual method is provided for different data types, providing alternate implementations of similar functionality. The size virtual method, for example, returns 1 for a scalar item, the number of elements in a list, or the number of key/value pairs in a hash array.

Virtual methods are invoked using the regular dot operator syntax:

```
[% string.length %]

[% list.join %]

[% hash.size %]
```

They can be chained together in compound variables, as shown here:

```
[% hash.keys.sort.join(', ') %]
```

The majority of virtual methods compute and return a value without modifying the underlying data (e.g., size). However, there are a number of virtual methods that do, one of which is pop, which removes the last item from a list. Example 3-12 shows examples of both in use.

Example 3-12. beer

```
[% beers   = [ 'Bass' 'Guinness' "Murphy's" ]
   bottles = 'bottles';

   WHILE (n = beers.size)
-%]
   [% n %] [% bottles %] of beer in my list,
   [% n %] [% bottles %] of beer,
   Take one down,
   Pass it around,
[%
   beer = beers.pop
   bottles = beers.max ? 'bottles' : 'bottle'
-%]
   (a bottle of [% beer %] is hastily drunk)
   [% beers.size or 'no' %] [% bottles %] of beer in my list.

[% END %]
```

Example 3-12 will output the following:

```
3 bottles of beer in my list,
3 bottles of beer,
Take one down,
```

Pass it around,

(a bottle of Murphy's is hastily drunk)

2 bottles of beer in my list.

2 bottles of beer in my list,

2 bottles of beer,

Take one down,

Pass it around,

(a bottle of Guinness is hastily drunk)

1 bottle of beer in my list.

1 bottle of beer in my list,

1 bottle of beer,

Take one down,

Pass it around,

(a bottle of Bass is hastily drunk)

no bottles of beer in my list.

3.3.1 Scalar Virtual Methods

The Template Toolkit defines the following virtual methods that operate on scalar values.

3.3.1.1 chunk(size)

This splits the input text into a list of smaller chunks. The argument defines the maximum length in characters of each chunk.

```
[% ccard_no = "1234567824683579";
   ccard_no.chunk(4).join
%]
```

It outputs the following:

```
1234 5678 2468 3579
```

If the size is specified as a negative number, the text will be chunked from right to left. This gives the correct grouping for numbers, for example:

```
[% number = 1234567;
   number.chunk(-3).join(',')
%]
```

and outputs the following:

```
1,234,567
```

3.3.1.2 defined

This returns true if the value is defined, even if it contains an empty string or the number zero. It returns false if the item is undefined.

```
foo [% foo.defined ? 'is' : 'is not' %] defined
```

3.3.1.3 hash

This returns a hash reference containing the original item as the single entry, indexed by the key value:

```
[% name = 'Slartibartfast' %]
[% user = name.hash %]
[% user.value %]        # Slartibartfast
```

3.3.1.4 length

This virtual method returns the number of characters in the string representation of the item:

```
[% IF password.length < 8 %]
  Your password is too short, please try again.
[% END %]
```

3.3.1.5 list

This returns the value as a single element list:

```
[% things = thing.list %]
```

The list virtual method can also be called against a list and will return the list itself, effectively doing nothing. Hence, if thing is already a list, thing.list will return the original list. Either way, things ends up containing a reference to a list.

Most of the time, you don't need to worry about the difference between scalars and lists. You can call a list virtual method against any scalar item and it will be treated as if it were a single element list. The FOREACH directive also works in a similar way. If you pass it a single scalar item instead of a reference to a list, it will behave as if you passed it a reference to a list containing that one item, and will iterate through the block just once.

The list vmethod is provided for those times when you really do want to be sure that you've got a list reference. For example, if you are calling a Perl subroutine that expects a reference to a list, adding the .list vmethod to the argument passed to it will ensure that it gets a list, even if the original argument is a scalar:

```
[% item = 'foo';
  mysub(item.list)  # same as mysub([item])
%]                  #  - item is a scalar

[% item = ['foo'];
  mysub(item.list)  # same as mysub(item)
%]                  #  - item is already a list
```

3.3.1.6 match(pattern)

The match virtual method performs a Perl regular expression match on the string using the pattern passed as an argument. Example 3-13 shows it being used to test whether the value of the serial variable matches the regular expression pattern ^\w{3}-\d{4}$. This pattern requires the string to be composed of exactly three alphanumeric "word" characters (\w{3}), followed by a dash (-), and then exactly four digits (\d{4}). The ^ and $ characters anchor the pattern to the start and end of the string, respectively. Without them, the pattern could match anywhere in what might be a much longer string. In this case, we want to make sure that the serial number is exactly eight characters long—no more, no less.

Example 3-13. serial

```
[% FOREACH serial IN ['ABC-1234', 'FOOD-4567', 'WXYZ-789'];
  IF serial.match('^\w{3}-\d{4}$');
    "GOOD serial number: $serial\n";
  ELSE;
    "BAD serial number: $serial\n";
  END;
END
%]
```

Example 3-13 outputs the following:

```
GOOD serial number: ABC-1234
BAD serial number: FOOD-4567
BAD serial number: WXYZ-789
```

The pattern can contain parentheses to capture parts of the matched string. If the entire pattern matches, the vmethod returns a reference to a list of the captured strings:

```
[% name = 'Arthur Dent' %]
[% matches = name.match('(\w+) (\w+)') %]
[% matches.1 %], [% matches.join('') %]  #  Dent, ArthurDent
```

In this example, the match vmethod returns a list of the two strings matched by the parenthesized patterns, (\w+). Here they are the values Arthur and Dent.

Remember that match returns false if the pattern does not match. It does *not* return a reference to an empty list, which both Perl and the Template Toolkit would treat as a true value, regardless of how many entries it contains. This allows you to test the value returned by match to determine whether the pattern matched.

The following example shows how the results of the match vmethod can be saved in the matches variable, while also testing that the pattern matched. The assignment statement is enclosed in parentheses and used as the expression for an IF directive.

```
[% IF (matches = name.match('(\w+) (\w+)')) %]
   pattern matches: [% matches.join(', ') %]
[% ELSE %]
   pattern does not match
[% END %]
```

Any regular expression modifiers can be embedded in the pattern using the (?imsx-imsx) syntax. For example, a case-insensitive match can be specified by using the (?i) construct at the start of the pattern:

```
[% matched = name.match('(?i)arthur dent') %]
```

In the following fragment, the (?x) flag is set to have whitespace and comments in the pattern ignored:

```
[% matched = name.match(
     '(?x)
        (\w+)   # match first name
         \s+    # some whitespace
        (\w+)   # match second name
     '
   )
%]
```

The details of Perl's regular expressions are described in the perlre(1) manpage. For a complete guide to learning and using regular expressions, see *Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly).

3.3.1.7 repeat(n)

This virtual method returns a string containing the original item repeated a number of times. The repeat value should be passed as an argument.

```
[% name = 'foo ' %]

[% name.repeat(3) %]             # foo foo foo
```

3.3.1.8 replace(search, replace)

This virtual method performs a global search and replace on the input string. The first argument provides a Perl regular expression to match part of the text. The second argument is the replacement value. Each occurrence of the pattern in the input string will be replaced (hence the "global" part of "global search and replace").

```
[% name = 'foo, bar & baz' %]

[% name.replace('\W+', '_') %]    # foo_bar_baz
```

The replace vmethod returns a copy of the string with the appropriate values replaced. The original string is not modified.

3.3.1.9 size

This virtual method always returns 1 for scalar values. It is provided for consistency with the hash and list virtual methods of the same name.

3.3.1.10 split(pattern)

This virtual method splits the input text into a list of strings that is then returned. It uses the regular expression passed as an argument as the delimiter, or whitespace as the default if an explicit delimiter is not provided.

```
[% path  = '/here:/there:/every/where';

   paths = path.split(':');

   paths.join;    # /here /there /every/where

%]
```

3.3.2 List Virtual Methods

The following virtual methods operate on a reference to a list and on scalar items that are treated as if they were single item lists. They can also be called against objects that are implemented as a blessed reference to a list. If the object defines a method—say, size—it will take precedence over the list virtual method of the same name. If the object does not define that method explicitly, the virtual method will instead be called.

```
[% mylistobj.size %]  # object method or list virtual method
```

3.3.2.1 first(n)

This virtual method returns the first item in the list without removing it from the list:

```
[% list  = [10, 20 30] %]

[% list.first      %]   # 10

[% list.join(', ') %]   # 10, 20, 30
```

A number can be provided as an argument. In this case, the vmethod returns a reference to a list containing that many items copied from the start of the list:

```
[% list.first(2).join(', ') %]   # 10, 20
```

3.3.2.2 grep(pattern)

The grep vmethod returns a list of the items in the list that match the regular expression pattern passed as an argument. For example, you can use it to select all the files in a directory listing, files, that have a *.txt* ending:

```
[% txtfiles = files.grep('\.txt$') %]
```

3.3.2.3 join(delimiter)

This virtual method returns the items in the list joined into a single string. By default it uses a single space to join the items.

```
[% list  = [10, 20 30] %]

[% list.join %]        # 10 20 30
```

An alternate delimiter can be provided as an argument:

```
[% list.join(', ') %]   # 10, 20, 30
```

3.3.2.4 last(n)

The last virtual method returns the last item in the list without removing it from the list:

```
[% list  = [10, 20 30] %]

[% list.last       %]   # 30

[% list.join(', ') %]   # 10, 20, 30
```

As with first, an argument can be provided indicating the number of items that should be returned from the end of the list:

```
[% list.last(2).join(', ') %]   # 20, 30
```

3.3.2.5 max

The max virtual method returns the index number for the last element in the list. It is always one less than the value returned by the size virtual method.

```
[% list  = [10, 20 30] %]

[% list.max %]  # 2
```

3.3.2.6 merge(list)

The merge virtual method returns a list composed of the original items in the list plus those from any additional lists passed as arguments:

```
[% list_a = [ 1 2 3 ];

   list_b = [ 4 5 6 ];

   list_c = [ 7 8 9 ];

   list_d = list_a.merge(list_b, list_c);

%]
```

The new list, list_d, contains the items merged from list_a, list_b, and list_c. The original lists are left unmodified.

```
[% list_a.join(', ') %]  # 1, 2, 3

[% list_b.join(', ') %]  # 4, 5, 6

[% list_c.join(', ') %]  # 7, 8, 9

[% list_d.join(', ') %]  # 1, 2, 3, 4, 5, 6, 7, 8, 9
```

3.3.2.7 pop

This virtual method removes the last item from the list and returns it:

```
[% list = [10, 20 30] %]

[% list.pop  %]  # 30
```

3.3.2.8 reverse

The reverse virtual method returns a reference to a new list containing the items in the original list, but in reverse order:

```
[% list  = [10, 20 30] %]

[% list.reverse.join(', ') %]   # 30, 20, 10
```

3.3.2.9 shift

This vmethod removes the first item from the list and returns it:

```
[% list = [10, 20 30] %]

[% list.shift %]  # 10
```

3.3.2.10 size

This virtual method returns the number of elements in the list:

```
[% list = [10, 20 30] %]

[% list.size %]  # 3
```

3.3.2.11 slice(from, to)

This virtual method returns the items in the list between the bounds passed as arguments. If the second argument is not specified, it defaults to the last item in the list. The original list is not modified.

```
[% list = [10, 20 30] %]

[% list.slice(0, 1).join(', ') %]   # 10, 20

[% list.join(', ') %]               # 10, 20, 30
```

The arguments can also be negative numbers, in which case they are counted from the end of the list:

```
[% list.slice(-2, -1).join(', ') %] # 20, 30
```

3.3.2.12 sort, nsort

The sort vmethod returns a list of the items in alphabetical order:

```
[% list = ['foo', 'bar', 'baz'] %]

[% list.sort.join(', ') %]  # bar baz foo
```

The nsort vmethod is similar, but sorts the items in numerical order. The following example illustrates the difference between the two:

```
[% list = ['0.1', '1', '02', '3', '010', '11'] %]

[% list.sort.join(', ')  %]  # 0.1, 010, 02, 1, 11, 3

[% list.nsort.join(', ') %]  # 0.1, 1, 02, 3, 010, 11
```

When the items in the list are references to hash arrays, an optional argument can be used to specify a sort key. This corresponds to an entry in each hash array, the value of which is used to sort the items. This is shown in Example 3-14, where the id and name keys as specified as arguments to the sort virtual method.

Example 3-14. products

```
[% products = [
     { id = 'xyz789', name = 'Foo Widget' }
     { id = 'def456', name = 'Bar Widget' }
     { id = 'abc123', name = 'Baz Widget' }
   ]
-%]
Products sorted by id:
[% FOREACH product IN products.sort('id') -%]
  * [% product.id %] [% product.name %]
[% END -%]

Products sorted by name:
[% FOREACH product IN products.sort('name') -%]
  * [% product.id %] [% product.name %]
[% END -%]
```

The output generated by Example 3-14 is as follows:

```
Products sorted by id:
  * abc123 Baz Widget
  * def456 Bar Widget
  * xyz789 Foo Widget

Products sorted by name:
  * def456 Bar Widget
  * abc123 Baz Widget
  * xyz789 Foo Widget
```

3.3.2.13 splice(offset, length, list)

This virtual method behaves just like Perl's splice function, allowing you to selectively remove or replace part of a list. The first argument defines the offset in the list of the part to be removed, starting at 0 for the first item. With just one argument provided, the vmethod removes everything from that element onward, returning the removed items in a new list.

```
[% primes = [2, 3, 5, 7, 11, 13];
  others = primes.splice(2);
  primes.join(', ');      # 2, 3
  others.join(', ');      # 5, 7, 11, 13
%]
```

The offset can also be specified as a negative number, in which case it is counted backward from the end of the list:

```
[% primes = [2, 3, 5, 7, 11, 13];
   others = primes.splice(-2);
   primes.join(', ');      # 2, 3, 5, 7
   others.join(', ');      # 11, 13
%]
```

A second optional argument can be provided to specify the length of the section to be removed:

```
[% primes = [2, 3, 5, 7, 11, 13];
   others = primes.splice(2, 3);
   primes.join(', ');      # 2, 3, 13
   others.join(', ');      # 5, 7, 11
%]
```

A third optional argument can be used to provide a list of items that will be inserted into the list in place of the removed section. This can be specified as a reference to a list or as a list of items.

```
[% primes1 = [2, 3, 5, 7, 11];
   primes2 = [13, 17, 19];

   # pass reference to list
   primes3 = primes1.splice(1, 2, primes2);
   primes1.join(', ');     # 2, 13, 17, 19, 7, 11
   primes2.join(', ');     # 13, 17, 19
   primes3.join(', ');     # 3, 5

   # pass list of items
   primes4 = primes1.splice(1, 3, 3, 5);
   primes1.join(', ');     # 2, 3, 5, 7, 11
   primes4.join(', ');     # 13, 17, 19
%]
```

3.3.2.14 unique

This vmethod returns a copy of the list with any duplicate values removed:

```
[% mylist = [ 1 2 3 2 3 4 1 4 3 4 5 ];
   numbers = mylist.unique;
   numbers.join(', ');    # 1, 2, 3, 4, 5
%]
```

3.3.2.15 unshift(item)

This virtual method adds an item to the start of a list:

```
[% numbers = [ 2.718, 3.142 ];
   numbers.unshift(1.414);
   numbers.join(', ');    # 1.414, 2.718, 3.142
%]
```

3.3.2.16 push(item)

The push vmethod is similar to unshift, but adds the item to the end of the list:

```
[% numbers = [ 1.414, 2.718 ];
   numbers.push(3.142);
   numbers.join(', ');    # 1.414, 2.718, 3.142
%]
```

3.3.3 Hash Virtual Methods

The following virtual methods operate on hash references. They can also be called against objects that are implemented as blessed hash arrays. As with list virtual methods, any method explicitly provided by the object will take precedence over a hash virtual method of the same name.

```
[% myhashobj.keys %]  # object method or hash virtual method
```

3.3.3.1 defined(key)

The defined virtual method returns true or false to indicate whether a particular item is defined in the hash. A key for the item in question should be passed as an argument:

```
foo [% hash.defined('foo') ? 'is' : 'is not' %] defined
```

3.3.3.2 each

The each virtual method, as shown in Example 3-15, returns a list of the keys and values in the hash, interleaved as key1, value1, key2, value2, etc.

Example 3-15. each

```
[% product = {
     id    = 'ABC-123'
     name  = 'ABC Widget #123',
     price = 7.99,
   }

   keyvals = product.each;

   WHILE (keyvals.size);
     key = keyvals.shift;
     val = keyvals.shift;
```

```
    "$key => $val\n";
  END
%]
```

Example 3-15 outputs the following:

```
id => ABC-123
price => 7.99
name => ABC Widget #123
```

Hash arrays do not maintain any particular order for the items in them, so the each virtual method (and also keys and values, covered later in this section) returns the items in what appears to be a random order.[2] This ensures that key and value return their items in a corresponding order, even if we're not sure what that order will be.

[2] Strictly speaking, it is nondeterministic rather than truly random, although Perl does, of course, have an idea how to determine the "correct" traversal order for a hash array.

3.3.3.3 exists(key)

The exists virtual method performs a similar function to defined, but indicates whether the item exists in the hash. If it does exist, the exists vmethod will return true, even if it is set to an undefined value. In contrast, the defined vmethod returns false if an item exists in the hash but is set to an undefined value.

```
foo [% hash.exists('foo') ? 'does' : 'does not' %] exist
```

3.3.3.4 import(hash)

The import virtual method can be called against a hash array to have it import the elements of another hash array:

```
[% hash1 = {
     foo = 'Foo'
     bar = 'Bar'
  }
  hash2 = {
     wiz = 'Wiz'
     woz = 'Woz'
  }
%]

[% hash1.import(hash2) %]
[% hash1.wiz %]                # Wiz
```

You can also call the import vmethod by itself to import the items in a hash array into the current variable namespace. In effect, the items in the hash array become new template variables.

```
[% user = { id = 'dent' name = 'Arthur Dent' } %]
[% import(user) %]
[% id %]: [% name %] # dent: Arthur Dent
```

3.3.3.5 item(key)

This vmethod performs a simple lookup in the hash, returning the value for the key passed as an argument:

```
[% hash.item('foo') %]
```

This has the same effect as retrieving an item directly:

```
[% hash.foo %]
```

The item virtual method can be used to fetch an item from the hash that might otherwise be confused for a hash virtual method. In the following example, the size item is fetched from the font hash using the item virtual method:

```
[% size = font.item('size') %]  # hash item
```

If the font hash does not contain a size key, it will return an undefined value. If instead we access it directly using the dot operator, the size virtual method will automatically be called if the hash does not contain a defined value for size.

```
[% size = font.size %]          # hash item or vmethod
```

In this case, we would end up with a value defined for size, even if the hash doesn't contain a size item.

3.3.3.6 keys

This virtual method performs the same task as the equivalent Perl function. It returns a reference to a list containing the keys of the hash. As with each, these are returned in no particular order, although it is guaranteed to be the same order as the corresponding values returned by the values vmethod.

```
[% product = {
     id    = 'widget2k'
     name  = "Widget 2000"
     about = "Ultra-fast dynamic widget"
     price = 4.99
   }
%]
[% FOREACH key = product.keys -%]
   [% key %] => [% product.$key %]
[% END %]
```

This generates the following output:

```
about => Ultra-fast dynamic widget
id => widget2k
price => 4.99
name => Widget 2000
```

3.3.3.7 list

The list virtual method returns the contents of the hash as a reference to a list. An argument can be passed to indicate the desired items required in the list: keys to return a list of the keys (same as hash.keys), values to return a list of the values (same as hash.values), or each to return as list of key/value pairs (same as hash.each). When called without an argument, it returns a list of hash references, each of which contains a key and value item representing a single key/value pair in the hash.

Consider the following hash:

```
[% hash = {
      one   = 1
      two   = 2
      three = 3
   };

%]
```

Calling hash.list('keys'):

```
[% FOREACH key IN hash.list('keys') -%]
   [% key %]
[% END %]
```

generates this output:

```
one
three
two
```

Calling hash.list('values'):

```
[% FOREACH key IN hash.list('values') -%]
   [% key %]
[% END %]
```

generates this output:

```
1
3
2
```

Calling hash.list('each'):

```
[% FOREACH key IN hash.list('each') -%]
   [% key %]
[% END %]
```

generates this output:

```
one
1
three
3
two
2
```

Calling hash.list:

```
[% FOREACH keyval IN hash.list -%]
   [% keyval.key %] => [% keyval.value %]
[% END %]
```

generates this output:

```
one => 1
three => 3
two => 2
```

3.3.3.8 size

This virtual method returns the number of key/value pairs in the hash.

3.3.3.9 sort, nsort

The sort virtual method returns a list of the keys sorted alphabetically:

```
[% FOREACH term IN terms.sort %]

  [% term %] means '[% terms.$term %]',

[% END %]
```

The nsort vmethod performs a similar function but returns the keys sorted by their numerical value. See the sort and nsort list virtual methods for an example.

3.3.3.10 values

The values virtual method returns a list of the values in a hash array. They are returned in the same apparently random order as for each and keys.

```
[% keys = product.keys;

  vals = product.vals;



  WHILE keys.size;

    key = keys.shift;

    val = vals.shift;

    "$key => $val\n";

  END

%]
```

3.3.4 Defining New Virtual Methods

You can define your own virtual methods for scalars, lists, and hash arrays. You might do this to add useful functionality not provided by the Template Toolkit itself, or to add methods specific to your data. For example, if you want to offer template designers a way to format a number as a dollar-and-cents string, you might do this with a new virtual method on numbers.

To add a new virtual method from Perl, manipulate package variables yourself to add the new method to the stash:

```
# load Template::Stash to make method tables visible

use Template::Stash;



# define list method to return a new list of palindromic strings only

$Template::Stash::LIST_OPS->{ palindromes } = sub {

    my $list = shift;

    return [ grep { $_ eq reverse($_) } @$list ];

};
```

Alternatively, use the define_vmethod( ) method on the Template Toolkit's context:

```
# locate the context

use Template;

my $template = Template->new( );

my $context  = $tt->context( );



# define list method to return a new list of palindromic strings only
```

```
$context->define_vmethod('list', 'palindromes', sub {
    my $list = shift;
    return [ grep { $_ eq reverse($_) } @$list ];
};
```

3.3.4.1 Stash package variables

The Template::Stash package variables $SCALAR_OPS, $LIST_OPS, and $HASH_OPS are references to hash arrays that define these virtual methods. The HASH_OPS and LIST_OPS virtual methods are implemented as subroutines that accept a hash or list reference as the first item, respectively. The SCALAR_OPS virtual methods are subroutines that accept a scalar value as the first item.

Any other arguments specified when the method is called will also be passed to the subroutine. Any named arguments will be collated into a single hash reference and passed as the last argument, as for any subroutine or method call. This example, therefore:

```
# load Template::Stash to make method tables visible
use Template::Stash;

# define list method to return new list of odd numbers only
$Template::Stash::LIST_OPS->{ odd } = sub {
    my $list = shift;
    return [ grep { $_ % 2 } @$list ];
};
```

creates this template:

```
[% primes = [ 2, 3, 5, 7, 9 ] %]
[% primes.odd.join(', ') %]         # 3, 5, 7, 9
```

New virtual methods can perform arbitrarily complex actions, or very simple actions:

```
$Template::Stash::SCALAR_OPS->{ int } = sub { int($_[0]) };

use Digest::MD5 qw(md5_hex);
$Template::Stash::SCALAR_OPS->{ md5 } = sub { md5_hex($_[0]) };
```

Here is a vmethod to pick an element randomly from a list (courtesy of Slash):

```
$Template::Stash::LIST_OPS->{ rand } = sub {
    my $list = shift;
    return $list->[rand @$list];
};
```

Implementing delete for hashes is straightforward:

```
$Template::Stash::HASH_OPS->{ delete } = sub {
    my ($hash, $key) = @_;
    delete $hash->{ $key } if (defined $key);
}
```

It can be used as you would expect:

```
[% hash.delete('key') %]
```

delete returns the deleted element, just like Perl's delete. This can be chained with other vmethods:

```
[% hash.delete('ccard_no').md5 %]
```

3.3.4.2 Stash and context methods

The Template::Stash and Template::Context modules both implement define_vmethod( ) methods that handle the installation of new virtual methods into the stash package variables. In the case of Template::Context, it simply delegates the task to the current Template::Stash object in use.

The internal architecture of the Template Toolkit is described in painful detail in Chapter 7, but you don't need to know too much about it to be able to define your own virtual methods. The Template object implements a context( ) method that returns the current Template::Context object (the internal template processing engine) that it is using:

```
my $template = Template->new( );

my $context  = $tt->context( );
```

The define_vmethod( ) method can then be called against the $context object. The first argument denotes the data type and should be one of the values scalar, list, or hash. For convenience, item is provided as an alias for scalar, and array as an alias for list. The second argument is the name of the virtual method. The third argument is a reference to the subroutine implementing it.

Here is an example showing another way of adding the odd list virtual method:

```
$context->define_vmethod('list', 'odd', sub {

    my $list = shift;

    return [ grep { $_ % 2 } @$list ];

};
```

This example shows a hash virtual method being added to print a Perl representation of the hash array in sorted order. Here we are using Perl's => operator, which acts just like a comma but saves us from having to quote the hash and dump values.

```
$context->define_vmethod(hash => dump => sub {

    my $hash = shift;

    return '{ '

        . join( ', ',

                map { "$_ => '$hash->{$_}'" }

                sort keys %$hash )

        . ' }';

});
```

If you enable the EVAL_PERL configuration option, you can also define virtual methods in a PERL block from within a template. The $context variable is automatically available for use in PERL blocks.

```
[% PERL %]

$context->define_vmethod(hash => dump => sub {

    my $hash = shift;

    return '{ '

        . join( ', ',

                map { "$_ => '$hash->{$_}'" }

                sort keys %$hash )

        . ' }';

});

[% END %]
```

It is also possible to write a plugin that defines virtual methods. This is covered in Chapter 8.

Chapter 3. The Template Language

While a programming language is designed to manipulate data, a presentation language is used to turn the data into plain text, HTML, or some other format.[1] As long as the data is made available to us in a textual representation when we ask for it, we really don't need to worry too much about how it is stored or computed behind the scenes.

[1] We'll assume for now that the presentation formats are all different kinds of text, although you can also use the Template Toolkit to generate binary files such as images.

That's not to say that you can't create and manipulate variables in templates. However, their most common use is for dealing only with presentation aspects, by using variables to define colors or other layout parameters, displaying the first *N* search results, or sorting a list of names into alphabetical order, for example. It is unusual (but not unheard of) to use the Template Toolkit to modify data that has any lasting effect. In general, data is passed to a template and then thrown away, so it doesn't matter if it's changed in any way.

In this chapter, we take a closer look at the details of the Template Toolkit presentation language. The general syntax of templates comes under scrutiny first, and we give examples of how the default style can be customized using configuration options and template directives. The rest of the chapter is then dedicated to an in-depth study of variables. We describe the various data types, showing how they are defined and used in both Perl and template markup.

We concentrate on the general characteristics of the language without looking too closely at any of the specific directives that the Template Toolkit provides (PROCESS, WRAPPER, USE, and so on). These are described in detail in Chapter 4. A full discussion of filters and plugins is left for Chapter 5 and Chapter 6, respectively.

While you can write templates that have a lasting effect on data—say, by updating a database directly—that's not really how the Template language was intended to be used. We return to this subject in Chapter 11 and Chapter 12, when we look more closely at separating the functional parts of an application from those that deal only with presentation.

4.1 Accessing Variables

The Template Toolkit allows you to define variables in your templates. In this section, we will look at the various directives that the Template Toolkit provides for manipulating template variables.

4.1.1 GET

The GET directive retrieves and outputs the value of the named variable:

```
[% GET foo %]
```

The GET keyword is optional. A variable can be specified in a directive tag by itself:

```
[% foo %]
```

The variable name can have an unlimited number of elements, each separated by a . (dot). Each element can have arguments specified within parentheses:

```
[% foo %]

[% bar.baz %]

[% biz.baz(10) %]
```

See Chapter 3 for a full discussion of template variables.

The GET directive can also take an expression as an argument:

```
[% GET total + tax %]


[% GET length * breadth * height %]
```

Expressions can use any of the mathematical operators +, -, *, /, mod, div, and %. They can be combined using the logical operators and, or, and not. &&, ||, and ! are provided as aliases for and, or, and not.

```
[% GET golgafrincham.answer or 42 %]
```

The mod, div, and % operators carry out integer division. div returns the result of the division and mod returns the modulus (or remainder) from the division:

```
[% SET people = 4

   pies = 10 %]


[% pies %] pies shared between [% people %] people

is [% pies div people %] pies each

(and [% pies mod people %] pies left over)
```

The % operator is a synonym for mod.

The logical operator ?: is also available:

```
[% pies > people * 2 ? 'everyone happy' : 'not enough pies' %]
```

This operator works by evaluating the expression that comes before the question mark to see if it is true or false. If it is true, the operator returns the expression that comes before the : character. If it is false, the operator returns the expression that follows the : character. In the example, status is set to either everyone happy or not enough pies depending on whether we have at least two pies for everyone.

The comparison operators = =, !=, <, <=, >, and >= are also provided. Note that they always compare their operands as strings.

```
[% GET name = = 'Zaphod' ?

        'Greetings Mr. President' :

        'Hello Monkey' %]
```

4.1.2 SET

The SET directive allows you to assign new values to existing variables or to create new temporary variables:

```
[% SET title = 'Hello World' %]
```

The SET keyword is optional when it is unambiguous:

```
[% title = 'Hello World' %]
```

Variables may be assigned the values of other variables, unquoted numbers (digits), literal text (single quotes), or quoted text (double quotes). In the latter case, any variable references within the text will be interpolated when the string is evaluated. Variables should be prefixed by $, using curly braces to explicitly scope the variable name where necessary.

```
[% foo  = 'Foo'  %]              # literal value 'Foo'

[% bar  =  foo   %]              # value of variable 'foo'

[% cost = '$100' %]              # literal value '$100'

[% item = "$bar: ${cost}.00" %]   # value "Foo: $100.00"
```

Multiple variables may be assigned in the same directive and are evaluated in the order specified. Thus, the previous example could have been written:

```
[% foo  = 'Foo'

   bar  = foo

   cost = '$100'

   item = "$bar: ${cost}.00"

%]
```

Simple expressions can also be used, as they can with GET:

```
[% ten    = 10

   twenty = 20

   thirty = twenty + ten

   forty  = 2 * twenty

   fifty  = 100 div 2

   six    = twenty mod 7

%]
```

You can concatenate strings together using the underscore (_) operator. In Perl 5, the . is used for string concatenation, but in Perl 6, as in the Template Toolkit, the . will be used as the method-calling operators and the underscore (_) operator will be used for string concatenation.[1] Note that the operator must be specified with surrounding whitespace that, as Larry says, is construed as a feature:

[1] Larry has since changed his mind and it looks as if the ~ will be the Perl 6 string concat operator. As always, this is all subject to change.

```
[% copyright = '(C) Copyright ' _ year _ ' ' _ author %]
```

You can, of course, achieve a similar effect with double-quoted string interpolation:

```
[% copyright = "(C) Copyright $year $author" %]
```

The SET directive can also take arguments that are expressions in exactly the same way as the GET directive:

```
[% total = price + (price * tax_rate) %]
```

4.1.3 CALL

The CALL directive is similar to GET in evaluating the variable named, but doesn't print the result returned. This can be useful when a variable is bound to a subroutine or object method that you want to call but whose returned value you aren't interested in.

```
[% CALL dbi.disconnect %]


[% CALL inc_page_counter(page_count) %]
```

4.1.4 DEFAULT

The DEFAULT directive is similar to SET but updates only variables that are currently undefined or have no "true" value (in the Perl sense):

```
[% DEFAULT
   name = 'John Doe'
   id   = 'jdoe'
%]
```

This can be particularly useful in common template components to ensure that some sensible default is provided for otherwise undefined variables. If a true value is provided for variables with DEFAULT values, the provided value will be used; otherwise, the default value will be used.

```
[% DEFAULT
   title = 'Hello World'
   bgcol = '#ffffff'
%]
<html>
<head>
<title>[% title %]</title>
</head>

<body bgcolor="[% bgcol %]">
```

DEFAULT can also take an expression as an argument in exactly the same way as GET:

```
[% DEFAULT pies = 3 * people %]
```

DEFAULT has no effect on variables that already have values.

4.10 Exception Handling

No matter how careful you are, things always go wrong. Errors are a fact of life. Your templates could contain bad code and fail to compile. Or you could get an error thrown from the Template Toolkit—maybe it can't find the header file you asked for. Or your back-end code could raise an error—you failed to connect to the required database. The Template Toolkit wouldn't be of much use if common errors such as these caused it to keel over and die. That's why it provides an exception-handling mechanism in the form of TRY...CATCH.

Exceptions are just a fancy way of saying errors. They're structured as objects so that an error can have a type (just a word to identify the kind of error that occurred, such as database, user, or file) and an info field that provides further information about the specifics of the error. They get thrown just like regular errors, via Perl's *die*, but rather than saying die 'bad apple', we say THROW bad apple.

You don't have to explicitly add code to handle errors. If you don't and an error occurs, it gets reported in the usual way. But if you know that errors might occur and you have a sensible way of recovering from them, it's good to add *TRY...CATCH* to do that.

Using the exception mechanism doesn't force you to worry about all errors that might occur. You can filter on the type of error and just look out for your one custom error code to catch, letting everything else pass through. Exceptions can also be nested, so you can catch them at the most appropriate level in your template.

4.10.1 TRY / THROW / CATCH / FINAL

The Template Toolkit supports fully functional, nested exception handling. The *TRY* directive introduces an exception-handling scope that continues until the matching *END* directive. Any errors that occur within that block will be caught and can be handled by one of the *CATCH* blocks defined.

```
[% TRY %]
   ...blah...blah...
   [% CALL somecode %]
   ...etc...
   [% INCLUDE someblock %]
   ...and so on...
[% CATCH %]
   An error occurred!
[% END %]
```

Errors are raised as exceptions (objects of the Template::Exception class) and contain two fields, *type* and *info*. The exception *type* can be any string containing letters, numbers, "_" or ".", and is used to indicate the kind of error that occurred. The *info* field contains an error message indicating what actually went wrong. Within a *CATCH* block, the exception object is aliased to the *error* variable. You can access the *type* and *info* fields directly.

```
[% mydsn = 'dbi:MySQL:foobar' %]
...

[% TRY %]
   [% USE DBI(mydsn) %]
[% CATCH %]
   ERROR! Type: [% error.type %]
          Info: [% error.info %]
[% END %]
```

The previous example generates the following output (assuming a nonexistent database called foobar):

```
ERROR!  Type: DBI

        Info: Unknown database "foobar"
```

The error variable can also be specified by itself and will return a string of the form $type error - $info:

```
...

[% CATCH %]

ERROR: [% error %]

[% END %]
```

The previous example generates the following output:

```
ERROR: DBI error - Unknown database "foobar"
```

Each *CATCH* block may be specified with a particular exception type denoting the kind of error that it should catch. Multiple *CATCH* blocks can be provided to handle different types of exceptions that may be thrown in the *TRY* block. A *CATCH* block specified without any type, as in the previous example, is a default handler that will catch any otherwise uncaught exceptions. This also can be specified as [% CATCH DEFAULT %].

```
[% TRY %]

  [% INCLUDE myfile %]

  [% USE DBI(mydsn) %]

  [% CALL somecode %]

  ...

[% CATCH file %]

  File Error! [% error.info %]

[% CATCH DBI %]

  [% INCLUDE database/error.html %]

[% CATCH %]

  [% error %]

[% END %]
```

Remember that you can specify multiple directives within a single tag, each delimited by ;. Thus, you might prefer to write your simple *CATCH* blocks more succinctly as:

```
[% TRY %]

  ...

[% CATCH file; "File Error! $error.info" %]

[% CATCH DBI;  INCLUDE database/error.html %]

[% CATCH; error %]

[% END %]
```

or even:

```
[% TRY %]

  ...

[% CATCH file ;

     "File Error! $error.info" ;

  CATCH DBI ;

     INCLUDE database/error.html ;
```

```
  CATCH ;
     error ;
  END
%]
```

The *DBI* plugin throws exceptions of the *DBI* type (in case that wasn't already obvious). The other specific exception caught here is of the *file* type.

A *file* error is automatically thrown by the Template Toolkit when it can't find a file, or fails to load, parse, or process a file that has been requested by an *INCLUDE*, *PROCESS*, *INSERT*, or *WRAPPER* directive. If myfile can't be found in the previous example, the [% INCLUDE myfile %] directive will raise a file exception, which is then caught by the [% CATCH file %] block, generating the output:

```
File Error! myfile: not found
```

Note that the *DEFAULT* option (disabled by default) allows you to specify a default file to be used any time a template file can't be found. This will prevent file exceptions from ever being raised when a nonexistent file is requested (unless, of course, the DEFAULT file doesn't exist). Errors encountered once the file has been found (i.e., read error, parse error) will be raised as file exceptions as per usual.

Uncaught exceptions (i.e., the *TRY* block doesn't have a type-specific or default *CATCH* handler) may be caught by enclosing *TRY* blocks that can be nested indefinitely across multiple templates. If the error isn't caught at any level, processing will stop and the Template *process( )* method will return a false value to the caller. The relevant Template::Exception object can be retrieved by calling the *error( )* method.

```
[% TRY %]
  ...
  [% TRY %]
    [% INCLUDE $user.header %]
  [% CATCH file %]
    [% INCLUDE header %]
  [% END %]
  ...
[% CATCH DBI %]
  [% INCLUDE database/error.html %]
[% END %]
```

In this example, the inner *TRY* block is used to ensure that the first *INCLUDE* directive works as expected. We're using a variable to provide the name of the template we want to include, *user.header*, and it's possible this contains the name of a nonexistent template, or perhaps one containing invalid template directives. If the *INCLUDE* fails with a file error, we *CATCH* it in the inner block and *INCLUDE* the default header file instead. Any DBI errors that occur within the scope of the outer *TRY* block will be caught in the relevant *CATCH* block, causing the *database/error.html* template to be processed. Note that included templates inherit all currently defined template variables, so these error files can quite happily access the error variable to retrieve information about the currently caught exception. For example:

database/error.html:

```
<h2>Database Error</h2>

A database error has occurred: [% error.info %]
```

You can also specify a *FINAL* block. This is always processed regardless of the outcome of the *TRY* and/or *CATCH* block. If an exception is uncaught, the *FINAL* block is processed before jumping to the enclosing block or returning to the caller.

```
[% TRY %]
  ...
[% CATCH this %]
  ...
```

```
[% CATCH that %]

   ...

[% FINAL %]

   All done!

[% END %]
```

The output from the *TRY* block is left intact up to the point where an exception occurs. For example, this template:

```
[% TRY %]

   This gets printed

   [% THROW food 'carrots' %]

   This doesn't

[% CATCH food %]

   culinary delights: [% error.info %]

[% END %]
```

generates the following output:

```
This gets printed

culinary delights: carrots
```

The *CLEAR* directive can be used in a *CATCH* or *FINAL* block to clear any output created in the *TRY* block. For example, this template:

```
[% TRY %]

   This gets printed

   [% THROW food 'carrots' %]

   This doesn't

[% CATCH food %]

   [% CLEAR %]

   culinary delights: [% error.info %]

[% END %]
```

generates the following output:

```
culinary delights: carrots
```

Exception types are hierarchical, with each level being separated by the familiar dot operator. A DBI.connect exception is a more specific kind of *DBI* error. Similarly, a myown.error.barf is a more specific kind of myown.error type, which itself is also a myown error. A *CATCH* handler that specifies a general exception type (such as *DBI* or myown.error) will also catch more specific types that have the same prefix as long as a more specific handler isn't defined. Note that the order in which *CATCH* handlers are defined is irrelevant; a more specific handler will always catch an exception in preference to a more generic or default one.

```
[% TRY %]

   ...

[% CATCH DBI ;

     INCLUDE database/error.html ;

   CATCH DBI.connect ;

     INCLUDE database/connect.html ;

   CATCH ;
```

```
    INCLUDE error.html ;
  END
%]
```

In this example, a DBI.connect error has its own handler, a more general *DBI* block is used for all other *DBI* or *DBI.** errors, and a default handler catches everything else.

Exceptions can be raised in a template using the *THROW* directive. The first parameter is the exception type, which doesn't need to be quoted (but can be, it's the same as *INCLUDE*), followed by the relevant error message, which can be any regular value such as a quoted string, variable, etc.

```
[% THROW food "Missing ingredients: $recipe.error" %]


[% THROW user.login 'no user id: please login' %]


[% THROW $myerror.type "My Error: $myerror.info" %]
```

It's also possible to specify additional positional or named parameters to the *THROW* directive if you want to pass more than just a simple message back as the error info field:

```
[% THROW food 'eggs' 'flour' msg='Missing Ingredients' %]
```

In this case, the error info field will be a hash array containing the named arguments—in this case msg => 'Missing Ingredients'—and an args item that contains a list of the positional arguments—in this case eggs and flour. The error type field remains unchanged; here it is set to food.

```
[% CATCH food %]
  [% error.info.msg %]
  [% FOREACH item = error.info.args %]
    * [% item %]
  [% END %]
[% END %]
```

This produces the output:

```
Missing Ingredients
  * eggs
  * flour
```

In addition to specifying individual positional arguments as [% error.info.args.n %], the info hash contains keys directly pointing to the positional arguments, as a convenient shortcut:

```
[% error.info.0 %]   # same as [% error.info.args.0 %]
```

Exceptions can also be thrown from Perl code that you've bound to template variables, or defined as a plugin or other extension. To raise an exception, call *die( )* passing a reference to a Template::Exception object as the argument. This will then be caught by any enclosing *TRY* blocks from where the code was called.

```
use Template::Exception;
...


my $vars = {
    foo => sub {
        # ... do something ...
        die Template::Exception->new('myerr.naughty',
                                     'Bad, bad error');
```

```
    },
};
```

Therefore, this template:

```
[% TRY %]
   ...
   [% foo %]
   ...
[% CATCH myerr ;
     "Error: $error" ;
   END
%]
```

produces the following output:

```
Error: myerr.naughty error - Bad, bad error
```

The info field can also be a reference to another object or data structure, if required:

```
die Template::Exception->new('myerror', {
    module => 'foo.pl',
    errors => [ 'bad permissions', 'naughty boy' ],
});
```

Later, it can be used in a template:

```
[% TRY %]
   ...
[% CATCH myerror %]
   [% error.info.errors.size or 'no';
      error.info.errors.size =  = 1 ? ' error' : ' errors' -%]
   in [% error.info.module %]:
      [% error.info.errors.join(', ') %].
[% END %]
```

generating the output:

```
2 errors in foo.pl:
   bad permissions, naughty boy.
```

You can also call *die( )* with a single string, as is common in much existing Perl code. This will automatically be converted to an exception of the *undef* type (that's the literal string `undef', not the undefined value). If the string isn't terminated with a newline, Perl will append the familiar at $file line $line message.

```
sub foo {
    # ... do something ...
    die "I'm sorry, Dave, I can't do that\n";
}
```

If you're writing a plugin, or some extension code that has the current Template::Context in scope (you can safely skip this section if this means nothing to you), you can also raise an exception by calling the *context throw( )* method. You can pass it a Template::Exception object reference, a pair of *($type, $info)* parameters, or just a *$info* string to create an exception of *undef* type.

```
$context->throw($e);            # exception object

$context->throw('Denied');      # 'undef' type

$context->throw('user.passwd', 'Bad Password');
```

4.10.2 CLEAR

The *CLEAR* directive can be used to clear the output buffer for the current enclosing block. It is most commonly used to clear the output generated from a *TRY* block up to the point where the error occurred.

```
[% TRY %]

   blah blah blah            # this is normally left intact

   [% THROW some 'error' %]  # up to the point of error

   ...

[% CATCH %]

   [% CLEAR %]               # clear the TRY output

   [% error %]               # print error string

[% END %]
```

4.11 Flow Control

Flow control is about making unexpected changes to the execution order of a template. This can be as simple as ending a FOREACH loop early, or as significant as ending the entire template processing process. These are generally exceptional cases, so you probably won't need to use flow-control directives that often, but we discuss them here just in case.

4.11.1 RETURN

The *RETURN* directive can be used to stop processing the current template and return to the template from which it was called, resuming processing at the point immediately after the *INCLUDE*, *PROCESS*, or *WRAPPER* directive. If there is no enclosing template, the Template *process( )* method will return to the calling code with a true value.

```
Before
[% INCLUDE half_wit %]
After

[% BLOCK half_wit %]
This is just half...
[% RETURN %]
...a complete block
[% END %]
```

The previous example produces the following output:

```
Before
This is just half...
After
```

4.11.2 STOP

The *STOP* directive can be used to indicate that the processor should stop gracefully without processing any more of the template document. This is a planned stop, and the Template *process( )* method will return a true value to the caller. This indicates that the template was processed successfully according to the directives within it.

```
[% IF something.terrible.happened %]
  [% INCLUDE fatal/error.html %]
  [% STOP %]
[% END %]

[% TRY %]
  [% USE DBI(mydsn) %]
  ...
[% CATCH DBI.connect %]
  <p>Cannot connect to the database: [% error.info %]</p>
  <br>
  We apologize for the inconvenience.  The cleaning lady
```

```
    has removed the server power to plug in her vacuum cleaner.

    Please try again later.

    </p>

    [% INCLUDE footer %]

    [% STOP %]

[% END %]
```

4.11.3 NEXT

The *NEXT* directive can be used to start the next iteration of a *FOREACH* or *WHILE* loop:

```
[% FOREACH user = userlist %]

    [% NEXT IF user.isguest %]

    Name: [% user.name %]    Email: [% user.email %]

[% END %]
```

4.11.4 LAST

The *LAST* directive can be used to prematurely exit a *FOREACH* or *WHILE* loop:

```
[% FOREACH user = userlist %]

    Name: [% user.name %]    Email: [% user.email %]

    [% LAST IF some.condition %]

[% END %]
```

BREAK can also be used as an alias for *LAST*.

4.12 Debugging

It's possible that you won't get everything just right in your templates the first time you write them. If you have problems working out what exactly is going on as the Template Toolkit is processing your template, the DEBUG directive can help you.

The DEBUG directive enables and disables directive debug messages within a template. It is used with an on or off parameter to enable or disable directive debugging messages from that point forward. When enabled, the output of each directive in the generated output will be prefixed by a comment indicating the file, line, and original directive text.

```
[% DEBUG on %]

directive debugging is on (assuming DEBUG option is set to true)

[% DEBUG off %]

directive debugging is off
```

The *format* parameter can be used to change the format of the debugging message:

```
[% DEBUG format '<!-- $file line $line : [% $text %] -->' %]
```

The DEBUG configuration option must be set to include DEBUG_DIRS for the DEBUG directives to have any effect. If DEBUG_DIRS is not set, the parser will automatically ignore and remove any DEBUG directives.

4.13 Perl Blocks

The Template Toolkit directives that we have seen up to now together define a presentation language that allows you to do just about anything you need to in order to control the display of your data. This is in keeping with the Template Toolkit philosophy of separating processing from presentation.

However, there may be times when you want to go beyond what Template Toolkit offers you. Very occasionally you might need the power of a full programming language within your templates. When nothing else will do, the Template Toolkit also gives you the option of embedding Perl directly in your templates in PERL and RAWPERL directive blocks.

Using PERL and RAWPERL blocks isn't something that is widely encouraged because it tends to make templates messy and hard to read. It also leads to a poor separation of concerns when you mix application code with presentation templates. However, the Template Toolkit doesn't enforce this separation, so you can embed Perl code inside your templates if you really want to. Because we don't encourage it, this feature is disabled by default. You will have to enable the EVAL_PERL configuration option to embed Perl code.

4.13.1 PERL

The PERL directive allows you to embed a block of Perl code in a template. It looks like this:

```
[% PERL %]
  print "Hello world\n"
[% END %]
```

The EVAL_PERL configuration option must be enabled in order to use PERL blocks. If you try to use a PERL block when EVAL_PERL is disabled, a perl exception will be thrown with the message `EVAL_PERL not set':

```
my $template = Template->new({
    EVAL_PERL => 1,
});
```

The Template Toolkit evaluates Perl code in the Template::Perl package. A number of special variables are predefined, providing access to various Template Toolkit objects.

The $context package variable contains a reference to the current Template::Context object. This can be used to access the functionality of the Template Toolkit to process other templates, and load plugins, filters, etc.:

```
[% PERL %]
  print $context->include('myfile');
[% END %]
```

The $stash variable contains a reference to the top-level stash object, which manages template variables. Through this, variable values can be retrieved and updated.

```
[% PERL %]
  $stash->set(foo => 'bar');
  print "foo value: ", $stash->get('foo');
[% END %]
```

The previous example generates the following output:

```
foo value: bar
```

Output is generated from the PERL block by calling print. Before evaluating the code, a filehandle called Template::Perl::PERLOUT is set up and selected as the default output filehandle. This will be connected to whatever output device was defined in the call to process. Your code should use this filehandle instead of STDOUT.

```
[% PERL %]
   print "foo\n";                          # OK
   print PERLOUT "bar\n";                  # OK, same as above
   print Template::Perl::PERLOUT "baz\n";  # OK, same as above
   print STDOUT "qux\n";                   # WRONG!
[% END %]
```

The PERL block may contain other template directives. These are processed before the Perl code is evaluated.

```
[% name = 'Fred Smith' %]

[% PERL %]
   print "[% name %]\n";
[% END %]
```

Thus, the Perl code in the previous example is evaluated as:

```
print "Fred Smith\n";
```

Exceptions may be thrown from within PERL blocks via die, and will be correctly caught by enclosing *TRY* blocks:

```
[% TRY %]
   [% PERL %]
      die "nothing to live for\n";
   [% END %]
[% CATCH %]
   error: [% error.info %]
[% END %]
```

The previous example generates the following output:

```
error: nothing to live for
```

4.13.2 RAWPERL

The Template Toolkit parser reads a source template and generates the text of a Perl subroutine as output. It then uses *eval( )* to evaluate it into a subroutine reference. This subroutine is then called to process the template, passing a reference to the current Template::Context object through which the functionality of the Template Toolkit can be accessed. The subroutine reference can be cached, allowing the template to be processed repeatedly without requiring any further parsing.

For example, a template such as:

```
[% PROCESS header %]
The [% animal %] sat on the [% location %]
[% PROCESS footer %]
```

is converted into the following Perl subroutine definition:

```
sub {
    my $context = shift;
    my $stash   = $context->stash;
    my $output  = '';
```

```
    my $error;

    eval { BLOCK: {
        $output .=  $context->process('header');
        $output .=  "The ";
        $output .=  $stash->get('animal');
        $output .=  " sat on the ";
        $output .=  $stash->get('location');
        $output .=  $context->process('footer');
        $output .=  "\n";
    } };
    if ($@) {
        $error = $context->catch($@, \$output);
        die $error unless $error->type eq 'return';
    }

    return $output;
}
```

To examine the Perl code generated, such as in the previous example, set the $Template::Parser::DEBUG package variable to any true value. You can also set the $Template::Directive::PRETTY variable to true to have the code formatted in a readable manner for human consumption. The source code for each generated template subroutine will be printed to STDERR on compilation (i.e., the first time a template is used).

```
$Template::Parser::DEBUG = 1;
$Template::Directive::PRETTY = 1;

...

$tt->process($file, $vars)
    || die $tt->error( ), "\n";
```

The PERL ... END construct allows Perl code to be embedded into a template (when the *EVAL_PERL* option is set), but it is evaluated at "runtime" using *eval( )* each time the template subroutine is called. This is inherently flexible but not as efficient as it could be, especially in a persistent server environment where a template may be processed many times.

The *RAWPERL* directive allows you to write Perl code that is integrated directly into the generated Perl subroutine text. It is evaluated once at compile time and is stored in cached form as part of the compiled template subroutine. This makes *RAWPERL* blocks more efficient than *PERL* blocks.

The downside is that you must code much closer to the metal. Within *PERL* blocks, you can call *print( )* to generate some output. *RAWPERL* blocks don't afford such luxury. The code is inserted directly into the generated subroutine text and should conform to the convention of appending to the $output variable.

```
[% PROCESS  header %]

[% RAWPERL %]
   $output .= "Some output\n";
```

```
  ...
  $output .= "Some more output\n";
[% END %]
```

The critical section of the generated subroutine for this example would then look something like this:

```
...
eval { BLOCK: {
   $output .=  $context->process('header');
   $output .=  "\n";
   $output .= "Some output\n";
   ...
   $output .= "Some more output\n";
   $output .=  "\n";
} };
...
```

As with *PERL* blocks, the $context and $stash references are predefined and available for use within *RAWPERL* code.

Only very advanced Template Toolkit users will ever need to use a *RAWPERL* block.

4.2 Accessing External Templates and Files

Variables are for storing little bits of data. Templates are for writing larger chunks of content. As with variables, it is often useful to be able to reuse the contents of a template. For example, the output of a template will often actually be composed of the output of a number of lower-level templates. These lower-level templates can be reused in other templates. This is very similar to the modular approach to writing programs that encourages code reuse.

The Template Toolkit provides a number of directives for manipulating templates. The first three of these all work in a very similar way. INSERT, PROCESS, and INCLUDE all insert the contents of another named template into the current template. The basic syntax for these directives looks like this:

```
[% INCLUDE filename %]
```

You may optionally include arguments (in a name = value format) that define variables to use while processing the included template:

```
[% INCLUDE filename title = "la la la"
                moonphase = "waxing" %]
```

With all of these directives, the results of processing the template are included in the output in place of the directive. The WRAPPER directive works a little differently. It is a block directive and it allows you to define a template that is wrapped around the block of content. The content of the block is made available to the *wrapper* template in a special variable called content.

```
[% WRAPPER layout %]
  blah blah
[% END %]
```

We discuss the directives for manipulating templates in the next four sections.

4.2.1 INSERT

The INSERT directive is used to insert the contents of an external file at the current position:

```
[% INSERT myfile %]
```

No attempt to parse or process the file is made. The contents, possibly including any embedded template directives, are inserted intact.

The filename specified should be relative to one of the INCLUDE_PATH directories. Absolute (i.e., starting with /) and relative (i.e., starting with .) filenames may be used if the ABSOLUTE and RELATIVE options are set, respectively. Both of these options are disabled by default.

```
my $tt = Template->new({
    INCLUDE_PATH => '/here:/there:/every/where',
});


$tt->process('myfile');
```

The contents of *myfile* are:

```
[% INSERT foo %]            # looks for /here/foo then /there/foo
[% INSERT /etc/passwd %]    # file error: ABSOLUTE not set
[% INSERT ../secret %]      # file error: RELATIVE not set
```

For convenience, the filename does not need to be quoted as long as it contains only alphanumeric characters, underscores, dots, or forward slashes. Names containing any other characters should be quoted.

```
[% INSERT misc/legalese.txt            %]
[% INSERT 'dos98/Program Files/foobar' %]
```

To evaluate a variable to specify a filename, you should explicitly prefix it with a $ or use double-quoted string interpolation:

```
[% language = 'en'
   legalese = 'misc/legalese.txt'
%]

[% INSERT $legalese %]              # 'misc/legalese.txt'
[% INSERT "$language/$legalese" %]  # 'en/misc/legalese.txt'
```

Multiple files can be specified using + as a delimiter. All files should be unquoted names or quoted strings. Any variables should be interpolated into double-quoted strings.

```
[% INSERT legalese.txt + warning.txt %]
[% INSERT  "$legalese" + warning.txt %]  # requires quoting
```

4.2.2 INCLUDE

The INCLUDE directive is used to process and include the output of another template file or block:

```
[% INCLUDE header %]
```

If a BLOCK of the specified name is defined in the same file or in a file from which the current template has been called (i.e., a parent template), it will be used in preference to any file of the same name.

```
[% INCLUDE table %]             # uses BLOCK defined below

[% BLOCK table %]
   <table>
   ...
   </table>
[% END %]
```

If a BLOCK definition is not currently visible, the template name should be a file relative to one of the INCLUDE_PATH directories, or an absolute or relative filename if the ABSOLUTE / RELATIVE options are appropriately enabled. The INCLUDE directive automatically quotes the filename specified, as per INSERT described earlier. When a variable contains the name of the template for the INCLUDE directive, it should be explicitly prefixed by $ or double-quoted:

```
[% myheader = 'my/misc/header' %]
[% INCLUDE   myheader  %]           # 'myheader'
[% INCLUDE  "myheader" %]           # 'myheader'
[% INCLUDE  $myheader  %]           # 'my/misc/header'
[% INCLUDE "$myheader" %]           # 'my/misc/header'
```

Any template directives embedded within the file will be processed accordingly. All variables currently defined will be visible and accessible from within the included template.

```
[% title = 'Hello World' %]
[% INCLUDE header %]
<body>
...
```

Therefore, this *header* template:

```
<html>
<title>[% title %]</title>
```

provides the following output:

```
<html>
<title>Hello World</title>
<body>
...
```

Local variable definitions may be specified after the template name, temporarily masking any existing variables. Insignificant whitespace is ignored within directives, so you can add variable definitions on the same line, on the next line, or split across several lines with comments interspersed, if you prefer.

```
[% INCLUDE table %]


[% INCLUDE table title="Active Projects" %]


[% INCLUDE table
     title   = "Active Projects"
     bgcolor = "#80ff00"    # chartreuse
     border  = 2
%]
```

The INCLUDE directive localizes (i.e., copies) all variables before processing the template. Any changes made within the included template will not affect variables in the including template.

```
[% foo = 10 %]


foo is originally [% foo %]
[% INCLUDE bar %]
foo is still [% foo %]


[% BLOCK bar %]
   foo was [% foo %]
   [% foo = 20 %]
   foo is now [% foo %]
[% END %]
```

The preceding example produces the following output:

```
foo is originally 10
   foo was 10
   foo is now 20
foo is still 10
```

> The localization of the stash (that is, the process by which variables are copied before an INCLUDE to prevent being overwritten) is only skin-deep. The top-level variable namespace (hash) is copied, but no attempt is made to perform a deep-copy of other structures (hashes, arrays, objects, etc.). Therefore, a foo variable referencing a hash will be copied to create a new foo variable that points to the same hash array. Thus, if you update compound variables (e.g., foo.bar), you will change the original copy, regardless of any stash localization. If you're not worried about preserving variable values, or you trust the templates you're including, you might prefer to use the PROCESS directive, which is faster by virtue of not performing any localization.

You can specify dotted variables as "local" variables to an INCLUDE directive. However, be aware that because of the localization issues explained earlier (if you skipped the previous Note, you might want to go back and read it, or else skip this section too), the variables might not actually be "local." If the first element of the variable name already references a hash array, the variable update will affect the original variable.

```
[% foo = {
       bar = 'Baz'
   }
%]

[% INCLUDE somefile foo.bar='Boz' %]

[% foo.bar %]            # Boz
```

This behavior can be a little unpredictable (and may well be improved upon in a future version). If you know what you're doing with it and you're sure that the variables in question are defined (nor not) as you expect them to be, you can rely on this feature to implement some powerful "global" data-sharing techniques. Otherwise, you might prefer to steer clear and always pass simple (undotted) variables as parameters to INCLUDE and other similar directives.

If you want to process several templates simultaneously, you can specify each of their names (quoted or unquoted names only, no unquoted $variables) joined together by +. The INCLUDE directive will then process them in order.

```
[% INCLUDE html/header + "site/$header" + site/menu
     title = "My Groovy Web Site"
%]
```

The variable stash is localized once and then the templates specified are processed in order, all within that same variable context. This makes it slightly faster than specifying several separate INCLUDE directives (because you clone the variable stash only once instead of *n* times), but it's not quite as "safe" because any variable changes in the first file will be visible in the second, third, and so on. This might be what you want, of course, but then again, it might not.

4.2.3 PROCESS

The PROCESS directive is similar to INCLUDE but does not perform any localization of variables before processing the template. Any changes made to variables within the included template will be visible in the including template. For example, this code:

```
[% foo = 10 %]

foo is [% foo %]
[% PROCESS bar %]
foo is [% foo %]

[% BLOCK bar %]
  [% foo = 20 %]
  changed foo to [% foo %]
[% END %]
```

produces this output:

```
foo is 10
  changed foo to 20
foo is 20
```

Parameters may be specified in the PROCESS directive, but these too will become visible changes to current variable values. As such, the following code:

```
[% foo = 10 %]

foo is [% foo %]

[% PROCESS bar

   foo = 20

%]

foo is [% foo %]



[% BLOCK bar %]

   this is bar, foo is [% foo %]

[% END %]
```

produces the following output:

```
foo is 10

   this is bar, foo is 20

foo is 20
```

The PROCESS directive is slightly faster than the INCLUDE directive because it avoids the need to localize (i.e., copy) the variable stash before processing the template. As with INSERT and INCLUDE, the first parameter does not need to be quoted as long as it contains only alphanumeric characters, underscores, periods, or forward slashes. A $ prefix can be used to explicitly indicate a variable that should be interpolated to provide the template name:

```
[% myheader = 'my/misc/header' %]

[% PROCESS  myheader %]              # 'myheader'

[% PROCESS $myheader %]              # 'my/misc/header'
```

As with INCLUDE, multiple templates can be specified, delimited by +, and are processed in order:

```
[% PROCESS html/header + my/header %]
```

4.2.4 WRAPPER

It's not unusual to find yourself adding common headers and footers to pages or sub-sections within a page. For example:

```
[% INCLUDE section/header

   title = 'Quantum Mechanics'

%]

   Quantum mechanics is a very interesting subject which

   should prove easy for the layman to fully comprehend.

[% PROCESS section/footer %]



[% INCLUDE section/header

   title = 'Desktop Nuclear Fusion for Under $50'

%]

   This describes a simple device that generates significant

   sustainable electrical power from common tap water via the process
```

```
  of nuclear fusion.
[% PROCESS section/footer %]
```

The individual template components being included might look like the folowing examples:

section/header:

```
<p>
<h2>[% title %]</h2>
```

section/footer:

```
</p>
```

The WRAPPER directive provides a way of simplifying this a little. It encloses a block to a matching END directive, which is first processed to generate some output. This is then passed to the named template file or BLOCK as the content variable.

```
[% WRAPPER section
  title = 'Quantum Mechanics'
%]
  Quantum mechanics is a very interesting subject which
  should prove easy for the layman to fully comprehend.
[% END %]


[% WRAPPER section
  title = 'Desktop Nuclear Fusion for Under $50'
%]
  This describes a simple device that generates significant
  sustainable electrical power from common tap water via the process
  of nuclear fusion.
[% END %]
```

The single *section* template can then be defined as:

```
<p>
<h2>[% title %]</h2>
[% content %]
</p>
```

Like other block directives, it can be used in side-effect notation:

```
[% INSERT legalese.txt WRAPPER big_bold_table %]
```

It's also possible to specify multiple templates to a WRAPPER directive. The specification order indicates outermost to innermost wrapper templates. For example, given the following template block definitions:

```
[% BLOCK bold   %]<b>[% content %]</b>[% END %]
[% BLOCK italic %]<i>[% content %]</i>[% END %]
```

the directive:

```
[% WRAPPER bold + italic %]Hello World[% END %]
```

would generate the following output:

```
<b><i>Hello World</i></b>
```

4.3 Defining Local Template Blocks

Sometimes, particularly in a project that involves a large number of small templates, it doesn't seem very efficient to create an external file for every template that you need. The BLOCK ... END construct can be used to avoid this. It allows you to define template component blocks that can be processed with the INCLUDE, PROCESS, and WRAPPER directives.

```
[% BLOCK tabrow %]

<tr><td>[% name %]<td><td>[% email %]</td></tr>

[% END %]



<table>

[% PROCESS tabrow  name='Fred'  email='fred@nowhere.com' %]

[% PROCESS tabrow  name='Alan'  email='alan@nowhere.com' %]

</table>
```

A BLOCK definition can be used before it is defined, as long as the definition resides in the same file. The block definition itself does not generate any output.

```
[% PROCESS tmpblk %]



[% BLOCK tmpblk %] This is OK [% END %]
```

You can use an anonymous BLOCK to capture the output of a template fragment:

```
[% julius = BLOCK %]

   And Caesar's spirit, ranging for revenge,

   With Ate by his side come hot from hell,

   Shall in these confines with a monarch's voice

   Cry  'Havoc', and let slip the dogs of war;

   That this foul deed shall smell above the earth

   With carrion men, groaning for burial.

[% END %]
```

Like a named block, an anonymous block can contain any other template directives that are processed when the block is defined. The output generated by the block is then assigned to the variable julius.

Anonymous BLOCKs can also be used to define block macros. The enclosing block is processed each time the macro is called.

```
[% MACRO locate BLOCK %]

   The [% animal %] sat on the [% place %].

[% END %]



[% locate(animal='cat', place='mat') %]    # The cat sat on the mat

[% locate(animal='dog', place='log') %]    # The dog sat on the log
```

4.4 Loops

It is very common to want to repeat parts of a template. You might want to produce similar output for every item in a list, or you might want to repeat a piece of content a set number of times. The Template Toolkit provides two loop directives that deal with both of these situations—FOREACH (also spelled FOR) and WHILE.

Use FOREACH in cases where you know the size of the data set over which you are iterating, or in cases where you need access to loop metadata, such as the next or previous element, the index of the iteration, or the size of the data set. WHILE is useful for performing an action until a condition is true, for looping over a very large data set, or when termination of the loop depends on a condition external to the data set. Both directives are discussed in the sections that follow.

4.4.1 FOREACH

The FOREACH directive defines a block, up to the corresponding END tag, that is processed repeatedly for each item in a list. The basic syntax is:

```
[% FOREACH item IN list %]
   # content of block
[% END %]
```

You can also use = in place of IN if you find that more natural:

```
[% FOREACH item = list %]
   # content of block
[% END %]
```

FOREACH loops over each element in a list and creates an alias to the current item:

```
[% numbers = [ 1 .. 5 ] %]


[% FOREACH num IN numbers %]
    * [% num %]
[% END %]
```

In this example, numbers is an array of five elements, the numbers 1 through 5. In the FOREACH loop, these elements are assigned to num, one at a time, in the order that they occur in numbers:

```
* 1
* 2
* 3
* 4
* 5
```

4.4.1.1 Complex data

The elements of the array can be any kind of complex data:

```
[% fabfour = [
    {
      name       = "John Lennon"
      instrument = "guitar"
```

```
    }
    {
      name       = "Paul McCartney"
      instrument = "bass guitar"
    }
    {
      name       = "George Harrison"
      instrument = "lead guitar"
    }
    {
      name       = "Ringo Starr"
      instrument = "drums"
    }
  ]
%]
[% FOREACH beatle IN fabfour -%]
   [% beatle.name %] played [% beatle.instrument %].
[% END %]
```

The beatle variable is aliased to each hash in the fabfour list, and through it we can access the various elements:

```
John Lennon played guitar.
Paul McCartney played bass guitar.
George Harrison played lead guitar.
Ringo Starr played drums.
```

The original array is not modified, but the elements of the array can be modified within the FOREACH loop.

4.4.1.2 Importing hash array items

When the FOREACH directive is used without specifying a target variable, any iterated values that are hash references will be automatically imported:

```
[% FOREACH fabfour -%]
   [% name %] played [% instrument %].
[% END %]
```

This particular usage creates a localized variable context to prevent the imported hash keys from overwriting any existing variables. The imported definitions and any other variables defined in such a FOREACH loop will be lost at the end of the loop, when the previous context and variable values are restored.

4.4.1.3 Iterating over entries in a hash array

The FOREACH directive can also be used to iterate over the entries in a hash array. Each entry in the hash is returned in sorted order (based on the key) as a hash array containing "key" and "value" items.

```
[% users = {
    tom   = 'Thomas'
    dick  = 'Richard'
```

```
    larry = 'Lawrence'
  }
%]

[% FOREACH user IN users %]
  * [% user.key %] : [% user.value %]
[% END %]
```

The previous example generates the following output:

```
* dick : Richard
* larry : Lawrence
* tom : Thomas
```

To iterate over the keys of a hash, use the keys virtual method on the hash:

```
[% FOREACH key IN hash.keys %]
   * [% key %] : [% hash.$key %]
[% END %]
```

4.4.1.4 The loop iterator object

The underlying implementation of the FOREACH directive involves the creation of a special object called an *iterator*, which maintains metadata about the data set being processed. This object can be accessed within the body of the FOREACH using the special variable loop:

```
[% FOREACH item IN items %]
   [% IF loop.first %]
     <ul>
   [% END %]
       <li>[% item %] ([% loop.count %] of [% loop.size %])</li>
   [% IF loop.last %]
     </ul>
   [% END %]
[% END %]
```

The iterator defines several useful methods that return information about the current loop:

size

Returns the size of the data set, or returns undef if the dataset has not been defined

max

Returns the maximum index number (i.e., the index of the last element), which is equivalent to size - 1

index

Returns the number of the current item, in the range 0 to max

count

Returns the current iteration count in the range 1 to size, equivalent to index + 1 *first*

Returns a Boolean value to indicate whether the iterator is currently on the first iteration of the set

last

Returns a Boolean value to indicate whether the iterator is currently on the last iteration of the set

prev

Returns the previous item in the data set, or returns undef if the iterator is on the first item

next

Returns the next item in the data set, or undef if the iterator is on the last item

An iterator plugin is available that enables you to control how an iterator is created; if an iterator object is passed to a FOREACH loop, it is used as is (a new iterator is not created).

```
[% USE all_data = iterator(list_one.merge(list_two)) %]

[% FOREACH datum = all_data %]

   ...

[% END %]
```

4.4.1.5 Nested FOREACH loops

Nested loops will work as expected, with the loop variable correctly referencing the innermost loop and being restored to any previous value (i.e., an outer loop) at the end of the loop:

```
[% FOREACH group IN grouplist;

     # loop => group iterator

     "Groups:\n" IF loop.first;



     FOREACH user IN group.userlist;

        # loop => user iterator

        "$loop.count: $user.name\n";

     END;



     # loop => group iterator

     "End of Groups\n" IF loop.last;

  END

%]
```

The iterator plugin can also be used to explicitly create an iterator object. This can be useful within nested loops where you need to keep a reference to the outer iterator within the inner loop. The iterator plugin effectively allows you to create an iterator by a name other than loop. See the manpage for Template::Plugin::Iterator for further details.

```
[% USE giter = iterator(grouplist) %]



[% FOREACH group IN giter %]
```

```
  [% FOREACH user IN group.userlist %]
      user #[% loop.count %] in
      group [% giter.count %] is
      named [% user.name %]
  [% END %]
[% END %]
```

4.4.2 WHILE

WHILE loops are used to repeatedly process a template block. This block is enclosed within [% WHILE (test) %] ... [% END %] blocks and can be arbitrarily complex. The test condition follows the same rules as those for IF blocks.

```
[%  total = 0;
    WHILE total <= 100 %]
        Total: [% total;
        total = total + 1;
    END;
%]
```

An assignment can be enclosed in parentheses to evaluate the assigned value:

```
[% WHILE (user = next_user) %]
  [% user.name %]
[% END %]
```

The Template Toolkit uses a fail-safe counter to limit the number of loop iterations to prevent runaway loops that never terminate. If the loop exceeds 1,000 iterations, an undef exception will be thrown, reporting the error:

```
WHILE loop terminated (> 1000 iterations)
```

This number can be adjusted from within Perl by setting the $Template::Directive::WHILE_MAX variable.

4.4.2.1 Flow control: NEXT and LAST

The NEXT directive starts the next iteration in a FOREACH or WHILE loop:

```
[% FOREACH user IN userlist %]
  [% NEXT IF user.isguest %]
  Name: [% user.name %]    Email: [% user.email %]
[% END %]
```

The LAST directive can be used to prematurely exit the loop. BREAK is also provided as an alias for LAST.

```
[% FOREACH match IN results.nsort('score').reverse %]
  [% LAST IF match.score < 50 %]
  [% match.score %] : [% match.url %]
[% END %]
```

See the section titled Section 4.11 later in this chapter for more details.

4.5 Conditionals

Often you don't know exactly what output is required until you process the template. Perhaps your web site should be orange on certain days of the week, or maybe negative numbers should be displayed in red. The Template Toolkit has a number of conditional directives that allow your template to make decisions about what path to take.

A *conditional* controls execution of a block of code, based on the value of a variable. In the Template Toolkit, there are two main conditional directives: IF and SWITCH. In addition, there is the UNLESS directive, which is a negated IF.

4.5.1 IF, ELSIF, ELSE, and UNLESS

The primary directive for *conditional execution* is the IF statement. The basic syntax is:

```
[% IF test %]

   action

[% END %]
```

where action is executed only if test is true (the Template Toolkit's definition of "truth" is explained later in this section). IF statements allow for an optional ELSE clause, which is executed if test is not true. There can be multiple test/action pairs as well; these are written using the ELSIF statement:

```
[% IF today = = "friday" %]

   Yay! It's Friday!

[% ELSIF today = = "monday" %]

   Yuck.  It's Monday.

[% ELSE %]

   ...

[% END %]
```

There can be any number of ELSIF clauses, including none. The ELSE clause is also optional. Because the IF directive defines a block, the END token is not optional.

The test clause can be any statement, even just a single variable name; the extreme case is a test clause of 1—i.e., always true. If the result of this statement is 0 or "" (the empty string), test is considered to be false; everything else is true. Variables that have not been assigned a value, either with DEFAULT or SET, are considered to be false (the value of an undefined variable is an empty string).

More complex statements are possible, such as the earlier example. test can be arbitrarily complex. Other than simple variable value, another common test is equality or comparison: what value does a variable contain? The notation = = is used to compare strings because = is used for assignment—it is an error to try to assign to a variable in an IF statement, to prevent subtle errors and hard-to-diagnose problems. Comparison operators include:

```
= =        Test for equality

!=         Test for inequality

<          Less than

<=         Less than or equal to

>          Greater than

>=         Greater than or equal to

&&, AND    grouping

||, OR     grouping

!, NOT     negation
```

Some of these make sense only for numbers, such as >, >=, <, and <=. NOT is used to reverse the meaning of a test:

```
[% IF NOT today %]
   Error! 'today' not defined!
[% END %]
```

There is a special version of IF that does exactly this: UNLESS.

```
[% UNLESS today %]
   ...
```

UNLESS is exactly equivalent to IF NOT, and often clarifies the intent of the condition (but can be more confusing when combined with ELSIF clauses, even though this is a syntactically legal thing to do).

AND and OR can be used to construct compound statements that might otherwise require nested IF blocks:

```
[% IF today = = "Friday" AND time >= 1700 %]
   Go home!  It's the weekend!
[% END %]
```

Without grouping, this would need to be:

```
[% IF today = = "Friday" %]
   [% IF time >= 1700 %]
      Go home!  It's the weekend!
   [% END %]
[% END %]
```

As you can imagine, this would get very tedious for blocks with many options.

4.5.2 SWITCH and CASE

The SWITCH directive makes writing long IF / ELSIF / ELSE statements easier when the test condition needs to be compared to a number of possible outcomes. SWITCH consists of a single statement, which is evaluated once, and a number of CASE statements, against which the evaluated value is compared. For example:

```
[% SWITCH today %]
   [% CASE "Monday" %]
   Hi ho, hi ho, it's off to work we go.

   [% CASE "Friday" %]
   Friday's here, almost time for the weekend!

   [% CASE [ "Saturday" "Sunday" ] %]
   It's the weekend! Party!

   [% CASE %]
   Ho hum, just another workday...
[% END %]
```

The value in today is compared against each successive CASE statement until a match is found; the contents of the matching CASE statement are processed, or the contents of the default CASE statement are processed if no match is found (if there is a default CASE statement, of course). An equivalent IF / ELSIF / ELSE block would look like this:

```
[% IF today = = "Saturday" OR today = = "Sunday" %]
```

```
    It's the weekend! Party!

[% ELSIF today = = "Monday" %]

    Hi ho, hi ho, it's off to work we go.

[% ELSIF today = = "Friday" %]

    Friday's here, almost time for the weekend!

[% ELSE %]

    Ho hum, just another workday...

[% END %]
```

The SWITCH statement is cleaner and there is less syntax to maintain. Most important, however, is that if the *test* statement requires computation instead of just variable comparison, the SWITCH will be more efficient and has less potential for side effects.

4.6 Filters

One of the problems with templates is that you can never be completely sure what content will be produced at the end. This is particularly true if you are pulling in some of your data from an external source. Perhaps you are producing an HTML page from news stories that have been entered into a database by reporters. You can't be sure the stories don't contain characters such as < or & that should be plain text but will be interpreted as HTML. Or perhaps you have room for only a certain number of characters and you don't know how long a story will be.

The Template Toolkit provides filters to deal with these cases. A filter can be applied to part of a template and will postprocess those parts in a defined manner. For example, the html filter converts troublesome characters to their equivalent HTML entities, and the truncate filter will truncate text to a given length.

The FILTER directive introduces a filter, which operates on a block:

```
[% FILTER html %]

  HTML text may have < and > characters embedded

  that you want converted to the correct HTML entities.

[% END %]
```

The previous example produces the following output:

```
HTML text may have &lt; and &gt; characters embedded

that you want converted to the correct HTML entities.
```

The FILTER directive can also follow various other nonblock directives. For example:

```
[% INCLUDE mytext FILTER html %]
```

The | character can also be used as an alias for FILTER:

```
[% INCLUDE mytext | html %]
```

Multiple filters can be chained together and will be called in sequence:

```
[% INCLUDE mytext FILTER html FILTER html_para %]
```

or:

```
[% INCLUDE mytext | html | html_para %]
```

A number of standard filters are provided with the Template Toolkit; these are detailed in Chapter 5.

4.7 Plugins

It is obviously impossible for the Template Toolkit to do everything that everyone might want to do with it. For one thing, we haven't heard of every possible piece of software that you might want to talk to, and for another, no one would want a template processor that is infinite in size! Instead, we provided the plugin mechanism, which makes it possible to write extensions to the Template Toolkit. This is a far saner solution.

Plugins are externally defined extensions that can be dynamically loaded into templates to provide functionality. A plugin is a regular Perl module that conforms to a particular object-oriented interface, allowing it to be loaded into and used automatically by the Template Toolkit. The next subsections discuss directives for working with plugins.

4.7.1 USE

The *USE* directive loads and initializes "plugin" extension modules:

```
[% USE date %]
```

This makes a date plugin object available to the template, which can be used by referencing the variable date:

```
Today is [% date.format(date.now, "%A") %].
```

which might return:

```
Today is Monday.
```

The plugin name is case sensitive and will be appended to the PLUGIN_BASE value (which defaults to *Template::Plugin*) to construct a full module name. Any periods (i.e., .), in the name will be converted to ::.

```
[% USE MyPlugin %]     #  => Template::Plugin::MyPlugin

[% USE Foo.Bar  %]     #  => Template::Plugin::Foo::Bar
```

Various standard plugins are included with the Template Toolkit (see Chapter 6). These can be specified in lowercase and are mapped to the appropriate name:

```
[% USE cgi   %]        # => Template::Plugin::CGI

[% USE table %]        # => Template::Plugin::Table
```

Any additional parameters supplied in parentheses after the plugin name also will be passed to the *new( )* constructor. A reference to the current Template::Context object is always passed as the first parameter. Thus:

```
[% USE MyPlugin('foo', 123) %]
```

is equivalent to:

```
Template::Plugin::MyPlugin->new($context, 'foo', 123);
```

Named parameters may also be specified. These are collated into a hash that is passed by reference as the last parameter to the constructor, as per the general code-calling interface. Thus:

```
[% USE url('/cgi-bin/foo', mode='submit', debug=1) %]
```

is equivalent to:

```
Template::Plugin::URL->new($context, '/cgi-bin/foo',

                           { mode => 'submit', debug => 1 });
```

The plugin may represent any data type—a simple variable, hash, list, or code reference—but in general it will be an object reference. Methods can be called on the object (or on the relevant members of the specific data type) in the usual way:

```
[% USE table(mydata, rows=3) %]



[% FOREACH row = table.rows %]

   <tr>

  [% FOREACH item = row %]

     <td>[% item %]</td>
```

```
  [% END %]

  </tr>

[% END %]
```

A plugin can be referenced by an alternative name:

```
[% USE scores = table(myscores, cols=5) %]



[% FOREACH row = scores.rows %]

  ...

[% END %]
```

You can use this approach to create multiple plugin objects with different configurations. This example shows how the format plugin is used to create subroutines bound to variables for formatting text as per *printf( )*.

```
[% USE bold = format('<b>%s</b>') %]

[% USE ital = format('<i>%s</i>') %]



[% bold('This is bold')   %]

[% ital('This is italic') %]
```

The previous example generates the following output:

```
<b>This is bold</b>

<i>This is italic</i>
```

This next example shows how the *URL* plugin can be used to build dynamic URLs from a base part and optional query parameters:

```
[% USE mycgi = URL('/cgi-bin/foo.pl', debug=1) %]

<a href="[% mycgi %]">...

<a href="[% mycgi(mode='submit') %]"...
```

The previous example generates the following output:

```
<a href="/cgi-bin/foo.pl?debug=1">...

<a href="/cgi-bin/foo.pl?mode=submit&debug=1">...
```

The *LOAD_PERL* option (disabled by default) provides a further way by which external Perl modules may be loaded. If a regular Perl module (i.e., not a Template::Plugin::* or other module relative to some PLUGIN_BASE) supports an object-oriented interface and a *new( )* constructor, it can be loaded and instantiated automatically. The following trivial example shows how the IO::File module might be used:

```
[% USE file = IO.File('/tmp/mydata') %]



[% WHILE (line = file.getline) %]

  <!-- [% line %] -->

[% END %]
```

Chapter 6 discusses plugins in excruciating detail.

4.8 Macros

Sometimes Template Toolkit code can get very complicated. You can often have complex pieces of code that get repeated a number of times throughout your template. One solution to this problem is to extract the code into another template and call it with PROCESS whenever it is needed:

```
[% PROCESS my/gnarly/code day='Monday' %]

...later...

[% PROCESS my/gnarly/code day='Tuesday' %]
```

This idea works well for larger chunks of code, but it can be a little unwieldy if used often. A far better idea is to define a macro. A macro is a piece of arbitrary Template Toolkit code that is given a name, enabling you to call it later in the template. For example:

```
[% USE date -%]

[% MACRO now GET date.format(date.now, '%H:%M:%S') -%]

[% MACRO today GET date.format(date.now, '%Y-%m-%d') -%]
```

This defines two macros called now and today that will output the current time and date whenever they are called in the template:

```
[% now %] [% today %]
```

The following subsection introduces the directive for working with macros.

4.8.1 MACRO

The MACRO directive allows you to define a directive or directive block that is evaluated each time the macro is called:

```
[% MACRO header INCLUDE header %]
```

Calling the macro as:

```
[% header %]
```

is then equivalent to:

```
[% INCLUDE header %]
```

Macros can be passed named parameters when called. These values remain local to the macro. Therefore, calling the macro as:

```
[% header(title='Hello World') %]
```

is equivalent to:

```
[% INCLUDE header title='Hello World' %]
```

A *MACRO* definition may include parameter names. Values passed to the macros are then mapped to these local variables. Other named parameters may follow these.

```
[% MACRO header(title) INCLUDE header %]


[% header('Hello World') %]

[% header('Hello World', bgcol='#123456') %]
```

There are equivalent to:

```
[% INCLUDE header title='Hello World' %]

[% INCLUDE header title='Hello World' bgcol='#123456' %]
```

Here's another example, defining a macro for display numbers in comma-delimited groups of three, using the *chunk* and *join* virtual method:

```
[% MACRO number(n) GET n.chunk(-3).join(',') %]


[% number(1234567) %]    # 1,234,567
```

A *MACRO* may precede any directive, including block directives, but must conform to the structure of the directive:

```
[%  terms = {
        sass  = 'know, be aware of, meet, have sex with',
        hoopy = 'really together guy',
        frood = 'really, amazingly together guy'
    };


    MACRO explain(term)
        IF (explanation = terms.$term);
            "$term ($explanation)";
        ELSE;
            term;
        END;
%]
```

Here we define the explain(term) macro as an IF / ELSE directive. It consults a hash table to locate an explanation for the term passed as an argument. It generates a string containing the term and explanation, or the term by itself if no explanation is found.

```
Hey you [% explain('sass') %] that
[% explain('hoopy') %] Ford Prefect?
There's a [% explain('frood') %]
who really knows where his towel is.
```

This generates the following output:

```
Hey you sass (know, be aware of, meet, have sex with) that
hoopy (really together guy) Ford Prefect?
There's a frood (really, amazingly together guy)
who really knows where his towel is.
```

A *MACRO* can also be defined as an anonymous *BLOCK*. The block will be evaluated each time the macro is called.

```
[%  MACRO translate(text)
      BLOCK;
        words = [ ];
        FOREACH word IN text.split;
          IF (explanation = terms.$word);
            words.push("$word ($explanation)");
          ELSE;
            words.push(word);
          END;
```

```
        END;

        words.join(' ');

      END

%]
```

This macro splits the text passed as an argument into words, attempts to explain them, and then joins them back up into a single piece of text:

```
[% translate(

     "Hey you sass that hoopy Ford Prefect?

      There's a frood who really knows where

      his towel is."

   )

%]
```

This is the output generated by the previous template fragment:

```
Hey you sass (know, be aware of, meet, have sex with)

that hoopy (really together guy)

Ford Prefect? There's a frood (really, amazingly together guy)

who really knows where his towel is.
```

A MACRO can also be defined as a PERL block, but will require the EVAL_PERL option to be set:

```
[% MACRO triple(n) PERL %]

     my $n = $stash->get('n');

     print $n * 3;

[% END -%]
```

The PERL and RAWPERL directives are covered at the end of this chapter.

4.9 Template Metadata

The Template Toolkit compiles a template into a Perl object (an instance of the class Template::Document). This object contains Perl code that reproduces the required behavior of the template. You can access the data in this object via the template variable.

The Template::Document has access to various items of metadata about the template that you can access via template. This always includes the name of the template and the last modification time, so it is always possible to include things such as this in your template:

```
[% USE date(format => '%Y-%m-%d %H:%M:%S') %]

[% template.name %]

Last modified: [% date.format(template.modtime) %]
```

Further metadata items can be added using the META directive, discussed next. These new items will also be available through the template variable.

```
[% META moon_phase = 'first quarter' -%]

Phase of moon: [% template.moon_phase %]
```

4.9.1 META

The *META* directive allows simple metadata items to be defined within a template. These are evaluated when the template is parsed, and as such may contain only simple values (e.g., it's not possible to interpolate other variable values into *META* variables).

```
[% META

   title   = 'The Cat in the Hat'

   author  = 'Dr. Seuss'

   version = 1.23

%]
```

The *template* variable contains a reference to the main template being processed. These metadata items may be retrieved as attributes of the template.

```
<h1>[% template.title %]</h1>

<h2>[% template.author %]</h2>
```

The *name* and *modtime* metadata items are automatically defined for each template to contain its name and modification time in seconds since the epoch:

```
[% USE date %]              # use Date plugin to format time

...

[% template.name %] last modified

at [% date.format(template.modtime) %]
```

The *PRE_PROCESS* and *POST_PROCESS* options allow common headers and footers to be added to all templates. The *template* reference is correctly defined when these templates are processed, allowing headers and footers to reference metadata items from the main template:

```
$tt = Template->new({

    PRE_PROCESS  => 'header',

    POST_PROCESS => 'footer',

});


$tt->process('cat_in_hat');
```

header:

```
<html>
<head>
<title>[% template.title %]</title>
</head>
<body>
```

cat_in_hat:

```
[% META
   title   = 'The Cat in the Hat'
   author  = 'Dr. Seuss'
   version = 1.23
   year    = 2000
%]

The cat in the hat sat on the mat.
```

footer:

```
<hr />
&copy; [% template.year %] [% template.author %]
</body>
</html>
```

The output generated from the preceeding example is:

```
<html>
<head>
<title>The Cat in the Hat</title>
</head>
<body>

The cat in the hat sat on the mat.

<hr />
&copy; 2000 Dr. Seuss
</body>
</html>
```

Chapter 4. Template Directives

Templates consist of a combination of fixed text and template directives. The template directives are recognized by the Template Toolkit and are expanded in a processor's output. In this chapter, we will take a close look at all of the directives that the Template Toolkit provides. We've already seen examples of many of them in previous chapters, but now we'll go back and fill in all of the details.

The Template Toolkit has directives for common presentation tasks. There are directives for accessing and setting variables, loading and using both external and local templates, repetition, conditional processing, flow control, and exception handling. Directives are also provided to define macros and access template metadata. If that's not enough for you, you can extend the functionality of the Template Toolkit using filters, plugins, or even inline Perl code.

5.1 Using Filters

As we have seen, a filter is used to postprocess the text from a template. The filter acts after any other template processing on the text and transforms the text before the output phase. Example 5-1 shows the format filter being used to put HTML comment characters around a piece of text.

Example 5-4. Using the format filter to add comments

```
[% text = "The white zone is for loading and unloading only." %]


[% FILTER format("<!-- %s -->");
  text;
END
%]
```

Example 5-1 generates the following output:

```
<!-- The white zone is for loading and unloading only. -->
```

Filters can be invoked in two different ways—either by enclosing a block of template markup between the FILTER and END directives, as in:

```
[% FILTER html %] ... [% END %]
```

or in side-effect notation with the FILTER coming after the item to be filtered:

```
[% text FILTER html %]
```

In the second form, the pipe symbol (|) can be used as an alias for the FILTER keyword to give a more Unix-like pipeline feel:

```
[% text | truncate(30) | format("<!-- %s -->") %]
```

As the previous example shows, a number of FILTERs can be chained together. The filters are applied from left to right.

Filters can be applied to many Template Toolkit expressions other than plain strings and scalar variables, including any block directive:

```
[% FILTER indent("> ") %]
[% INSERT "mail.txt" %]
[% END %]
```

Or, more concisely:

```
[% INSERT "mail.txt" | indent("> ") %]
```

5.2 Standard Template Toolkit Filters

The Template Toolkit comes with a large number of preinstalled filters. In this section, we will take a look at these standard filters and see examples of their usage.

5.2.1 collapse

The collapse filter replaces any amount of whitespace with a single space character. It uses Perl's definition of whitespace, which includes spaces, tabs, carriage returns, newlines, and a few more esoteric characters. Example 5-2 gives an example of using this filter.

Example 5-5. The collapse filter

```
[% FILTER collapse %]

You'll   love

  it,   it's            a    way

      of      life.

[% END %]
```

The output is nice and clean:

```
You'll love it, it's a way of life.
```

5.2.2 eval / evaltt

The eval filter evaluates the block as template text, processing any directives embedded within it. This allows template variables to contain template fragments, or for some method to be provided for returning template fragments from an external source such as a database, which can then be processed in the template as required.

```
my $vars  = {

    fragment => "The cat sat on the [% place %]",

};

$tt->process($file, $vars);

    || die $tt->error( );
```

The following example:

```
[% fragment | eval %]
```

is therefore equivalent to:

```
The cat sat on the [% place %]
```

The evaltt filter is provided as an alias for eval.

5.2.3 format(fmt)

The format filter takes a *sprintf*-style format string and applies it to the input, line by line. It can be used to preface blocks with comment markers, truncate lines, or do numeric conversions.

The *format* filter can be used for commenting out sections of text, as shown in Example 5-3.

Example 5-6. The format filter used to comment out code

```
[% FILTER format("<!-- %s -->") -%]

<script language="VBScript" type="text/vbscript">

   // evil vbscript here...

</script>

[% END %]
```

Example 5-3 produces the following output:

```
<!--     <script language="VBScript" type="text/vbscript"> -->

<!--         // evil vbscript here... -->

<!--     </script> -->
```

Because format passes its arguments to sprintf, any sprintf format strings can be used, including the field width and padding modifiers, as shown in Example 5-4.

Example 5-7. Left- and right-justified text

```
[% string = "Hello, I must be going." %]

Space padded, right justified: '[% string | format("% 32s") %]'

Space padded, left justified: '[% string | format("%- 32s") %]'
```

Example 5-4 produces the following output:

```
Space padded, right justified: '         Hello, I must be going.'

Space padded, left justified: 'Hello, I must be going.         '
```

The format filter also handles numerical transformations. Example 5-5 shows the same number being displayed in a number of different formats.

Example 5-8. Number formats

```
[% num = 42 %]

Unfiltered: [% num %]

Decimal: [% num | format("%d") %]

Binary: [% num | format("%b") %]

Hex: [% num | format("%x") %]

Hex, 0x-padded: [% num | format("%#x") %]

Octal: [% num | format("%o") %]

Octal, 0-padded: [% num | format("%#o") %]

Floating point: [% num | format("%f") %]

Scientific Notation: [% num | format("%e") %]
```

Example 5-5 produces the following output:

```
Unfiltered: 42

Decimal: 42

Binary: 101010

Hex: 2a

Hex, 0x-padded: 0x2a

Octal: 52
```

```
Octal, 0-padded: 052

Floating point: 42.000000

Scientific Notation: 4.200000e+01
```

Example 5-6 demonstrates the use of the %f format definition to control the number of decimal places displayed by a floating-point number.

Example 5-9. Controlling the number of decimal places

```
[% pi = 3.1415926536 %]

[% pi | format('%3.1f') %]

[% pi | format('%4.2f') %]

[% pi | format('%5.3f') %]
```

Its output is shown here:

```
3.1

3.14

3.142
```

Example 5-7 shows that variable interpolation works as you'd expect.

Example 5-10. Variable interpolation in format definitions

```
[% pi = 3.1415926536 %]

[% FOREACH dp = [ 1 .. 10 ] -%]

[% pi | format("%.${dp}f") %]

[% END %]
```

Here is its output:

```
3.1

3.14

3.142

3.1416

3.14159

3.141593

3.1415927

3.14159265

3.141592654

3.1415926536
```

In this example, the { } around dp is required so that the Template Toolkit knows to interpolate dp and not dpf, which is undefined (at least from the earlier snippet).

5.2.4 html

The html filter does very basic HTML encoding: it replaces the most commonly troublesome characters (<, >, &, and ") with their encoded counterparts. This is enough for many encoding jobs, and this filter is very lightweight. More complex encoding will need to use the html_entity filter, which implements a more general-purpose and extended encoding filter, but which is slower and more involved. Example 5-8 shows this filter in action.

Example 5-11. Using the html filter

```
<p>Creating an HTML anchor is simple:</p>

<pre>

[% FILTER html %]

<a href="http://www.template-toolkit.org/docs/">

  Read the documentation!

</a>

[% END %]

</pre>
```

The output from Example 5-8 is as follows:

```
<p>Creating an HTML anchor is simple:</p>

<pre>


 &lt;a href=&quot;http://www.template-toolkit.org/docs/&quot;&gt;

    Read the documentation!

 &lt;/a&gt;


</pre>
```

5.2.5 html_break / html_para_break

The html_break filter looks for sequences of two or more newlines in the text and replaces them with the HTML tag sequence

 (see Example 5-9).

Example 5-12. Using the html_break filter

```
[% FILTER html_break %]

The cat sat on the mat.



Mary had a little lamb.

[% END %]
```

This example outputs the following:

```
The cat sat on the mat.

<br />

<br />

Mary had a little lamb.
```

5.2.6 html_entity

The html filter is fast and simple, but it doesn't encode the full range of HTML entities that your text may contain. The html_entity filter uses the Apache::Util module if it can be loaded (it is written in C and is therefore faster) or the HTML::Entities module (written in Perl but equally as comprehensive) to perform the encoding. If the *Apache::Util* or the *HTML::Entities* module is installed on your system, the text will be encoded (via the escape_html or encode_entities subroutines, respectively) to convert all extended characters into their appropriate HTML entities (e.g., converting é to é). If neither module is available on your system, an html_entity exception will be thrown reporting an appropriate message.

Example 5-10 gives one example of a character that is converted to an HTML entity by this filter. The British £ symbol is converted to £.

Example 5-13. Using the html_entity filter

```
[% price = '£19.99' -%]

<p>

  The book cost [% price | html_entity %].

</p>
```

Example 5-10 produces the following output:

```
<p>

  The book cost £19.99.

</p>
```

For further information on HTML entity encoding, see http://www.w3.org/TR/REC-html40/sgml/entities.html.

5.2.7 html_line_break

The html_line_break filter replaces any newlines with
 HTML tags, thus preserving the line breaks of the original text in the HTML output. Example 5-11 shows its use.

Example 5-14. Using the html_line_break filter

```
[% FILTER html_line_break -%]

The cat sat on the mat.

Mary had a little lamb.

[% END %]
```

The example produces the following output:

```
The cat sat on the mat.<br />

Mary had a little lamb.<br />
```

5.2.8 html_para

The html_para filter formats a block of text into HTML paragraphs. A sequence of two or more newlines is used as the delimiter for paragraphs, which are then wrapped in HTML <p> ... </p> tags (see Example 5-12).

Example 5-15. Using the html_para filter

```
[% FILTER html_para -%]

The cat sat on the mat.



Mary had a little lamb.

[% END %]
```

This example produces the following output:

```
<p>

The cat sat on the mat.

</p>


<p>

Mary had a little lamb.

</p>
```

5.2.9 indent(pad)

The indent filter prefixes each line of input with a fixed string or number of spaces (defaults to four). If the supplied argument is a number, then that many spaces are used; otherwise it is taken to be a string and used literally.

This filter can be used to create bulleted lists, as shown in Example 5-13.

Example 5-16. Creating bullet points with the indent filter

```
[% FILTER indent("  * ") -%]

Item one

Item two

Item three

[%- END %]
```

Example 5-13 produces the following output:

```
* Item one

* Item two

* Item three
```

This filter also can be used to quote emails, as shown in Example 5-14.

Example 5-17. Quoting emails with the indent filter

```
[% quote = "> " %]

[% FILTER indent(quote) -%]

Dear Darren, Dave, and Andy,


You guys rock. The Template Toolkit book is fantastic.


Thanks for writing it.


A Fan

[% END %]
```

Example 5-14 produces the following output:

```
> Dear Darren, Dave, and Andy,

>

> You guys rock. The Template Toolkit book is fantastic.
```

```
>

> Thanks for writing it.

>

> A Fan
```

It also can be used to add a prefix to debugging messages, as shown in Example 5-15.

Example 5-18. Adding the template name to debug output

```
[% debug_msg | indent("[$template.name] ") | stderr %]
```

This example produces the following output:

```
[src/header] Some useful debug info (which goes to stderr)
```

If you give the indent filter no arguments, it indents by four spaces, as shown in Example 5-16.

Example 5-19. Default indent

```
[% FILTER indent -%]

A sample piece of text

that will be indented

[%- END %]

This isn't indented
```

Example 5-16 produces the following output:

```
    A sample piece of text

    that will be indented

This isn't indented
```

5.2.10 latex(outputType)

The latex filter passes the text block to LaTeX[2] and produces either PDF, DVI, or PostScript output. The outputType argument determines the output format, and it should be set to one of the following strings: "pdf" (default), "dvi", or "ps".

[2] If you have it installed on your system.

The text block should be a complete LaTeX source file. Example 5-17 shows the latex filter in action.

Example 5-20. Using the latex filter

```
[% FILTER latex("pdf") -%]

\documentclass{article}


\begin{document}


\title{A Sample TT2 \LaTeX\ Source File}

\author{Craig Barratt}

\maketitle
```

```
\section{Introduction}

This is some text.


\end{document}

[% END -%]
```

The output will be a PDF file. You should be careful not to prepend or append any extraneous characters or text outside the FILTER block because this text will wrap the (binary) output of the latex filter. Notice the - character placed before the %] end tag to remove the trailing newline.

One instance in which you might prepend text is in a CGI script, where you might include the Content-Type before the latex output, as shown in Example 5-18.

Example 5-21. Using the latex filter in a CGI program

```
Content-Type: application/pdf


[% FILTER latex("pdf") -%]

\documentclass{article}

\begin{document}

...

\end{document}

[% END -%]
```

In other cases, you might use the redirect filter to put the output into a file, rather than delivering it to STDOUT. This might be suitable for batch scripts, as shown in Example 5-19.

Example 5-22. Redirecting output from the latex filter

```
[% output = FILTER latex("pdf") -%]

\documentclass{article}

\begin{document}

...

\end{document}

[% END; output | redirect("document.pdf", 1) -%]
```

(Notice the second argument to redirect to force binary mode.)

The *latex* filter runs one or two external programs, so it isn't very fast. But for modest documents, the performance is adequate, even for interactive applications.

An error of type latex will be thrown if an error is reported by *latex*, *pdflatex*, or *dvips*.

5.2.11 lcfirst

The lcfirst filter folds the first character of the input to lowercase, as shown in Example 5-20.

Example 5-23. Using the lcfirst filter

```
[% "FIREHOSE" FILTER lcfirst %]
```

Example 5-20 produces the following output:

```
fIREHOSE
```

The lcfirst filter can be chained to the upper, ucfirst, and lower filters (described later in this chapter). In Example 5-21 the first letter of the sentence is folded to uppercase, with the remaining letters folded to lowercase.

Example 5-24. Combining the lower and ucfirst filters

```
[% sentence = "sOmE tExT iN rAnDoM cAsE" -%]

[% sentence | lower | ucfirst %]
```

Example 5-21 produces the following output:

```
Some text in random case
```

This sequence of filters would make a very useful macro, as shown in Example 5-22.

Example 5-25. The sentence_case macro

```
[% MACRO sentence_case(str) str | lower | ucfirst %]
```

The upper, lower, ucfirst, and lcfirst filters are subject to Perl's normal locale considerations. The perllocale documentation, which came with your copy of Perl, has all the details.

5.2.12 lower

The lower filter folds all the characters in the input text to lowercase (see Example 5-23).

Example 5-26. Using the lower filter

```
[% "Hello World" | lower %]
```

Example 5-23 produces the following output:

```
hello world
```

5.2.13 null

The null filter prints nothing. This is useful for plugins whose methods return values that you don't want to appear in the output. You can use CALL on each plugin method call to ignore the value returned, or you can wrap the block in a *null* filter (see Example 5-24).

Example 5-27. Using the null filter

```
[% FILTER null;

  USE im = GD.Image(100,100);

  black = im.colorAllocate(0,   0, 0);

  red   = im.colorAllocate(255,0,  0);

  blue  = im.colorAllocate(0,  0,  255);

  im.arc(50,50,95,75,0,360,blue);

  im.fill(50,50,red);

  im.png | stdout(1);

END;

-%]
```

Notice the use of the stdout filter to ensure that a particular expression generates output to STDOUT (in this case, in binary mode).

5.2.14 perl / evalperl

The perl filter evaluates the block as Perl code. The EVAL_PERL option must be set to a true value or a perl exception will be thrown (see Example 5-25).

Example 5-28. Using the perl filter

```
[% my_perl_code | perl %]
```

In most cases, the PERL ... END directive block should suffice for evaluating Perl code. Thus, Example 5-25 could have been written in the more verbose forms shown in Example 5-27.

Example 5-29. Using a PERL block in place of the perl filter

```
[% PERL %]

[% my_perl_code %]

[% END %]
```

Example 5-30. Using the perl filter in block form

```
[% FILTER perl %]

[% my_perl_code %]

[% END %]
```

The evalperl filter is provided as an alias for perl for backward compatibility.

5.2.15 redirect(file, options)

The redirect filter redirects the output of the block to the named file, relative to a location defined in the OUTPUT_PATH configuration option.

The redirect( ) filter will throw a file exception if the file specified cannot be opened. The filter should be used in a TRY ... CATCH block if you want to trap these kind of errors (see Example 5-28).

Example 5-31. Using the redirect filter

```
[% USE translate("src" = "en");

 FOREACH language = languages;

  file = "index.html.$language";

  TRY;

   text | $translate("dest" => language) | redirect(file);

   msg  = " + Successfully translated $file to $language.";

  CATCH file;

   msg = " - Cannot open $file: $error";

  CATCH;

   msg = " - Error: $error";

  END;

  emsg | stderr;

 END;

%]
```

5.2.16 remove(string)

The remove filter removes parts of the text block, based on the regular expression specified by the string. The regular expression is passed directly to Perl, and can contain anything regular Perl regexes can contain. Example 5-29 removes every occurence of the letter "e" from a string:

Example 5-32. Using the remove filter

```
[% string = "Hello, I must be going.";
   string | remove("e") %]
```

Example 5-29 produces the following output:

```
Hllo, I must b going.
```

Example 5-30 shows a more complex example that removes all occurences of "e" preceeded by an "H" and followed by "ll", but without removing the "H" or "ll". It combines a zero-width positive lookbehind assertion (?<=) with a zero-width positive lookahead assertion (?=).

Example 5-33. Using the remove filter with a regular expression

```
[% string = "Hello, I must be going.";
   string | remove("(?x)   # whitespace is not important
     (?<=H)              # an 'H'
       e                 # strip the 'e'!
     (?=ll)              # followed by 'll'
") %]
```

Example 5-30 produces the following output:

```
Hllo, I must be going.
```

5.2.17 repeat(iterations)

The repeat filter repeats the text iteration number of times. The default for iterations is 1 and the text is printed only once (see Example 5-31).

Example 5-34. Using the repeat filter

```
[% FILTER repeat(5) %]
All work and no play make Jack a dull boy.
[% END %]
```

Example 5-31 produces the following output:

```
All work and no play make Jack a dull boy.

All work and no play make Jack a dull boy.

All work and no play make Jack a dull boy.

All work and no play make Jack a dull boy.
```

```
All work and no play make Jack a dull boy.
```

5.2.18 replace(search, replace)

The replace filter is similar to the remove filter, but also takes a replacement string. Example 5-32 replaces every "e" in the input text with a "u".

Example 5-35. Using the replace filter

```
[% string = "Hello, I must be going.";

   string | replace("e", "u") %]
```

Example 5-32 produces the following output:

```
Hullo, I must bu going.
```

5.2.19 stderr

The stderr filter, shown in Example 5-33, prints the input text to STDERR. The binmode argument can be used as described in the stdout filter, explained next.

Example 5-36. Using the stderr filter

```
[% PROCESS something/cool | stderr(binmode=1) %]
```

5.2.20 stdout(options)

The stdout filter prints the output generated by the enclosing block to STDOUT. Currently, the only supported option is binmode, which can be passed as either a named parameter or a single argument to set STDOUT to binary mode (see Example 5-34).

Example 5-37. Using the stdout filter

```
[% PROCESS something/cool

   FILTER stdout(binmode=1) # recommended %]



[% PROCESS something/cool

   FILTER stdout(1)        # alternate %]
```

Setting binmode is mostly of use for Win32 and VMS users; see the *perlfunc(1)* manpage for all the gory details.

The stdout filter can be used to force binmode on STDOUT, or inside redirect, null, or stderr blocks to make sure that particular output goes to standard output. See Example 5-24 earlier in this chapter for an example of this usage.

5.2.21 trim

The trim filter removes any leading and trailing whitespace from the input text. Example 5-35 shows a string with leading and trailing whitespace, both of which are removed when passed through the trim filter.

Example 5-38. Using the trim filter

```
[% text = "  some text with leading and trailing spaces   " %]

+[% text | trim %]+
```

Example 5-35 produces the following output:

```
+some text with leading and trailing spaces+
```

This filter is particularly useful when working with BLOCK definitions. In Example 5-36, the foo block will be defined as *\nLine 1 of foo\n*. The surrounding newlines will also be introduced whenever the template is loaded using INCLUDE or PROCESS.

Example 5-39. Extra newlines when processing blocks

```
[% BLOCK foo %]
between
[% END %]

before-[% PROCESS foo %]-after
```

Example 5-36 produces the following output:

```
before-
between
-after
```

When run through the trim filter, leading and trailing newlines (which count as whitespace) will be removed from the output of the BLOCK (see Example 5-37).

Example 5-40. Using the trim filter to remove the extra newlines

```
[% BLOCK foo %]
between
[% END %]

before-[% PROCESS foo | trim %]-after
```

Example 5-37 produces the following output:

```
before-between-after
```

5.2.22 truncate(length)

The truncate filter returns the first length characters of the input text. The default value for length is 32. The text will actually be truncated three characters short of this, to make room for an ellipsis (. . .) to be appended to it. The returned text will be exactly length characters long, or less.

Example 5-38 shows it being used in a search results page.

Example 5-41. Using the truncate filter

```
[% FOREACH result = results %]
  * [% result.description | truncate(24) %]
   <a href="[% result.link %]">Read more</a>
[% END %]
```

When using truncate from within HTML, there is a danger that simply truncating the text will leave hanging HTML tags, as demonstrated in Example 5-39.

Example 5-42. Hanging HTML tags

```
[%- result.description = "Hello, <blink>world</blink>!" %]

Description: [% result.description | truncate(20) %]
```

Example 5-39 produces the following output:

```
Description: Hello, <blink>wor...
```

Using the remove filter in conjunction with the truncate filter, we get the desired results, as shown in Example 5-40.

Example 5-43. Using the remove filter to fix the hanging HTML tags

```
[%- result.description = "Hello, <blink>world</blink>!" %]

Description: [% result.description | remove("<[^>]*?>") | truncate(20) %]
```

Example 5-40 produces the following output:

```
Description: Hello, world!
```

5.2.23 ucfirst

The ucfirst filter folds the first character of the input to uppercase, as shown in Example 5-41.

Example 5-44. Using the ucfirst filter

```
[% "hello" | ucfirst %]
```

Example 5-41 produces the following output:

```
Hello
```

5.2.24 upper

The upper filter uppercases the input, similar to Perl's uc function (see Example 5-42).

Example 5-45. Using the upper filter

```
[% 'do not leave it is not real' | upper %]
```

Example 5-42 produces the following output:

```
DO NOT LEAVE IT IS NOT REAL
```

5.2.25 uri

The uri filter performs *URI-escaping*, which is the transformation of a URI string into a specific set of characters that are guaranteed not to cause any clients to do funny things. As defined by RFC 2396, a URI may consist of a limited number of "safe" characters; all others must be escaped using hexadecimal equivalents in the format %nn, where nn is the hex number that represents the ASCII code for the character. This is demonstrated in Example 5-43.

Example 5-46. Using the uri filter

```
[% filename = 'C:\My Documents\My Web Page.html' %]

Visit <a href="[% filename | uri %]">My Web Page</a>!
```

Example 5-43 produces the following output:

```
Visit <a href="C:%5CMy%20Documents%5CMy%20Web%20Page.html">My Web Page</a>!
```

Escaping a URI that doesn't need it cannot hurt, although escaping a URI that has already been escaped can lead to bugs that are difficult to track down. For example, the % character by itself is always escaped because it marks the beginning of an escaped sequence. Because an escaped URI is not necessarily HTML-safe, many URIs will also need to be passed through the html filter. A good rule of thumb is to escape anything that might need escaping immediately, as shown in Example 5-44.

Example 5-47. Using the uri filter with the html filter

```
[% url = "this page.cgi";
  prev = "$url?page=1&search=1" | uri | html;
  next = "$url?page=3&search=1" | uri | html;
%]
<a href="[% prev %]">Previous</a>
<a href="[% next %]">Next</a>
```

Example 5-44 produces the following output:

```
<a href="this%20page.cgi?page=1&amp;search=1">Previous</a>
<a href="this%20page.cgi?page=3&amp;search=1">Next</a>
```

For more information about URI escaping, see RFC 2396 and 2732.

Chapter 5. Filters

Filters are a powerful feature of the Template Toolkit that allow you to postprocess parts of the output of your template in many different ways. A number of filters for carrying out common tasks are included with the standard Template Toolkit distribution, and it is possible to extend this set by writing your own.

A good example of a filter that comes with the Template Toolkit is the html filter. In an HTML document, a number of characters have special meanings, so if you want these characters to appear in your document they need to be converted to *HTML Entities*. The html filter converts the characters <, >, ", and & to <, >, ", and &, respectively.[1]

[1] There is also another filter called html_entity, which converts far more characters.

Example 5-1 shows the html filter in action. Without the filter, the JavaScript section in the example would be treated as actual JavaScript code and executed. The filter converts the < characters, thereby changing the JavaScript to text that would be displayed by a browser rather than being executed.

Example 5-1. Filtering Javascript

```
<p>Here is what the JavaScript should look like:</p>

<pre>

[% FILTER html %]

<script language="JavaScript" type="text/javascript">

<!--

document.writeln("Hello, world");

//-->

</script>

[% END %]

</pre>
```

The processed document looks like this:

```
<p>Here is what the JavaScript should look like:</p>

<pre>



&lt;script language=&quot;JavaScript&quot; type=&quot;text/javascript&quot;&gt;

&lt;!--

    document.writeln(&quot;Hello, world&quot;);

//--&gt;

&lt;/script&gt;

</pre>
```

This example also demonstrates a good reason for using filters. The kinds of transformations that a filter makes might well be appropriate only for a particular output medium. For example, the html filter will be used only on HTML documents that are being sent to a browser. If you were printing out the document for some reason, the html filter would only make it harder to follow. Having the *FILTER* functionality available as a postprocessing option makes it easy to decide whether to use it in certain circumstances, and easy to add it to certain parts of a template without changing the way that most of the template works.

In Example 5-1, we used the block syntax for using the *FILTER* directive. This is useful for filtering large parts of a template. If you are filtering the output from a single tag, there is an inline version of the syntax, as shown in Example 5-2.

Example 5-2. Formatting numbers

```
[% pi = 3.1415926536;

  pi FILTER format('%0.3f')

%]
```

This example uses the format filter, which reformats data using format definitions such as those used by the printf function common in many programming languages. In the example, we reformat a decimal number to display only two decimal places (note also that the last digit displayed is rounded up).

The processed output looks like this:

```
3.142
```

It is possible to abbreviate this even further. The pipe character (|) can be used as a synonym for FILTER, as shown in Example 5-3.

Example 5-3. Filtering using the pipe symbol

```
[% pi = 3.1415926536;

  pi | format('%0.3f')

%]
```

These two examples also demonstrate the differences between the two types of filters. The html filter is an example of a *static filter*, whereby the filter has the same effect each time it is used. The format filter is an example of a *dynamic filter*, whereby the exact transformation is controlled by a parameter that is passed to the filter on each use.

In this chapter, we look at the different ways you can use filters in your own templates, and also look at the standard filters that are part of the Template Toolkit.

6.1 Using Plugins

As we saw in Chapter 2, using plugins from a template is done with the USE directive:

```
[% USE date %]
```

This makes a date plugin object available to the template, which can be used by referencing the variable date. Many plugins accept arguments as part of the USE directive, to control the initial configuration. For example, to tell the date plugin to use GMT as the default time zone, instead of the local time zone, you would use:

```
[% USE date(gmt = 1) %]
```

Once a plugin has been initialized, it can be treated like any other variable:

```
Today is [% date.format %].
```

The preceeding example might return:

```
Today is 09:31:55 11-Aug-2003.
```

A plugin reference can be optionally assigned to a variable:

```
[% USE today = date %]
```

and accessed as today, rather than date. This has the potential to make for less confusing templates, but, more importantly, it means that you can have multiple instances of a plugin in the same template:

```
[% USE here = Directory '.' %]

[% USE there = Directory '/etc' %]
```

The Template Toolkit ships with a large number of useful, general-purpose plugins, which we will examine here, and provides a supporting framework for creating your own plugins (see Chapter 8).

Many of the standard plugins are Template Toolkit wrappers around general-purpose modules. In order to use these plugins, the wrapped module must be installed. The general installation techniques discussed in this chapter are applicable for all CPAN modules; in particular, the CPAN shell is very useful, as it will decline to reinstall modules that are up-do-date, and can be used to automatically fetch new versions from your favorite CPAN mirror.

In addition to the standard plugins, a number of plugins are available on CPAN, at http://www.cpan.org/modules/by-module/Template.

6.2 Standard Template Toolkit Plugins

As of Version 2.10, the Template Toolkit ships with a large number of plugins. The functionality these plugins add varies from trivial helper wrappers to full-blown reformatting utilities.

Some of these plugins are of interest only to developers, such as the Template::Plugin::Procedural and Template::Plugin::Filter; these will not be covered here (see Chapter 8 for treatment of these).

6.2.1 Autoformat

The Autoformat plugin provides an interface to Damian Conway's Text::Autoformat Perl module, which provides advanced text wrapping and formatting. Text::Autoformat is designed to be intelligent about wrapping lines; in addition to doing basic text wrapping, it can handle unusual text, such as mail or news text with quoting, or text with bullets or numbering. The Autoformat plugin provides a simple plugin/filter interface to the module.

Configuration options may be passed to the plugin constructor via the USE directive:

```
[% USE autoformat %]
```

The Autoformat plugin can then be called like a function, passing in text items that will be wrapped and formatted according to the current configuration (see Example 6-1).

Example 6-1. Autoformatting a Martin Gardner quote

```
[% USE autoformat right = 42 %]
[% autoformat('
Biographical history, as taught in our public schools, is still
largely a history of boneheads:  ridiculous kings and queens, paranoid
political leaders, compulsive voyagers, ignorant generals -- the
flotsam and jetsam of historical currents.  The men who radically
altered history, the great scientists and mathematicians, are seldom
mentioned, if at all.


    -- Martin Gardner
')
%]
```

Output of Example 6-1:

```
Biographical history, as taught in our
public schools, is still largely a history
of boneheads: ridiculous kings and queens,
paranoid political leaders, compulsive
voyagers, ignorant generals -- the flotsam
and jetsam of historical currents. The men
who radically altered history, the great
```

```
scientists and mathematicians, are seldom

mentioned, if at all.


    -- Martin Gardner
```

Additional configuration items can be passed to the autoformat subroutine and will be merged with any existing configuration specified via the constructor.

In addition to the functional interface, the Autoformat plugin also provides a filter interface, which works identically, as shown in Example 6-2.

Example 6-2. Using autoformat in filter mode

```
[% FILTER autoformat justify = 'center' -%]

Programming is a Dark Art, and it will always be. The programmer is

fighting against the two most destructive forces in the universe:

entropy and human stupidity. They're not things you can always

overcome with a "methodology" or on a schedule.


    -- Damian Conway

[% END %]
```

Output of Example 6-2:

```
      Programming is a Dark Art, and it will

       always be. The programmer is fighting

    against the two most destructive forces in

    the universe: entropy and human stupidity.

    They're not things you can always overcome

      with a "methodology" or on a schedule.


                  -- Damian Conway
```

Configuration options are passed directly to Text::Autoformat; see the Text::Autoformat documentation for all of the available options.

The Text::Autoformat module is available from CPAN at http://search.cpan.org/dist/Text-Autoformat/.

6.2.2 CGI

The CGI plugin is a wrapper around Lincoln Stein's CGI module, which is included with Perl. CGI provides a simple way of interacting with form parameters and cookies without having to understand the messy details of the CGI interface.

The CGI plugin provides access to all of CGI's functionality, including parameter and cookie support, access to file uploads, and access to HTML generation methods.

All the usual methods of the CGI module are available when using the CGI plugin, including the ever-popular param:

```
[% USE q = CGI %]


Hello, [% q.param('name') OR 'Mr. Unnamed' %]!
```

When called without an argument, param returns a list of all the defined parameter names, which can then be iterated over in a FOREACH loop:

```
[% FOREACH param IN q.param %]
   [% param %] -> [% q.param(param) %]
[% END %]
```

The plugin adds another method, params, that returns all CGI parameters as a hash:

```
[% params = q.params;
   IF params.exists('story_id');
      PROCESS story id = params.story_id;
   END;
%]
```

This hash can be used like any other hash. For example, to import this hash so that the parameters can be accessed directly, use import:[1]

[1] This takes advantage of the fact that the *stash* is a hash; see Chapter 8 for an explanation of why this works.

```
[% USE q = CGI('uid=18&name=Dave+Cross&nick=davorg') %]
[% params = q.params %]
[% import(params) %]

UID: [% uid %]
Nick: [% nick %]
Name: [% name %]
```

Without calling import, these variables would have to be qualified:

```
UID: [% params.uid %]
Nick: [% params.nick %]
Name: [% params.name # or q.param('name') -- same thing %]
```

Cookies are available via the aptly named cookie method:

```
[% SessionID = q.cookie('SessionID') %]
```

The CGI module's HTML generation methods work as expected, for the most part:

```
[% q.start_ol;
  FOREACH param IN q.param;
    q.start_li;
    q.start_b;
    param;
    q.end_b;
    ": ";
    q.param(param);
    q.end_li;
  END;
    q.end_ol;
%]
```

CGI methods that return a list, such as checkbox_group, need to be explicitly joined into a string (using the join vmethod, for example), or iterated over (using a FOREACH loop). Otherwise, the unsightly (and most likely unintended!) stringified array reference will be the result, as shown in Example 6-3.

Example 6-3. Stringified array

```
[% USE  q = CGI %]

[% q.checkbox_group(name = 'modules'

                label = 'Modules to install'

                values = [ 'Template-Toolkit',

                           'DBD::Google',

                           'Calendar::Simple'

                         ])

%]
```

Output of Example 6-3:

```
ARRAY(0x859eab4)
```

When joined with the join vmethod, the results are a little more natural, as shown in Example 6-4.

Example 6-4. Joined array

```
[% USE q = CGI %]

[% q.checkbox_group(name = 'modules'

                label = 'Modules to install'

                values = [ 'Template-Toolkit',

                           'DBD::Google',

                           'Calendar::Simple'

                         ]).join("\n")

%]
```

Output of Example 6-4:

```
<input type="checkbox" name="modules" value="Template-Toolkit" label="Modules to install" />Template-Toolkit

<input type="checkbox" name="modules" value="DBD::Google" label="Modules to install" />DBD::Google

<input type="checkbox" name="modules" value="Calendar::Simple" label="Modules to install" />Calendar::Simple
```

The CGI module is available with all recent versions of Perl, or from CPAN at http://search.cpan.org/dist/CGI/.

6.2.3 Datafile

The Datafile plugin provides a simple interface to tabular file-based data, such as Comma Separated Value (CSV) files. It provides a simple facility to construct a list of hashes, each of which represents a data record of known structure, from the datafile.

Pass a file to USE:

```
[% USE datafile(filename, delim = ':') %]
```

The file specified by filename will be read and split on delim into an array of hashes. delim is optional, and defaults to :. Currently, no INCLUDE_PATH search is performed for the file, so an absolute path should be used (this may change in a future version of the plugin, however).

delim can be used to specify an alternate delimiter character, such as the Tab or comma keys:

```
[% USE machines = datafile('machine-list.txt', delim = ",") %]
```

The format of the file is intentionally simple. The first line defines the field names, delimited by $delim with optional surrounding whitespace. Subsequent lines then define records containing data items, also delimited by $delim.

The first line of the file contains the field definitions. Blank lines and lines beginning with the comment character (#) will be ignored.

Each line is read, split into composite fields, and then used to initialize a hash array containing the field names as relevant keys.

The Datafile plugin is ideal for mostly static data that may need to be reused in many places—for example, storing information about computers, as shown in the following datafile called *machine-list.txt*:

```
name,     os,          ip
apollo,   RedHat 7.3,  10.100.5.100
hera,     RedHat 7.2,  10.100.33.227
juno,     Solaris 8,   10.100.6.41
artemis,  RedHat 7.3,  10.100.6.42
hermes,   Solaris 9,   10.100.55.182
zeus,     RedHat 7.3,  10.100.6.78
```

Creating reports from this datafile is very simple, as Example 6-5 shows.

Example 6-5. Turning machine-list.txt into XML

```
[% USE machines = datafile('example/machine-list.txt',
                           delim = ',') -%]
<machines>
[% FOREACH machine IN machines.sort('name') -%]
  <machine name="[% machine.name %]"
           os="[% machine.os %]"
           ip="[% machine.ip %]" />
[% END -%]
</machines>
```

When Example 6-5 is run, we get:

```
<machines>
  <machine name="apollo"
           os="RedHat 7.3"
           ip="10.100.5.100" />
  <machine name="artemis"
           os="RedHat 7.3"
           ip="10.100.6.42" />
  <machine name="hera"
           os="RedHat 7.2"
           ip="10.100.33.227" />
  <machine name="hermes"
           os="Solaris 9"
           ip="10.100.55.182" />
  <machine name="juno"
           os="Solaris 8"
```

```
        ip="10.100.6.41" />
  <machine name="zeus"
        os="RedHat 7.3"
        ip="10.100.6.78" />
</machines>
```

6.2.4 Date

The Date plugin provides an easy way to manipulate dates and times, including generating formatted dates and times based on the formats defined by your system's strftime library (see the sidebar). The Date plugin also provides methods to perform date calculations using Date::Calc, and to perform general date manipulations using Date::Manip. (These modules, which are available from CPAN, must be installed in order to use this functionality. The rest of the plugin will work just fine without them, though.)

strftime

strftime is a system library function that returns a formatted date according to a *format string*. These format strings are a sort of templating system on their own—they contain plain text and format strings (which begin with %). These format strings are like the Template Toolkit's variables, and are replaced with the appropriate values. The supported format strings vary from system to system, but they all support the same basic subset, a summary of which follows:

%a The abbreviated weekday name.

%A The full weekday name.

%b The abbreviated month name.

%B The full month name.

%d The day of the month as a decimal number (range 01 to 31).

%H The hour as a decimal number using a 24-hour clock (range 00 to 23).

%I The hour as a decimal number using a 12-hour clock (range 01 to 12).

%j The day of the year as a decimal number (range 001 to 366).

%m The month as a decimal number (range 01 to 12).

%M The minute as a decimal number (range 00 to 59).

```
%p Either "AM" or "PM" according to the given time value,

      or the corresponding  strings  for  the current locale.

Noon is treated as "pm" and midnight as "am".

%S The second as a decimal number (range 00 to 59).

%w The day of the week as a decimal, range 0 to 6, Sunday being 0.

%Y The year as a decimal number, including the century.

%Z     The time zone, name, or abbreviation.
```

The plugin provides the format method, which accepts a time value, a format string, and a locale name. All of these parameters are optional with the current system time, default format (%H:%M:%S %d-%b-%Y), and current locale being used, respectively, if undefined. Default values for the time, format, and/or locale may be specified as named parameters in the USE directive:

```
[% USE date(format = '%Y/%m/%d'

          locale = 'fr_FR')

%]
```

When called without any parameters, the format method returns a string representing the current system time, formatted by strftime according to the default format and for the default locale (which may not be the current one, if locale is set in the USE directive):

```
[% date.format %]
```

The plugin allows a time/date to be specified as seconds since the epoch, as is returned by time:

```
File last modified: [% date.format(template.modtime) %]
```

The time/date can also be specified as a string of the form h:m:s d/m/y. A space or any of the characters :, /, or -, may be used to delimit fields:

```
[% USE day = date(format = '%A' locale = 'en_GB') %]

[% day.format('09:31:56 11-08-2003') %]
```

The previous code generates the following output:

```
Monday
```

A format string can also be passed to the format method, and a locale specification may follow that:

[% date.format(template.modtime, '%d-%b-%Y') %]

[% date.format(template.modtime, '%d-%b-%Y', 'en_GB') %]

A fourth parameter allows you to force output in GMT, in the case of seconds-since-the-epoch input:

[% date.format(template.modtime, '%d-%b-%Y', 'en_GB', 1) %]

Any or all of these parameters may be named. Positional parameters should always be in the order ($time, $format, $locale):

[% date.format(format = '%H:%M:%S') %]

[% date.format(time = template.modtime format = '%H:%M:%S') %]

[% date.format(mytime format = '%H:%M:%S') %]

[% date.format(mytime format = '%H:%M:%S' locale = 'fr_FR') %]

[% date.format(mytime format = '%H:%M:%S' gmt = 1) %]

The now method returns the current system time in seconds since the epoch:

[% date.format(date.now, '%A') %]

It has been [% date.now - template.modtime %] seconds since

[% template.name %] was last modified.

The calc method can be used to create an interface to the Date::Calc module (if installed on your system):

[% calc = date.calc %]

[% calc.Monday_of_Week(22, 2001).join('/') %]

Date::Calc provides a number of useful date-related methods, including date math (adding dates together, for example).

The manip method can be used to create an interface to the Date::Manip module (if installed on your system):

[% USE q = CGI %]

[% manip = date.manip %]

[% time = manip.UnixDate(q.param('date'), "%s") %]

[% date.format(time) %]

See the strftime sidebar for details about common format strings. Many versions of strftime, most notably GNU strftime, include more format strings, so check your system's manpages for the complete story.

Date::Calc is available from CPAN at http://search.cpan.org/dist/Date-Calc/. Date::Manip is also available from CPAN, at http://search.cpan.org/dist/Date-Manip/.

6.2.5 Directory

The Directory plugin provides a simple interface to a directory and the files within it. It provides methods for iterating over all the contained files and subdirectories. This plugin is in cahoots with the File plugin, and in fact uses instances of the File plugin to represent files within a directory (all the methods available to the File plugin are also available here, such as uid and mtime). Subdirectories within a directory are represented by further instances of this plugin.

The Directory plugin can be used to create an instance with a directory name as an argument:

[% USE dir = Directory '/tmp' %]

It then provides access to the files and subdirectories contained within the directory via the files and dirs methods, respectively:

regular files (not directories)

[% FOREACH file = dir.files %]

    [% file.name %]

```
[% END %]


# directories only

[% FOREACH file = dir.dirs %]

   [% file.name %]

[% END %]


# files and/or directories

[% FOREACH file = dir.list %]

   [% file.name %] ([% file.isdir ? 'directory' : 'file' %])

[% END %]
```

The plugin constructor will throw a Directory error if the specified path does not exist or is not a directory, or if there is some other error at the operating system level (such as NFS problems). Otherwise, it will scan the directory and create lists named files, containing files, dirs, containing directories, and list, containing both files and directories combined. The nostat option can be set to disable all file/directory checks and directory scanning; this speeds up the process of loading the plugin for large directories:

```
[% USE etc = directory '/etc/' nostat = 1 %]
```

Each file in the directory will be represented by an instance of the File plugin, and each directory will be represented by another Directory plugin. If the recurse flag is set, those directories will contain further nested entries, and so on. With the recurse flag unset, as it is by default, each is just a place marker for the directory and does not contain any further content unless its scan method is explicitly called. The isdir flag can be tested against files and/or directories, returning true if the item is a directory or false if it is a regular file:

```
[% FOREACH file = dir.list %]

   [% IF file.isdir %]

     * Directory: [% file.name %]

   [% ELSE %]

     * File: [% file.name %]

   [% END %]

[% END %]
```

6.2.6 DBI

The DBI plugin provides a template-level interface to Tim Bunce's DBI module. The DBI module provides a uniform database interface, and the DBI plugin ensures that it plays nicely with the Template Toolkit. The DBI plugin is covered extensively in Chapter 9.

6.2.7 Dumper

The Dumper plugin provides an interface to the Data::Dumper module. Data::Dumper will convert a complex variable into a human-readable structure.

The Dumper plugin provides the dump method, which is extremely useful for displaying the structure of a variable (see Example 6-6).

Example 6-6. Dumping a hash

```
[% USE dumper %]

[% terms = {

      sass  = 'know, be aware of, meet, have sex with'
```

```
    hoopy = 'really together guy'

    frood = 'really, amazingly together guy'

  } %]

[% dumper.dump(terms) %]
```

Coming out, terms looks almost exactly like it did going in, except for the order:[2]

> [2] Perl's hashes are not stored in the order in which they are inserted, but rather in an order optimized for fast lookup by name. This is called *hash order*, and Data::Dumper doesn't attempt to reorder the keys of a hash as it dumps them.

```
$VAR1 = {

     'hoopy' => 'really together guy',

     'frood' => 'really, amazingly together guy',

     'sass' => 'know, be aware of, meet, have sex with'

    };
```

Although the Dumper plugin is not so useful for a variable we've defined ourselves, it is much more useful for data structures that you don't have direct control over, as Example 6-7 shows.

Example 6-7. Dumping the CGI plugin

```
[% USE CGI %]

[% USE dumper %]

[% dumper.dump(CGI) %]
```

Output of Example 6-7:

```
$VAR1 = bless( {

        '.charset' => 'ISO-8859-1',

        '.parameters' => [  ],

        '.fieldnames' => {  },

        'escape' => 1

       }, 'CGI' );
```

The dump_html method takes the output of dump and formats it for HTML. Example 6-7 is the same as Example 6-8, except for the call to dump_html:

Example 6-8. Dumping the CGI plugin with dump_html

```
[% USE CGI %]

[% USE dumper %]

[% dumper.dump_html(CGI) %]
```

The output is very similar:

```
$VAR1 = bless( {<br>

        '.charset' =&gt; 'ISO-8859-1',<br>

        '.parameters' =&gt; [  ],<br>

        '.fieldnames' =&gt; {  },<br>

        'escape' =&gt; 1<br>

       }, 'CGI' );<br>
```

The Data::Dumper Pad, Indent, and Varname options are supported as constructor arguments to affect the output generated. Example 6-9 shows all the details.

Example 6-9. Modifying Data::Dumper's output

```
[% USE CGI %]

[% USE dumper(Pad = '// ', Varname = 'CGI') %]

[% dumper.dump(CGI) %]
```

Output of Example 6-9:

```
// $CGI1 = bless( {
//              '.charset' => 'ISO-8859-1',
//              '.parameters' => [  ],
//              '.fieldnames' => {  },
//              'escape' => 1
//            }, 'CGI' );
```

Data::Dumper comes with all recent versions of Perl, and is also available from CPAN at http://search.cpan.org/dist/Data-Dumper/.

6.2.8 File

This plugin provides an abstraction of a file. It can be used to fetch details about files from the filesystem, or to represent abstract files (e.g., when creating an index page) that may or may not exist on a filesystem.

A filename or path should be specified as a constructor argument:

```
[% USE file 'foo.html' %]

[% USE file 'foo/bar/baz.html' %]

[% USE file '/foo/bar/baz.html' nostat = 1 %]
```

The file should exist on the current filesystem (unless the nostat option is set, which we discuss in a bit) as an absolute file when specified with a leading / as per */foo/bar/baz.html*, or otherwise as one relative to the current working directory. The constructor performs a stat on the file and makes the 13 elements returned available as the plugin items:

```
dev ino mode nlink uid gid rdev size

atime mtime ctime blksize blocks
```

For example:

```
[% USE baz = File '/foo/bar/baz.html' %]



[% baz.mtime %]

[% baz.mode %]
```

In addition, the user and group items are set to contain the user and group names as returned by calls to getpwuid and getgrgid for the file uid and gid elements, respectively (see Example 6-10). On Win32 platforms on which getpwuid and getgrid are not available, these values are undefined.

Example 6-10. user and uid

```
[% USE Makefile = file 'Makefile'  %]

uid: [% Makefile.uid %]

user: [% Makefile.user %]
```

Output of Example 6-10:

```
uid: 500

user: darren
```

This user/group lookup can be disabled by setting the noid option, as shown in Example 6-11.

Example 6-11. noid = 1

```
[% USE Makefile = file 'Makefile' noid = 1 %]

uid: [% Makefile.uid %]

user: [% Makefile.user %]
```

Output of Example 6-11:

```
uid: 500

user:
```

If the stat on the file fails (e.g., file doesn't exist, bad permission, etc.), the constructor will throw a File exception. This can be caught within a TRY...CATCH block:

```
[% TRY %]

   [% USE File '/tmp/myfile'  %]

   File exists!

[% CATCH File %]

   File error: [% error.info %]

[% END %]
```

Note the capitalization of the exception type, File, to indicate an error thrown by the File plugin, to distinguish it from a regular file exception thrown by the Template Toolkit. Like all plugins, the File plugin can be referenced by the lowercase name file; exceptions are always thrown of the File type, regardless of the capitalization of the plugin name used.

The nostat option can be specified to prevent the plugin constructor from performing a stat on the file specified. In this case, the file does not have to exist in the filesystem, no attempt will be made to verify that it does, and no error will be thrown if it doesn't. The entries for the items usually returned by stat will be set empty.

```
[% USE file '/some/where/over/the/rainbow.html', nostat = 1 %]

[% file.mtime %]     # nothing
```

All File plugins, regardless of the *nostat* option, have set a number of items relating to the original path specified:

path

The full, original file path specified to the constructor.

```
[% USE file '/foo/bar.html' %]

[% file.path %]     # /foo/bar.html
```

name

The name of the file without any leading directories.

```
[% USE file '/foo/bar.html' %]

[% file.name %]     # bar.html
```

dir

The directory element of the path with the filename removed.

```
[% USE file '/foo/bar.html' %]

[% file.name %]     # /foo
```

ext

The file extension, if any, appearing at the end of the path following a dot operator (.) (not included in the extension).

```
[% USE file '/foo/bar.html' %]

[% file.ext %]      # html
```

home

This contains a string of the form *../..* to represent the upward path from a file to its root directory.

```
[% USE file 'bar.html' %]

[% file.home %]     # nothing

[% USE file 'foo/bar.html' %]

[% file.home %]     # ..

[% USE file 'foo/bar/baz.html' %]

[% file.home %]     # ../..
```

root

The root item can be specified as a constructor argument, indicating a root directory in which the named file resides. This is otherwise set empty.

```
[% USE file 'foo/bar.html', root='/tmp' %]

[% file.root %]     # /tmp
```

abs

This returns the absolute file path by constructing a path from the root and path options.

```
[% USE file 'foo/bar.html', root='/tmp' %]

[% file.path %]     # foo/bar.html

[% file.root %]     # /tmp

[% file.abs %]      # /tmp/foo/bar.html
```

In addition, the following method is provided:

rel(path)

This returns a relative path from the current file to another path specified as an argument. It is constructed by appending the path to the home item.

```
[% USE file 'foo/bar/baz.html' %]

[% file.rel('wiz/waz.html') %]      # ../../wiz/waz.html
```

6.2.9 Format

The Format plugin provides a simple way to format text according to a specific format. The *format* is a text string, and can contain regular text interspersed with sprintf-style placeholders (the format string is passed to Perl's sprintf). Each *%x* token will be replaced with successive elements of the list provided to the function call. This plugin is very similar to the *format* filter, described in Chapter 5.

USE format creates a functionlike variable that can be used for formatting. Example 6-12 shows a simple way to wrap text in HTML comments.

Example 6-12. HTML comments

```
[% USE commented = format('<!-- %s -->') -%]

[% commented('The cat sat on the mat') %]
```

Output of Example 6-12:

```
<!-- The cat sat on the mat -->
```

Mutiple elements can be included as well, by passing multiple items. Format tokens of %s will be treated as strings, but tokens of %d will be treated as numbers, as shown in Example 6-13.

Example 6-13. image tag

```
[% USE img = format('<img src="%s" width="%d" height="%d" />') -%]

[% img('logo.png', '0088', 42) %]
```

Output of Example 6-13:

```
<img src="logo.png" width="88" height="42" />
```

All of the formatting rules and tricks that apply to the *format* filter also apply to the Format plugin. See Chapter 5 for some more examples.

As with the *format* filter, width, precision, and minimum and maximum lengths can be provided as part of the filter, as Example 6-14 shows.

Example 6-14. Using precision and width with format

```
[% USE fmt = format("%2.8f");

   USE Math;

   fmt(Math.pi)

%]
```

Output of Example 6-14:

```
3.14159265
```

6.2.10 GD

Lincoln Stein's GD modules provide access to the gd graphics library. gd is a small, fast graphics library that allows you to create color drawings using a large number of graphics primitives, and emits the drawings in a number of popular graphics formats, such as PNG or JPEG.

In the following example, a new image is created with the USE call. The plugin's contructor takes the same arguments as GD::Image itself:

```
[% USE img1 = GD.Image              # empty image of default size (64x64) %]

[% USE img2 = GD.Image(X, Y)        # empty image (X x Y) %]

[% USE img3 = GD.Image(filename)    # a preexisting image %]
```

To use an existing image, use the filename form of the constructor. The GD plugin will attempt to determine the type of image based on the first few bytes of the file, and then Do The Right Thing.

Once you have an image object, you can call methods on it. Colors are allocated using the colorAllocate method, which accepts a (red, green, blue) triplet as integers:

```
[% orange = img.colorAllocate(255, 165, 0) %]

[% red = img.colorAllocate(255, 0, 0) %]

[% blue = img.colorAllocate(0, 0, 255) %]
```

The first color allocated becomes the background color,[3] so choose wisely!

[3] There are plenty of example colors in your system's *rgb.txt.*

The getPixel method is used in conjunction with the rgb method to return the color of a particular pixel.[4]

[4] GD stores images in a bitmapped form internally; getPixel returns the index into the color table of the color at the specified pixel, and the rgb method turns that back into a triplet.

To get the color at pixel (42,24), you could use this:

```
[% index = img.getPixel(42, 42);
   rgb = img.rgb(index)
%]
```

Or, more succinctly:

```
[% rgb = img.getPixel(42, 42).rgb(index) %]
```

GD supports several output types, including PNG, JPEG, WBMP, and its own GD and GD2 formats. You are likely to use only PNG and JPEG on a regular basis, though the GD2 format is useful for storing images that will be manipulated primarily by GD.

Here are the GD.Image output methods:

```
[% img.png      # emit the image as a PNG... %]
[% img.jpeg     # ... or as a JPEG... %]
[% img.gd       # ... or in GD %]
[% img.gd2      # ... or GD2 formats %]
```

When combined with the OUTPUT_PATH and redirect filter, the GD plugins can be used to automate image creation.

Because these plugins are used to create binary output, it is very important that no extraneous template output appear before or after the image. Because some methods return values that would otherwise appear in the output, it is recommended that this plugin code be wrapped in a null filter. The methods that produce the final output (e.g., *png*, *jpeg*, *gd*, etc.) can then explicitly make their output appear by using the *stdout* filter, with a non-zero argument to force binary mode (see Example 6-15).

Example 6-15. Strange, pointless shapes made entirely with GD

```
[%  FILTER null;
    USE im   = GD.Image(100, 100);
    USE c    = GD.Constants;
    USE poly = GD.Polygon;

    # allocate some colors; white is the background
    white  = im.colorAllocate(255, 255, 255);
    black  = im.colorAllocate(0, 0, 0);
    orange = im.colorAllocate(255, 165, 0);
    blue   = im.colorAllocate(0, 0, 255);

    # Put a black-bordered orange square in the middle
    im.filledRectangle(10, 10, 90, 90, orange);
    im.rectangle(10, 10, 90, 90, black);

    # Draw a diamond in the middle
    poly.addPt(0, 50);
```

```
    poly.addPt(50, 100);

    poly.addPt(100, 50);

    poly.addPt(50, 0);

    im.filledPolygon(poly, blue);


    # Put a smaller black-bordered white square in the middle of that

    im.filledRectangle(30, 30, 70, 70, white);

    im.rectangle(30, 30, 70, 70, black);


    # Output binary image in PNG format

    im.png | stdout(1);

    END;

-%]
```

The GD.Constants plugin provides templates with access to the many GD constants that define font types, styles, and other image attributes.

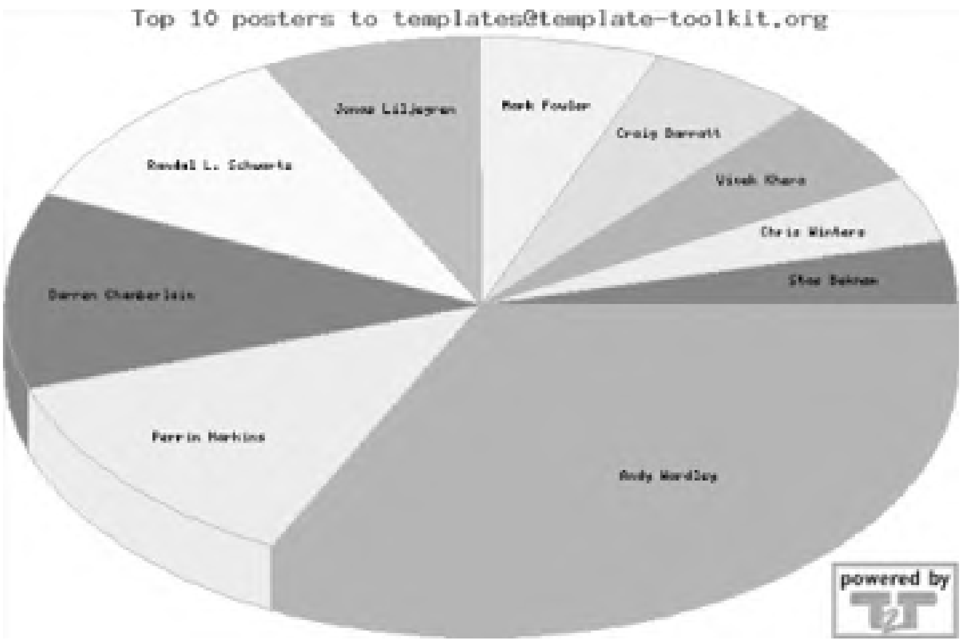
The GD.Graph plugins provide an interface to Martien Verbruggen's GD::Graph module. This module is built on top of GD and can generate graphs, plots, and charts.

The GD.Graph plugins are actually a group of several smaller plugins: GD.Graph.area, GD.Graph.bars, GD.Graph.bard3d, GD.Graph.lines, GD.Graph.lines3d, GD.Graph.linespoints, GD.Graph.mixed, GD.Graph.pie, GD.Graph.pie3d, and GD.Graph.points. All of the plugins have the same interface and differ mainly in the accepted arguments; see the GD::Graph documentation for a full API guide, including the differences between the types.

Figure 6-1 shows a pie chart generated from a datafile containing the top 10 posters to the Template Toolkit mailing list, generated from the single large *mbox* file that the mailman maintains.[5]

[5] At http://www.template-toolkit.org/pipermail/templates.mbox/templates.mbox.

Figure 6-1. Top 10 posters



This graph was generated using the simple template in Example 6-16.

Example 6-16. Generating a graph of the top 10 posters

```
[%
FILTER null;
   USE graph = GD.Graph.pie(600, 400);
   USE gdc = GD.Constants;
   USE posters = datafile('posters');

   data = [
      [ ]     # posters
      [ ]     # the count
   ];
   list = 'templates@template-toolkit.org';

   FOREACH poster IN posters;
      data.0.push(poster.name);
      data.1.push(poster.posts);
   END;

   graph.set(title       = "Top 10 posters to $list"
           transparent = 0
           logo        = 'tt2power.png'
           t_margin    = 4
           b_margin    = 4
           r_margin    = 4
           l_margin    = 4
           start_angle = -90 # aesthetics hack
   );

   # A big font for the title
   graph.set_title_font(gdc.gdGiantFont);

   graph.plot(data).png | stdout(1);
  END;
-%]
```

The GD.Text, GD.Text.Align, and GD.Text.Wrap plugins provide interfaces to the GD::Text module. GD::Text provides a font-independant way of dealing with text in GD and the GD plugins. This is useful primarily for aligning text on GD.Image objects; because positioning strings needs to be done based on pixel offsets, GD.Text's get('width') and get('height') functionality can be invaluable.

Here's an example of using GD.Text:

```
[%
   FILTER null;
   msg = 'Template Toolkit';
   USE gdc = GD.Constants;
   USE t = GD.Text(text = msg
                  font = gdc.gdGiantFont);

   width = t.get('width');      # width of the string in pixels
   height = t.get('height');    # height of the string in pixels

   imgwidth = width * 3;
   imgheight = height * 3;

   USE img = GD.Image(imgwidth, imgheight);
   black = img.colorAllocate(0, 0, 0);
   orange = img.colorAllocate(255, 165, 0);

   img.string(gdc.gdGiantFont, width, height, msg, orange);

   img.png | stdout(1);
END;
-%]
```

The GD module is available on CPAN at http://search.cpan.org/dist/GD/, and the underlying gd C library lives at http://www.boutell.com/gd/. The GD::Graph module is available on CPAN at http://search.cpan.org/dist/GD-Graph/, and the GD::Text module is available on CPAN at http://search.cpan.org/dist/GD-Text/.

6.2.11 HTML

The HTML plugin provides a simple mechanism for generating arbitrary HTML elements. It also provides utility methods for creating attribute lists and for HTML- and URL-escaping.

Generating a single element is done with the element method, as shown in Example 6-17.

Example 6-17. Generating a single element

```
[% USE HTML %]
[% HTML.element('html') %]
```

Output of Example 6-17:

```
<html>
```

Not very exciting. Any named parameters provided become attribute pairs, as shown in Example 6-18.

Example 6-18. Generating an element with attributes

```
[% USE HTML %]

[% HTML.element('img',

            src = 'logo.png'

            width = 88

            height = 38

            alt = 'Company Logo'

            name = 'logo')

%]
```

Output of Example 6-18:

```
<img alt="Company Logo" height="38" src="logo.png" width="88" name="logo">
```

If the plugin is used with the `sorted` option set, then the attributes are sorted in alphabetical order when the attribute list is produced, as shown in Example 6-19.

Example 6-19. Generating an element with sorted attributes

```
[% USE HTML(sorted=1) %]

[% HTML.element('img',

            src = 'logo.png'

            width = 88

            height = 38

            alt = 'Company Logo'

            name = 'logo')

%]
```

Output of Example 6-19:

```
<img alt="Company Logo" height="38" name="logo" src="logo.png" width="88">
```

The plugin also provides HTML- and URL-escaping utility methods, which can be used independently of the plugin's element generating methods. The `escape` method does HTML-escaping, as shown in Example 6-20.

Example 6-20. Using escape

```
[% USE HTML %]

[% HTML.escape("I'd prefer that you type that tag as <br />") %]
```

Output of Example 6-20:

```
I'd prefer that you type that tag as &lt;br /&gt;
```

And the `url` method does URL-escaping, as shown in Example 6-21.

Example 6-21. Using url

```
[% USE HTML %]

[% HTML.url("I'd prefer that you type that tag as <br />") %]
```

Output of Example 6-21:

```
I%27d%20prefer%20that%20you%20type%20that%20tag%20as%20%3Cbr%20%2F%3E
```

As Example 6-22 shows, the `attribute` method can be used to generate an attribute string from a hash (this method is used internally by `element`).

Example 6-22. Generating a string of attributes from a hash

```
[% USE HTML(sorted=1);
  attr = {
    type = 'submit'
    name = 'search'
    value = 'Go!'
  };
  HTML.attributes(attr);
%]
```

Output of Example 6-22:

```
name="search" type="submit" value="Go!"
```

6.2.12 Image

The Image plugin provides a wrapper for image files. This plugin makes available the wrapped image's size, type, and modification time as methods, and also provides methods for generating an HTML tag for the image:

```
[% USE image 'tt2power.png' %]
```

The Image plugin will use either the Image::Info or Image::Size modules (both are available from CPAN), or will throw a runtime error if neither is present on the system. Image::Info is used in preference to Image::Size because it provides more information about the image.

Regardless of which underlying module is used, the name, height, width, and modification time of the image will be available (see Examples Example 6-23 and Example 6-24).

Example 6-23. Basic image info

```
[% image.name %] was last modified on [% date.format(image.modtime) %].
```

Output of Example 6-23:

```
tt2power.png was last modified on 09:29:02 11-Aug-2003.
```

Example 6-24. Image height and width

```
Height: [% image.height %]
Width: [% image.width %]
```

Output of Example 6-24:

```
Height: 47
Width: 78
```

In addition, if the plugin uses Image::Info, several more methods are available, including the following:

file_media_type

Returns the media type in major/minor form and produces the following output:

```
Content-type: image/png
```

file_ext

Returns the extension of the image file and produces the following output:

png

resolution

The value of this field normally gives the physical size of the image on screen or paper. When the unit specifier is missing, this field denotes the squareness of pixels in the image.

The syntax of this field is:

```
<res> <unit>

<xres> "/" <yres> <unit>

<xres> "/" <yres>
```

The `<res>`, `<xres>`, and `<yres>` fields are numbers. The `<unit>` is a string such as dpi, dpm, or dpcm (denoting "dots per inch/meter/cm).

The previous example produces the following output:

```
Resolution: 1/1
```

In addition, several other attributes are available when using `Image::Info` that depend on the image type; for example, animated gifs have a `GIF_loop` attribute.

The Image plugin has two utility methods: `attr`, which returns the image's height and width as XHTML attributes; and `tag`, which returns a formatted XHTML string representing the image. For instance, this code:

```
[% image.attr %]
```

would produce this output:

```
width="78" height="47"
```

The `tag` method creates a full XHTML tag, with attributes (using the `attr` method). For instance, this code:

```
[% image.tag %]
```

produces this output:

```
<img src="tt2power.png" width="78" height="47" />
```

The `tag` method can also take arbitrary named parameters, and will Do The Right Thing with them:

```
[% image.tag(alt = 'Powered by TT', name = 'tt2power') %]
```

The previous code would output the following:

```
<img src="tt2power.png" width="78" height="47" alt="Powered by TT" name="tt2power" />
```

6.2.13 Iterator

The Iterator plugin provides a way to create a `Template::Iterator` object to iterate over a data set. An *iterator* is used for walking through the elements of a list; one is created automatically by the *FOREACH* directive and is aliased to the `loop` variable.

This plugin allows an iterator to be explicitly created with a given name, or with the default plugin name, `iterator`. Example 6-25 shows how to create your own iterator.

Example 6-25. Creating your own iterator

```
[% USE iterator(list) %]


[% FOREACH item IN iterator %]

   [% '<list>' IF iterator.first %]

   <item>[% item %]</item>

   [% '</list>' IF iterator.last %]

[% END %]
```

The Iterator plugin is useful when you want to use a portion of a list in a FOREACH loop, rather than the entire list, as shown in Example 6-26.

Example 6-26. Iterating over part of a list

```
[% days = [ 'Sunday'   'Monday' 'Tuesday' 'Wednesday'

          'Thursday' 'Friday' 'Saturday'

         ] %]

[% USE weekdays = iterator(days.slice(1,5)) %]

[% FOREACH weekday IN weekdays %]

   [% weekday %]

[% END %]
```

Because an iterator contains references to other objects and not copies of the objects themselves, this can be more efficient than simply creating a new list containing only the desired elements. This is especially when the list is large, true when it contains items other than simple data elements (such as objects), or when generating the data is expensive (as when generating database queries). So, in Example 6-26, weekdays persists beyond the FOREACH loop shown and can be reused.

Unlike the transient iterators created within FOREACH loops, specifically created iterators don't go out of scope at the end of their enclosing loop. This means that iterators can be reused. Example 6-27 illustrates this.

Example 6-27. Reusing iterators

```
[%  USE iterator([ 1 .. 3 ]);

   USE fmt = format("%02d => %02d/%02d\n");

   BLOCK iterate;

      fmt(i, it.count, it.size)

         FOREACH i IN it;

      "\n";

   END;

-%]


[% PROCESS iterate it = iterator FOREACH [ 1 .. 3 ] %]
```

Output of Example 6-27:

```
01 => 01/03

02 => 02/03

03 => 03/03


01 => 01/03

02 => 02/03

03 => 03/03


01 => 01/03

02 => 02/03

03 => 03/03
```

6.2.14 Pod

This plugin provides an interface to the Pod::POM module, which parses POD[6] documents into an internal object model that can then be traversed and presented through the Template Toolkit.

[6] POD, which stands for *Plain Old Documentation*, is Perl's internal documentation format. It is intentionally simple and extensible, and is designed to be readable without special processing.

You create a POD parser with USE:

```
[% USE pod %]
```

This parser can then be used to parse documents in POD format:

```
[% pom = pod.parse_file('Chapter6.pod') %]
```

Pod::POM presents POD documents as a tree, of which each branch represents successive =head1 tags in the document. =head2 elements form branches within these sections, and so on, down to the content nodes at the end. The Pod::POM documentation describes this *Pod Object Model* (that's what POM stands for) in great detail.

For more details on using the POD plugin, and on Pod::POM in general, please consult the Pod::POM documentation.

Pod::POM is available from CPAN at http://search.cpan.org/dist/Pod-POM/.

6.2.15 String

This is a plugin module for object-oriented string manipulation. A String object is created via the USE directive, adding any initial text value as an argument or as the named parameter text:

```
[% USE String %]

[% USE String 'initial text' %]

[% USE String text='initial text' %]
```

It's likely that there will be more than one string in a template, so assigning the plugin to a name is wise:

```
[% USE greeting = String 'Hello World' %]
```

Once you've got a String object, you can use it as a prototype to create other String objects with the new method:

```
[% USE String %]

[% greeting = String.new('Hello World') %]
```

The new method also accepts an initial text string as an argument or the named parameter text:

```
[% greeting = String.new( text => 'Hello World' ) %]
```

You can also call the copy method to create a new string as a copy of the original:

```
[% greet2 = greeting.copy %]
```

The String object has a text method to return the content of the string:

```
[% greeting.text %]
```

However, it is sufficient to simply print the string and let the overloaded stringification operator call the text method automatically for you:

```
[% greeting %]
```

Thus, you can treat String objects pretty much like any regular piece of text, interpolating it into other strings, for example:

```
[% msg = "It printed '$greeting' and then dumped core\n" %]
```

You also have the benefit of numerous other methods for manipulating the string:

```
[% msg.append("PS  Don't eat the yellow snow") %]
```

Note that all methods operate on and mutate the contents of the string itself. If you want to operate on a copy of the string, simply take a copy first:

```
[% msg.copy.append("PS  Don't eat the yellow snow") %]
```

These methods return a reference to the String object itself. This allows you to chain multiple methods together:

```
[% msg.copy.append('foo').right(72) %]
```

It also means that in the previous examples, the string is returned. This causes the text method to be called, which results in the new value of the string being printed. To suppress printing of the string, you can use the CALL directive:

```
[% foo = String.new('foo') %]


[% foo.append('bar') %]         # prints "foobar"


[% CALL foo.append('bar') %]    # nothing
```

There are several ways to create a new String object. Here is the "usual" way:

```
[% USE err = String text = 'Bad Things Happened' %]
```

Alternatively, calling the new method on an already initialized String object will create a new string:

```
[% msg = err.new('False alarm!') %]
```

Finally, copy will return a copy of the string object:

```
[% urgent_error = err.copy.append(' - lp1 on fire') %]
```

The plugin also implements many methods to inspect or modify the contents of the String object. Here is a list of the methods:

text

Returns the internal text value of the string. The stringification operator is overloaded to call this method. Thus, the following are equivalent:

```
[% msg.text %]

[% msg %]
```

length

Returns the length of the string.

```
[% USE String("foo") %]

[% String.length %]   # => 3
```

search($pattern)

Searches the string for the regular expression specified in $pattern, returning true if found, or returning false otherwise.

```
[% item = String.new('foo bar baz wiz waz woz') %]

[% item.search('wiz') ? 'WIZZY! :-)' : 'not wizzy :-(' %]
```

split($pattern, $limit)

Splits the string based on the delimiter $pattern and optional $limit. Delegates to Perl's internal split, so the parameters are exactly the same.

```
[% FOREACH item.split %]

    ...

[% END %]

[% FOREACH item.split('baz|waz') %]

    ...

[% END %]
```

The following methods modify the internal value of the string. For example:

```
[% USE str=String('foobar') %]


[% str.append('.html') %]      # str => 'foobar.html'
```

The value of the string str is now foobar.html. If you don't want to modify the string, simply take a copy first.

```
[% str.copy.append('.html') %]
```

These methods all return a reference to the String object itself. This has two important benefits. The first is that when used as shown earlier, the String object str returned by the append method will be stringified with a call to its text method. This will return the newly modified string content. In other words, a directive such as:

```
[% str.append('.html') %]
```

will update the string and also print the new value. If you just want to update the string but not print the new value, use CALL:

```
[% CALL str.append('.html') %]
```

The other benefit of these methods returning a reference to the string is that you can chain as many different method calls together as you like. For example:

```
[% String.append('.html').trim.format(href) %]
```

Here are the methods:

push($suffix, ...) / append($suffix, ...)

Appends all arguments to the end of the string. The append method is provided as an alias for push.

```
[% msg.push('foo', 'bar') %]

[% msg.append('foo', 'bar') %]
```

pop($suffix)

Removes the suffix passed as an argument from the end of the string.

```
[% USE String 'foo bar' %]

[% String.pop(' bar')  %]   # => 'foo'
```

unshift($prefix, ...) / prepend($prefix, ...)

Prepends all arguments to the beginning of the string. The prepend method is provided as an alias for unshift.

```
[% msg.unshift('foo ', 'bar ') %]

[% msg.prepend('foo ', 'bar ') %]
```

shift($prefix)

Removes the prefix passed as an argument from the start of the string.

```
[% USE String 'foo bar' %]

[% String.shift('foo ') %]   # => 'bar'
```

left($pad)

If the length of the string is less than $pad, the string is left-formatted and padded with spaces to $pad length.

```
[% msg.left(20) %]
```

right($pad)

As per left( ), but right-padding the string to a length of $pad.

```
[% msg.right(20) %]
```

center($pad) / centre($pad)

As per *left( )* and *right( )*, but formatting the string to be centered within a space-padded string of length $pad. The centre method is provided as an alias for center to account for misspellings.

```
[% msg.center(20) %]    # American spelling

[% msg.centre(20) %]    # European spelling
```

format($format)

Apply a format in the style of sprintf to the string.

```
[% USE String("world") %]

[% String.format("Hello %s\n") %]  # => "Hello World\n"
```

upper( )

Converts the string to uppercase.

```
[% USE String("foo") %]

[% String.upper %]  # => 'FOO'
```

lower( )

Converts the string to lowercase

```
[% USE String("FOO") %]

[% String.lower %]  # => 'foo'
```

capital( )

Converts the first character of the string to uppercase.

```
[% USE String("foo") %]

[% String.capital %]  # => 'Foo'
```

The remainder of the string is left untouched. To force the string to be all lowercase with only the first letter capitalized, you can do something like this:

```
[% USE String("FOO") %]

[% String.lower.capital %]  # => 'Foo'
```

chop( )

Removes the last character from the string:

```
[% USE String("foop") %]

[% String.chop %]       # => 'foo'
```

chomp( )

Removes the trailing newline from the string:

```
[% USE String("foo\n") %]

[% String.chomp %]       # => 'foo'
```

trim( )

Removes all leading and trailing whitespace from the string:

```
[% USE String("   foo   \n\n ") %]

[% String.trim %]       # => 'foo'
```

collapse( )

Removes all leading and trailing whitespace, and collapses any sequences of multiple whitespace to a single space:

```
[% USE String(" \n\r  \t  foo   \n \n bar  \n") %]

[% String.collapse %]   # => "foo bar"
```

truncate($length, $suffix)

Truncates the string to $length characters.

```
[% USE String('long string') %]

[% String.truncate(4) %]  # => 'long'
```

If $suffix is specified, it will be appended to the truncated string. In this case, the string will be further shortened by the length of the suffix to ensure that the newly constructed string, complete with suffix, is exactly $length characters long.

```
[% USE msg = String('Hello World') %]

[% msg.truncate(8, '...') %]   # => 'Hello...'
```

replace($search, $replace)

Replaces all occurrences of $search in the string with $replace.

```
[% USE String('foo bar foo baz') %]

[% String.replace('foo', 'wiz')  %]  # => 'wiz bar wiz baz'
```

remove($search)

Removes all occurrences of $search in the string.

```
[% USE String('foo bar foo baz') %]

[% String.remove('foo ')  %]  # => 'bar baz'
```

repeat($count)

Repeats the string $count times.

```
[% USE String('foo ') %]

[% String.repeat(3)  %]  # => 'foo foo foo '
```

6.2.16 Table

The Table plugin allows you to format a list of data items into a virtual table. When you create a Table plugin via the *USE* directive, simply pass a list reference as the first parameter and then specify a fixed number of rows or columns:

```
[% USE table list, rows = 5  %]
```

The plugin then presents a table-based view of the data set. The data isn't actually reorganized in any way, but is available via row, col, rows, and cols as if formatted into a simple two-dimensional table of n rows x n columns. Thus, if our sample alphabet list contained the letters a to z, the preceeding *USE* directives would create plugins that represent the views of the alphabet, as shown in Examples Example 6-28 and Example 6-29.

Example 6-28. rows

```
[% USE table alphabet, rows = 5 %]


[% FOREACH row IN table.row;
   FOREACH cell IN row;
      "$cell  ";
   END %]
[% END %]
```

Output of Example 6-28:

```
a f k p u z
b g l q v
c h m r w
d i n s x
e j o t y
```

Example 6-29. cols

```
[% USE table alphabet, cols = 5 %]


[% FOREACH col IN table.col;
   FOREACH cell IN col;
      "$cell  ";
   END %]
[% END %]
```

Output of Example 6-29:

```
a b c d e f
g h i j k l
m n o p q r
s t u v w x
y z
```

We can request a particular row or column using the row and col methods, as shown in Example 6-30.

Example 6-30. row(0)

```
[% USE table alphabet, rows = 5 %]
[% FOREACH item IN table.row(0);
   item %]
[% END %]
```

Output of Example 6-30:

a

f

k

p

u

z

Data in rows is returned from left to right, and in columns from top to bottom. The first row/column is 0. By default, rows or columns that contain empty values will be padded with the undefined value to fill it to the same size as all other rows or columns. For example, the last row (row 4) in the first example would contain the values [e j o t y undef]. The Template Toolkit will safely accept these undefined values and print an empty string. You can also use the *IF* directive to test whether the value is set.

You can explicitly disable the pad option when creating the plugin to returned shortened rows/columns where the data is empty, as shown in Example 6-31.

Example 6-31. pad = 0

```
[% USE table alphabet, cols=5, pad=0 %]

[% FOREACH item = table.col(4);

  item %]

[% END %]
```

The rows method returns all rows/columns in the table as a reference to a list of rows (themselves list references). The row method, when called without any arguments calls rows to return all rows in the table. cols and col behave analogously.

6.2.17 URL

The URL plugin provides a convenient way to construct URLs from a base stem and a hash of additional parameters, without having to worry about getting the syntax correct.

The constructor should be passed a base URL:

```
[% USE siteroot = url('http://www.template-toolkit.org') %]
```

The constructor can optionally be passed a hash reference of default parameters and values:

```
[% USE next = url('search.cgi', search = search, next = curpage + 1) %]
```

When the plugin is then called without any arguments, the default base and parameters are returned as a formatted URL, including any query parameters. Thus, one url object can be used as the base for another:

```
[% USE news = url("$siteroot/news") %]
```

Simply calling or interpolating the plugin is enough for the Template Toolkit to expand it, as shown in Example 6-32.

Example 6-32. url in action

```
[% USE tt = url('http://www.template-toolkit.org/') -%]

The <a href="[% tt %]">Template Toolkit</a> rules!
```

Output of Example 6-32:

```
The <a href="http://www.template-toolkit.org/">Template Toolkit</a> rules!
```

Any parameters passed into the call are combined with parameters specified when the plugin was created, and all become part of the resulting URL, as shown in Example 6-33.

Example 6-33. url + parameters

```
[% USE article = url('http://slashdot.org/article.pl'
                  mode = 'nested',
                  threshold = 1) %]
[% article(sid = 'xxx') %]
```

Output of Example 6-33:

```
http://slashdot.org/article.pl?mode=nested&amp;sid=xxx&amp;threshold=1
```

6.2.18 Wrap

The Wrap plugin provides a simple text wrapper, based on the Text::Wrap module. Paragraphs can be formatted using specific widths and leading indent, and can have padding applied to each line in the output.

The plugin defines a wrap subroutine that is called with the input text and further optional parameters to specify the page width (which defaults to 72) and tab characters for the first and subsequent lines (these have no defaults).

This plugin's simple wrapping is not aware of special prefixes and so forth; for more sophisticated wrapping, use the more complex autoformat plugin. For most simple wrapping jobs, however, wrap is capable enough (see Example 6-34).

Example 6-34. Basic wrapping

```
[% USE wrap %]

[% text = BLOCK -%]
First, attach the transmutex multiplier to the cross-wired quantum homogenizer.
[% END %]

[% wrap(text, 30) %]
```

Output of Example 6-34:

```
First, attach the transmutex
multiplier to the cross-wired
quantum homogenizer.
```

The plugin also registers a wrap filter that accepts the same three optional arguments, but takes the input text directly via the filter input (see Example 6-35).

Example 6-35. Wrap filter

```
[% FILTER bullet = wrap(40, '* ', '  ') -%]
First, attach the transmutex multiplier to the cross-wired quantum homogenizer.
[%- END %]

[% FILTER bullet -%]
Then remodulate the shield to match the harmonic frequency, taking care to correct the
phase difference.
[% END %]
```

Output of Example 6-35:

```
* First, attach the transmutex
  multiplier to the cross-wired quantum
  homogenizer.

* Then remodulate the shield to match
  the harmonic frequency, taking care
  to correct the phase difference.
```

Text::Wrap comes with recent versions of Perl, and is also available from CPAN at http://search.cpan.org/dist/Text-Wrap/.

6.2.19 XML::DOM

The XML::DOM plugin gives access to the XML Document Object Module via Clark Cooper and Enno Derksen's XML::DOM module. The following synopsis gives examples of some ways in which it can be used. See Chapter 10 for further details.

```
# load plugin
[% USE dom = XML.DOM %]

# also provide XML::Parser options
[% USE dom = XML.DOM(ProtocolEncoding => 'ISO-8859-1') %]

# parse an XML file
[% doc = dom.parse(filename) %]
[% doc = dom.parse(file => filename) %]

# parse XML text
[% doc = dom.parse(xmltext) %]
[% doc = dom.parse(text => xmltext) %]

# call any XML::DOM methods on document/element nodes
[% FOREACH node = doc.getElementsByTagName('report') %]
   * [% node.getAttribute('title') %]     # or just '[% node.title %]'
[% END %]

# define VIEW to present node(s)
[% VIEW report notfound='xmlstring' %]
   # handler block for a <report>...</report> element
   [% BLOCK report %]
      [% item.content(view) %]
   [% END %]
```

```
  # handler block for a <section title="...">...</section> element
  [% BLOCK section %]
  <h1>[% item.title %]</h1>
  [% item.content(view) %]
  [% END %]

  # default template block converts item to string representation
  [% BLOCK xmlstring; item.toString; END %]

  # block to generate simple text
  [% BLOCK text; item; END %]
[% END %]

# now present node (and children) via view
[% report.print(node) %]

# or print node content via view
[% node.content(report) %]
```

6.2.20 XML::RSS

The XML::RSS plugin is a simple interface to Jonathan Eisenzopf's XML::RSS module. A Rich Site Summary (RSS) file is typically used to store short news headlines describing different links within a site. This plugin allows you to parse RSS files and format the contents accordingly using templates.

```
[% USE news = XML.RSS(filename) %]

[% FOREACH item = news.items %]
  [% item.title %]
  [% item.link  %]
[% END %]
```

See Chapter 10 for more details.

6.2.21 XML::Style

This plugin defines a filter for performing simple stylesheet-based transformations of XML text.

Named parameters are used to define those XML elements that require transformation. These may be specified with the USE directive when the plugin is loaded and/or with the *FILTER* directive when the plugin is used.

This example shows how the default attributes border="0" and cellpadding="4" can be added to <table> elements:

```
[% USE xmlstyle
      table = {
         attributes = {
            border     = 0
```

```
            cellpadding = 4
         }
      }
%]

[% FILTER xmlstyle %]
<table>
  ...
</table>
[% END %]
```

This produces the output:

```
<table border="0" cellpadding="4">
  ...
</table>
```

Parameters specified within the *USE* directive are applied automatically each time the xmlstyle filter is used. Additional parameters passed to the *FILTER* directive apply only to that block.

```
[% USE xmlstyle
      table = {
         attributes = {
            border      = 0
            cellpadding = 4
         }
      }
%]

[% FILTER xmlstyle
      tr = {
         attributes = {
            valign="top"
         }
      }
%]
<table>
  <tr>
    ...
  </tr>
</table>
[% END %]
```

Of course, you may prefer to define your stylesheet structures once and simply reference them by name. Passing a hash reference of named parameters is just the same as specifying the named parameters as far as the Template Toolkit is concerned:

```
[% style_one = {
     table = { ... }
     tr    = { ... }
   }
   style_two = {
     table = { ... }
     td    = { ... }
   }
   style_three = {
     th = { ... }
     tv = { ... }
   }
%]

[% USE xmlstyle style_one %]

[% FILTER xmlstyle style_two %]
   # style_one and style_two applied here
[% END %]

[% FILTER xmlstyle style_three %]
   # style_one and style_three applied here
[% END %]
```

Any attributes defined within the source tags will override those specified in the stylesheet:

```
[% USE xmlstyle
      div = { attributes = { align = 'left' } }
%]

[% FILTER xmlstyle %]
<div>foo</div>
<div align="right">bar</div>
[% END %]
```

The output produced is:

```
<div align="left">foo</div>
<div align="right">bar</div>
```

The filter can also be used to change the element from one type to another:

```
[% FILTER xmlstyle
        th = {
            element = 'td'
```

```
        attributes = { bgcolor='red' }
    }
%]
<tr>
  <th>Heading</th>
</tr>
<tr>
  <td>Value</td>
</tr>
[% END %]
```

The output here is as follows (notice how the end tag </th> is changed to </td> as is the start tag):

```
<tr>
  <td bgcolor="red">Heading</td>
</tr>
<tr>
  <td>Value</td>
</tr>
```

You can also define text to be added immediately before or after the start or end tags. For example:

```
[% FILTER xmlstyle
    table = {
        pre_start = '<div align="center">'
        post_end  = '</div>'
    }
    th = {
        element    = 'td'
        attributes = { bgcolor='red' }
        post_start = '<b>'
        pre_end    = '</b>'
    }
%]
<table>
<tr>
  <th>Heading</th>
</tr>
<tr>
  <td>Value</td>
</tr>
</table>
[% END %]
```

The output produced is:

```
<div align="center">

<table>

<tr>

  <td bgcolor="red"><b>Heading</b></td>

</tr>

<tr>

  <td>Value</td>

</tr>

</table>

</div>
```

6.2.22 XML::XPath

The XML::XPath plugin provides an interface to Matt Sergeant's XML::XPath module. The following synopsis shows some examples of its use. See Chapter 10 and Chapter 11 for further examples of using this plugin.

```
[% USE xpath = XML.XPath(xmlfile) %]

[% USE xpath = XML.XPath(file => xmlfile) %]

[% USE xpath = XML.XPath(filename => xmlfile) %]


# load plugin and specify XML text to parse

[% USE xpath = XML.XPath(xmltext) %]

[% USE xpath = XML.XPath(xml => xmltext) %]

[% USE xpath = XML.XPath(text => xmltext) %]


# then call any XPath methods (see XML::XPath docs)

[% FOREACH page = xpath.findnodes('/html/body/page') %]

   [% page.getAttribute('title') %]

[% END %]


# define VIEW to present node(s)

[% VIEW repview notfound='xmlstring' %]

   # handler block for a <report>...</report> element

   [% BLOCK report %]

      [% item.content(view) %]

   [% END %]


   # handler block for a <section title="...">...</section> element

   [% BLOCK section %]

   <h1>[% item.getAttribute('title') | html %]</h1>

   [% item.content(view) %]
```

```
  [% END %]

  # default template block passes tags through and renders
  # out the children recursively
  [% BLOCK xmlstring;
    item.starttag; item.content(view); item.endtag;
  END %]

  # block to generate simple text
  [% BLOCK text; item | html; END %]
[% END %]

# now present node (and children) via view
[% repview.print(page) %]

# or print node content via view
[% page.content(repview) %]
```

Chapter 6. Plugins

A templating system that allow only minimal interaction with the outside world would become boring pretty quickly—most of the interesting stuff is going to be outside our templates, not inside. This chapter covers the Template Toolkit plugin system, designed to make interfacing with the outside world as simple as possible.

In the Template Toolkit, a *plugin* provides extra functionality that is otherwise not possible using only the core language. Many plugins create template-facing interfaces between external resources, such as a database or mail server, while some plugins provide tidy interfaces for complex formatting operations. Plugins allow developers to add functionality without having to modify or override core Template Toolkit components.

To a large extent, plugins are what give the Template Toolkit its power and flexibility: if the basic toolkit lacks the functionality you desire, it is very straightforward to add the functionality by creating plugins. External modules, designed without the Template Toolkit in mind, can be subverted for use within templates with just a little glue code. At the same time, however, a plugin can be used to enforce privacy within a module, and to make methods inaccessible, ensuring that the modules get used only as anticipated.

Unlike filters, which exist primarily to postprocess text, a plugin is unlimited in scope. The most popular use for plugins is to integrate other Perl modules—many, if not most, of the thousands of modules found on CPAN can be wrapped in a plugin and made available to a template designer.

7.1 Template Modules

The Template module is simply a frontend that creates and uses a Template::Service object and then pipes the output wherever you want it to go (standard output by default, or maybe a file, scalar variable, etc.). The Apache::Template module is another frontend, which uses a Template::Service::Apache object under the hood and sends the output back to the relevant Apache object. The now-familiar *tpage* and *ttree* scripts are command line-based frontends; *tpage* simply connects standard input and output by way of the Template Toolkit, while *ttree* does the same for source and destination files (with the intelligence to detect when they haven't changed).

These frontend modules are really there only to handle any specifics of the environment in which they're being used. Apache::Template does web-specific things, such as making form parameters and client request headers available as template variables and allowing configuration via *httpd.conf*. The *ttree* program parses command-line arguments and a configuration file. The regular Template frontend deals with standard output and writing to files. Otherwise, it is Template::Service (or a subclass) that does all the work. The process method calls $service->process and then spends most of its time figuring out where to send the results. Example 7-1 shows the *process* method in action.

Example 7-1. Template::process

```
sub process($name, \%vars, $output, \%options) {
    $content = SERVICE->process($name, $vars);


    if type($output) = = 'code':
        &$output($content);


    elsif type($output) = = 'filehandle':
        print $output $content;


    elsif type($output) = = 'scalar reference':
        $$output = $content;


    elsif type($output) = = 'array reference':
        push @$output, $content;


    elsif $output->can('print'):
        $output->print($content);


    else:
        open OUT, $output;
        if $options->{'binmode'}:
            binmode OUT;
        print OUT $content;


}
```

Apache::Template behaves a little differently, but the basic idea is the same. Because it's an Apache handler, the entry point is called handler, not process (see Example 7-2).

Example 7-2. Apache::Template::handler

```
sub handler($r) {
    $template = SERVICE->template($r);
    $params = SERVICE->params($r);
    $content = SERVICE->process($r);

    SERVICE->headers($r, $template, $content);

    $r->print($content);

    return OK;
}
```

As you can see, the service object (Apache::Template uses a Template::Service::Apache instance, which is a Template::Service subclass) has a few more responsibilities: params and header handle the Apache-specific stuff (reading client headers and form parameters), and template calls upon a special provider to get a compiled template based on the filename requested (more on template later). Let's look at these modules in more detail.

7.1.1 Template::Service

The Template::Service module provides a consistent template-processing environment. In addition to processing the main template (passed by name to process), the service object processes any additional templates (PRE_PROCESS, PROCESS, POST_PROCESS), wrappers (WRAPPER), or error handlers (ERROR) defined by the frontend. For the most part, the job of the service object is really just one of scheduling, dispatching, and handling runtime errors.

Actually, that's a bit of a lie: the service object doesn't process the templates itself, but instead makes process calls against a Template::Context object. In pseudocode, process looks like the code shown in Example 7-3.

Example 7-3. Template::Service::process

```
sub process($template, \%vars) {
    $output = '';

    $compiled_template = CONTEXT->template($template);
    $vars->{'template'} = $compiled_template;

    eval {
        foreach $name in PRE_PROCESS:
            $output += CONTEXT->process($name, $vars);

        @process = PROCESS || $compiled_template;
        foreach $name in @process:
            $output += CONTEXT->process($name, $vars);

        @wrapper = reverse WRAPPER;
```

```
        foreach $name in @wrapper:
            $output += CONTEXT->process($name, $vars);

        foreach $name in POST_PROCESS:
            $output += CONTEXT->process($name, $vars);
    }

    if $EVAL_ERROR:
        $output = CONTEXT->process(ERROR);

    return $output;
}
```

7.1.2 Template::Context

Template::Context is the runtime engine for the Template Toolkit—the module that hangs everything together in the lower levels and that does most of the real work, albeit by crafty delegation to various other friendly helper modules.

Given a template name, the context's process method must first get a handle on the compiled template that represents that name. It does this by calling its template method.

Within template, the context calls fetch on each member of the list of Template::Provider objects (the contents of the LOAD_TEMPLATES array), stopping when one of them returns a Template::Document object. If none of them does, the context throws a Template::Exception object back to process via throw, as shown in Example 7-4.

Example 7-4. Template::Context::template

```
sub template($name) {
    $template = undef;

    foreach $p in LOAD_TEMPLATES:
        $template = $p->fetch($name);
        last if $template;

    $self->throw('file', "$name not found") unless $template;

    return $template;
}
```

The throw method takes an error type, such as file, and a descriptive string (*$name not found*), and creates a Template::Exception object out of them. This exception object is first passed back to the Template::Service object, which tries to handle it with any ERROR handlers the user specified; if that fails (i.e., if the user hasn't defined a handler for this exception type), it is passed into the template, where it is available via the error variable. Template::Context also implements a catch method, which attempts to handle a thrown error. The context's catch method ensures that the error caught is a Template::Exception rather than a simple string, and is primarily used within compiled templates. We'll see catch when we talk about Template::Directive and Template::Document.

Once the context has a compiled template, it updates the *stash* (the data engine where template variables are managed) to set any template variable definitions specified as the second argument by reference to a hash array.

Then, it calls the document's process method, passing a reference to itself (the context) as an argument. In doing this, it provides itself as an object against which template code can make callbacks to access runtime resources and Template Toolkit functionality: not only does the Template::Context object receive calls from the *outside* (those originating in user code calling the process method on a Template object), but it also receives calls from the *inside* (those originating in template directives of the form [% PROCESS template %]).

process looks something like the code shown in Example 7-5.

Example 7-5. Template::Context::process

```
sub process(\@names, \%vars) {
    foreach $name in $names:
        push @templates, $self->template($name);

    STASH->update($vars);

    eval {
        foreach $template in @templates:
            $output += &$template($self);
    }

    if $EVAL_ERROR:
        $self->throw($EVAL_ERROR);

    return $output;
}
```

As you can see, process can take an array of template names, so the following:

```
[% PROCESS copyright + footer %]
```

and:

```
$context->process([ 'copyright', 'footer' ]);
```

are equivalent.

The context is also responsible for loading plugins and filters via the cleverly named plugins and filters methods. The context maintains arrays of plugin and filter providers (stored in LOAD_PLUGINS and LOAD_FILTERS, respectively) that are consulted in order, until one of them returns the requested item. plugin is very similar to template, as you can see in Example 7-6.

Example 7-6. Template::Context::plugin

```
sub plugin($name, \@args) {
    $plugin = undef;

    foreach $p in @LOAD_PLUGINS:
        $plugin = $p->fetch($name, $args);
        last if $plugin;

    $self->throw('plugin', "$name not found") unless $plugin;
```

```
    return $plugin;
}
```

filter is slightly different; as shown in Example 7-7, the context can store filters in a local cache, if $alias is provided.

Example 7-7. Template::Context::filter

```
sub filter($name, \@args, $alias) {
    $filter = undef;

    $filter = $self->filter_cache->$name;
    return $filter if $filter;

    foreach $p in @LOAD_FILTERS:
        $filter = $p->fetch($name, $args);
        last if $filter;

    return undef unless $filter;

    $self->filter_cache->$alias = $filter;

    return $filter;
}
```

7.1.3 Template::Stash

The Template::Stash module defines the data engine that powers the Template Toolkit. The stash goes out of its way to ensure that all the data it contains can be accessed in the same way by making variable access "magical": scalars, arrays, hashes, subroutines, and objects are all accessed the same way, courtesy of the dot operator (.). We'll have a lot more to say about the stash shortly in Section 7.2.

7.1.4 Template::Provider

Template::Provider is responsible for locating templates, compiling them with Template::Parser, and handing Template::Document instances back to the context, all via the fetch method. The provider also handles the details of template caching and hides filesystem differences.

In pseudocode, fetch looks something like the code shown in Example 7-8.

Example 7-8. Template::Provider::fetch

```
sub fetch($name) {
    if $name =~ /^\//:
        if ABSOLUTE:
            $data, $error = $self->_fetch(name);
        else:
```

```
            $data = undef;

            $error = 'ABSOLUTE paths not allowed';


    elsif $name =~ /^\.+\//:

        if RELATIVE:

            $data, $error = $self->_fetch($name);

        else:

            $data = undef;

            $error = 'RELATIVE paths not allowed';


    else:

        $data, $error = $self->_fetch_path($name);


    return $data, $error;

}
```

There are two other helper methods here: _fetch and _fetch_path. The primary difference between the two is that _fetch is expecting a direct path to a file (either absolute or relative), while _fetch_path walks the INCLUDE_PATH to find the template. Each checks to see whether the user requested memory or disk-based caching, and uses these versions in preference to recompiling the template itself. If caching is enabled, the provider checks timestamps to ensure that the version on disk hasn't been modified since it was last compiled, and either hands back the cached version, or recompiles it and hands that back (being sure to cache this new version).

_fetch looks like Example 7-9 in pseudocode.

Example 7-9. Template::Provider::_fetch

```
sub _fetch($name) {

    $compiled_filename = $self->_compiled_filename;


    if CACHE_SIZE:

        $cached = $self->template_cache->$name

        if $cached:

            $self->_refresh($cached);

            $doc = $cached;

        else:

            $filedata = $self->_load($name);

            $doc = $self->_compile($filedata, $compiled_filename);

    else:

        if $compiled_filename:

            $doc = $self->_load_compiled($compiled_template);

            $self->store($name, $doc);

        else:

            $filedata = $self->_load($name);
```

```
        $doc = $self->_compile($filedata, $compiled_filename);
        $self->store($name, $doc);

    return $doc;
}
```

We're leaving out a lot of private methods here: _compiled_filename concatenates COMPILE_DIR, the template name, and COMPILE_EXT to figure out where a compiled template should be written to disk, and _refresh does timestamp comparisons between $name and $compiled_filename, calling _load and _compile as necessary. _load opens the file $name on disk and reads it into a scalar variable, and adds the special elements name and modtime to $filedata; these are $name and $name's timestamp (from (stat($name))[9]).

_compile bears a closer look because it is in _compile that the parser comes into play (see Example 7-10).

Example 7-10. Template::Provider::_compile

```
sub _compile($filedata, $compiled_filename) {
    $parsed = PARSER->parse($filedata->{'text'}, $filedata);

    $parsed->{'name'}    = $filedata->{'name'};
    $parsed->{'modtime'} = $filedata->{'time'};

    if $compiled_filename:
        DOCUMENT->write_perl_file($parsed, $compiled_filename);

    return DOCUMENT->new($parsed);
}
```

As mentioned earlier, Template::Provider objects are stored in an array; template iterates over these providers, giving each one a chance to respond. This means that it is possible to layer special-purpose providers (database-based, HTTP-based, and so on) on top of the default provider, or even instead of it.

Once the provider finds the template it is looking for, it passes the contents of the file to a Template::Parser, which tokenizes the templates, checks them for syntactical correctness, and returns a compiled data structure, which is fed to Template::Document.

7.1.5 Template::Parser

Template::Parser does most of the hard work. It accepts a string representation of a template, which it tokenizes based on the current TAGS settings, and uses a Template::Grammar instance to determine the actions associated with each token.

parse is the parser's primary interface, and looks something like the code in Example 7-11.

Example 7-11. Template::Parser::parse

```
sub parse($text, $info) {
    @tokens = $self->split_text($text);

    $block = $self->_parse(@tokens, $info);

    return {
```

```
        BLOCK     = $block
        DEFBLOCKS = $self->DEFBLOCK
        METADATA  = $self->METADATA
    }
}
```

split_text is the tokenizer. It uses START_TAG and END_TAG to break apart the text, and handles any whitespace-chomping specified by PRE_CHOMP, POST_CHOMP, or TRIM. _parse uses the grammar to determine whether the stream of tokens is syntactically valid, and if so, uses Template::Directive to generate Perl code ($block). DEFBLOCK and METADATA are accumulated in the parser as the document is parsed.

7.1.6 Template::Grammar

The Template::Grammar module contains a big list of parser states and their associated actions, which are generated from a yacc-like grammar using Parse::Yapp. The grammar calls upon the Template::Directive factory class to actually generate the code.

Ninety-nine percent of the grammar is generated from the file *parser/Parser.py* (part of the source distribution), which we'll see in more detail later in Chapter 8. The last 1% is part of the grammar skeleton, *parser/Grammar.pm.skel*, which defines reserved words and special tokens.

7.1.7 Template::Directive

The Template::Directive module defines the nitty-gritty details of the compilation process. The grammar calls a method against a Template::Directive instance (called a *factory*), passing along the tokens the parser found. The factory returns Perl code that implements the directives, which is evaled into live code by Template::Document.

By way of example, let's look at the code generated for an anonymous block, such as the one shown in Example 7-12.

Example 7-12. An example template

```
[% BLOCK %]
A city is like a large, complex rabbit.
[% END %]
```

This relatively simple block generates a bunch of code, as shown in Example 7-13.

Example 7-13. Code implementing an anonymous block

```
# BLOCK
$output .=  do {
    my $output  = '';
    my $error;

    eval { BLOCK: {
        $output .=  "\nA city is like a large, complex rabbit.\n";
    } };
    if ($@) {
        $error = $context->catch($@, \$output);
        die $error unless $error->type eq 'return';
```

```
    }

    $output;
};
```

The nested calls to eval are necessary because the user can do pretty much anything in a block, such as attempt to load nonexistent plugins or process a file with syntax errors, as shown in Example 7-14.

Example 7-14. A malformed template

```
[% BLOCK %]
[% USE %]
[% END %]
```

Template::Directive makes use of compile-time constants, as specified by the CONSTANTS configuration directive. When generating the code for *GET* directives, the factory checks to see whether any constants are defined, and if so, calls upon a Template::Namespace::Constants object to do the interpolation then and there. This means that the compiled templates contain static strings for these variables, and not calls to the stash. We'll see the code generation process in much more detail in the later Section 7.2.

7.1.8 Template::Namespace::Constants

The Template::Namespace::Constants module is a specialized factory class (like a slimmed-down Template::Directive) that handles compile-time constant folding. A Template::Namespace::Constants object has its own stash, which is initialized with the contents of the CONSTANTS configuration directive (if it was specified). These variables are accessed in the templates using a special prefix (which is constants by default, but can be set to something else using the CONSTANT_NAMESPACE configuration option). We'll see when constant folding comes into play in the Section 7.2; also see the Appendix for more details about CONSTANTS and CONSTANT_NAMESPACE.

7.1.9 Template::Document

A Template::Document module is a thin object wrapper around a compiled template subroutine. The object implements a process method that performs a little bit of housekeeping and then calls the template subroutine. The object also defines template metadata (defined in [% META ... %] directives), and has a blocks method that returns a hash of any additional [% BLOCK xxxx %] definitions found in the template source.

The context processes a Template::Document instance by invoking its process method, passing itself as a parameter; within process, the document executes its main subroutine (which it gets via the block method) and returns a string of output. If there is an error, the context intercepts it with the catch method, which ensures that the error is a Template::Exception object and not a string, and then rethrows it via dia (which is caught by the context in its own process method). Example 7-15 shows this module in action.

Example 7-15. Template::Document::process

```
sub process($context) {
    $output = '';

    eval {
        $block = $self->block;
        $output = &$block($context);
    }

    if $EVAL_ERROR:
```

```
        die $context->catch($EVAL_ERROR);


    return $output;

}
```

7.2 The Runtime Engine

All of this has been building up to one big secret: there is no Template Toolkit runtime. The Template Toolkit uses Perl as its runtime environment. So far, all of the modules we've discussed have been a complex way of turning non-Perl (the templates) into code that the Perl interpreter can execute (compile subroutines).

To see exactly what this means, we need to see what a compiled template looks like. In fact, a compiled template is just a regular Perl subroutine. Here's a very simple one:

```
sub my_compiled_template {

    return "This is a compiled template.\n";

}
```

You're unlikely to see a compiled template this simple unless you wrote it yourself, but it is entirely valid. All a template subroutine is obliged to do is return some output (which may be an empty string, of course). If it can't for some reason, it should raise an error via die:

```
sub my_todo_template {

    die "This template not yet implemented\n";

}
```

If it wants to get fancy, it can raise an error as a Template::Exception object. An exception object is really just a convenient wrapper for the type and info fields.

```
sub my_solilique_template {

    die (Template::Exception->new('yorrick', 'Fellow of infinite jest'));

}
```

Templates generally need to do a lot more than just generate static output or raise errors. They may want to inspect variable values, process another template, load a plugin, run a filter, and so on. Whenever a template subroutine is called, it gets passed a reference to a Template::Context object. It is through this context object that template code can access the features of the Template Toolkit.

We described earlier how the Template::Service object calls on Template::Context to handle a process request from the *outside*. We can make a similar request on a context to process a template, but from within the code of another template. This is a call from the *inside*:

```
sub my_process_template {

    my $context = shift;



    my $output = $context->process('header', { title => 'Hello World' })

               . "\nsome content\n"

               . $context->process('footer');

}
```

This is then roughly equivalent to a source template something like this:

```
[% PROCESS header

    title = 'Hello World'

%]

some content

[% PROCESS footer %]
```

Template variables are stored in and managed by a Template::Stash object. This is a blessed hash array in which template variables are defined. The object wrapper provides get and set methods that implement all the magical variable features of the Template Toolkit.

Each context object has its own stash, a reference to which is returned by the appropriately named stash method. So to print the value of some template variable, or, for example, to represent the following source template:

```
<title>[% title %]</title>
```

we might have a subroutine definition something like this:

```
sub {

   my $context = shift;

   my $stash = $context->stash( );

   return '<title>' . $stash->get('title') . '</title>';

}
```

The stash get method hides the details of the underlying variable types, automatically calling code references, checking return values, and performing other such tricks. If title happens to be bound to a subroutine, we can specify additional parameters as a list reference passed as the second argument to get:

```
[% title('The Cat Sat on the Mat') %]
```

This translates to the stash get call:

```
$stash->get([ 'title' => [ 'The Cat Sat on the Mat' ] ]);
```

Dotted compound variables can be requested by passing a single list reference to the get method in place of the variable name. Each pair of elements in the list should correspond to the variable name and reference to a list of arguments for each dot-delimited element of the variable. Therefore, this:

```
[% foo(1, 2).bar(3, 4).baz(5) %]
```

is equivalent to:

```
$stash->get([ foo => [1,2], bar => [3,4], baz => [5] ]);
```

If there aren't any arguments for an element, you can specify an empty, zero, or null argument list:

```
[% foo.bar %]

$stash->get([ 'foo', 0, 'bar', 0 ]);
```

The set method works in a similar way. It takes a variable name and a variable value that should be assigned to it:

```
[% x = 10 %]

$stash->set('x', 10);



[% x.y = 10 %]

$stash->set([ 'x', 0, 'y', 0 ], 10);
```

So the stash gives us access to template variables and the context provides the higher-level functionality. Alongside the process method lies the include method. Just as with the *PROCESS* and *INCLUDE* directives, the key difference is in variable localization. Before processing a template, the process method simply updates the stash to set any new variable definitions, overwriting any existing values. In contrast, the include method creates a copy of the existing stash, in a process known as *cloning* the stash, and then uses that as a temporary variable store. Any previously existing variables are still defined, but any changes made to variables, including setting the new variable values passed as arguments, will affect only the local copy of the stash (although note that it's only a shallow copy, so it's not foolproof). When the template has been processed, the include method restores the previous variable state by *decloning* the stash.

The context also provides an insert method to implement the *INSERT* directive, but doesn't provide a wrapper method. This functionality can be implemented by rewriting the Perl code and calling include:

```
[% WRAPPER foo %]

  blah blah [% x %]

[% END %]



$context->include('foo', {

   content => "\n   blah blah " . $stash->get('x') . "\n",

});
```

In addition to the template processing methods process, include, and insert, the context defines methods for fetching plugin objects (plugin) and filters (filter):

```
[% USE foo = Bar(10) %]


$stash->set('foo', $context->plugin('Bar', [10]));


[% FILTER bar(20) %]
   blah blah blah
[% END %]


my $filter = $context->filter('bar', [20]);
&$filter("\n   blah blah blah\n");
```

Pretty much everything else you might want to do in a template you can do in Perl code. Things such as IF, UNLESS, FOREACH, and so on all have direct counterparts in Perl.

```
[% IF msg %]
   Message: [% msg %]
[% END %];


if ($stash->get('msg')) {
    $output .= "\n   Message: \n";
    $output .= $stash->get('msg');
    $output .= "\n";
}
```

The best way to get a better understanding of what's going on underneath the hood is to set the $Template::Parser::DEBUG flag to a true value and start processing templates. This will cause the parser to print the generated Perl code for each template it compiles to STDERR. You'll probably also want to set the $Template::Directive::PRETTY option to have the Perl pretty-printed for human consumption (see Example 7-16).

Example 7-16. debug.pl

```
use Template;
use Template::Parser;
use Template::Directive;


$Template::Parser::DEBUG = 1;
$Template::Directive::PRETTY = 1;


my $tt = Template->new( );
$tt->process(*DATA, { cat => 'dog', mat => 'log' })
    || die $tt->error;


_ _DATA_ _
The [% cat %] sat on the [% mat %]
```

The output sent to STDOUT remains as you would expect:

```
The dog sat on the log
```

The output sent to STDERR would look something like the code shown in Example 7-17.

Example 7-17. Compiled main template document block

```
sub {
    my $context = shift || die "template sub called without context\n";
    my $stash   = $context->stash;
    my $output  = '';
    my $error;

    eval { BLOCK: {
        $output .=  "The ";
        $output .=  $stash->get('cat');
        $output .=  " sat on the ";
        $output .=  $stash->get('mat');
        $output .=  "\n";
    } };
    if ($@) {
        $error = $context->catch($@, \$output);
        die $error unless $error->type eq 'return';
    }

    return $output;
}
```

Different versions of the Template Toolkit produce slightly different code. When the compiled document is written out to disk, the Template Toolkit version is part of the compiled code, as shown in Example 7-18.

Example 7-18. A compiled document

```
#------------------------------------------------------------------------
# Compiled template generated by the Template Toolkit version 2.09c
#------------------------------------------------------------------------

Template::Document->new({
    METADATA => {
        'modtime' => '1054300677',
        'name' => 'cat.tt2',
    },
    BLOCK => sub {
        my $context = shift || die "template sub called without context\n";
        my $stash   = $context->stash;
```

```
        my $output  = '';
        my $error;

        eval { BLOCK: {
            $output .=  "The ";
            $output .=  $stash->get('cat');
            $output .=  " sat on the ";
            $output .=  $stash->get('mat');
            $output .=  "\n";
        } };
        if ($@) {
            $error = $context->catch($@, \$output);
            die $error unless $error->type eq 'return';
        }

        return $output;
    },
    DEFBLOCKS => {

    },
});
```

Constants defined in the CONSTANTS configuration option are implemented by the Template::Namespace::Constants module. If we modify *debug.pl* slightly, as shown in Example 7-19, the code produced is slightly different, as shown in Example 7-20.

Example 7-19. debug-constants.pl

```
use Template;
use Template::Parser;
use Template::Directive;

$Template::Parser::DEBUG = 1;
$Template::Directive::PRETTY = 1;

my $tt = Template->new(
    CONSTANTS => {
        cat => 'dog',
    },
);
$tt->process(*DATA, { mat => 'log' })
    || die $tt->error;
```

```
_ _DATA_ _

The [% constants.cat %] sat on the [% mat %]
```

Example 7-20. Compiled main template document block (with constant folding)

```
sub {

    my $context = shift || die "template sub called without context\n";

    my $stash   = $context->stash;

    my $output  = '';

    my $error;


    eval { BLOCK: {

        $output .=  "The ";

        $output .=  'dog';

        $output .=  " sat on the ";

        $output .=  $stash->get('mat');

        $output .=  "\n\n";

    } };

    if ($@) {

        $error = $context->catch($@, \$output);

        die $error unless $error->type eq 'return';

    }


    return $output;

}
```

Notice that [% constants.dog %] was turned into 'dog' at *compile time*, rather than at runtime. This can be a potentially huge gain, especially for templates that contain data that changes infrequently.

7.3 Module Interfaces

Now that our idea of how the Template Toolkit is put together is coming into focus, we can begin discussing the individual modules. In this section, we will describe each core component of the Template Toolkit, as well as the public interface the components present. Developers who wish to extend the Template Toolkit programmatically, or who wish to replace components with their own versions, will do well to pay close attention to the APIs exposed by the components. Most methods are illustrated with small replacement versions that extend the functionality of the component, adding debugging or other simple enhancements—but keep in mind that these are intentionally small examples. You are limited only by your imagination.

Each Template Toolkit module knows about the other modules it needs to do its job, and will create instances of these objects unless one is passed explicitly. This means that modules are loaded and instances are created on demand.

The hash containing the configuration parameters is passed to each module's new method. For example, Template::Service creates a Template::Context instance like so:

```
# In Service.pm
sub _init {
    my ($self, $config) = @_;

    # Some other configuration

    $context = $self->{ CONTEXT } = $config->{ CONTEXT }
        || Template::Config->context($config)
        || return $self->error(Template::Config->error);

    return $self;
}
```

In this case, if a Template::Context instance was part of $config, a new one would not be created. This feature is most useful for overriding settings, such as *TOLERANT*, for specific instances:

```
my $context = Template::Context->new(TOLERANT => 1);
my $tt = Template->new({
    CONTEXT => $context,
    TOLERANT => 0
});
```

7.3.1 Template's process Method

The main interface to the Template Toolkit from within Perl is through the Template module. Recall our basic script from Chapter 6, shown again in Example 7-21.

Example 7-21. ttperl.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;
```

```
my $tt    = Template->new( );
my $input = 'answer.tt';
my $vars  = {
    answer => 42,
    author => 'Douglas Adams',
};

$tt->process($input, $vars)
    || die $tt->error( );
```

Chapter 6 covered the basics of this script; let's discuss the details in more depth. The process method is where the action begins:

```
$tt->process($input, $vars)
    || die $tt->error( );
```

We pass the name of the template file that we want processed, here stored in the $input variable followed by template variables defined in $vars. We could of course pass the template filename as the literal string 'answer.tt2' and save ourselves the effort of creating a temporary variable, but we'll continue to use the $input variable in the examples that follow. As we'll see when we look more closely at the process method, the first argument doesn't always have to be a filename, so it helps to keep things deliberately vague.

The process method returns a true value if the template was sucessfully processed. The output generated will be printed to STDOUT by default, so you'll see it scrolling up your screen when you run the program.

Suppose the source template *answer.tt2* contains the text shown in Example 7-22.

Example 7-22. answer.tt2

```
The answer to the Ultimate Question of Life, the
Universe and Everything is [% answer %].

              -- [% author %]
```

Then we can expect to see the following output generated:

```
The answer to the Ultimate Question of Life, the
Universe and Everything is 42.

              -- Douglas Adams
```

If an error occurs, the process method returns false. In this case, we call the error method to find out what went wrong and report it as a fatal error using die. An error can be returned for a number of reasons, such as the file specified could not be found, had embedded directives containing illegal syntax that could not be parsed, or generated a runtime error while the template was being processed.

7.3.1.1 The process method

The Template process method is the gateway into the Template Toolkit for processing templates:

```
$tt->process($input, $vars, $output, $options)
    || die $tt->error( );
```

process takes up to four arguments: the first specifies the input; the second is a reference to a hash of variables to be made available to the template; the third specifies the destination of the output; and the fourth defines modifiers for that output destination, such as setting binmode on Windows platforms.

7.3.1.1.1 Input template

The first parameter to process specifies where the input should come from. Most often this will be the name of a file:

```
$tt->process('H2G2/entry/earth');
```

The Template Toolkit looks for the template in the directory or directories specified in the INCLUDE_PATH option. If you haven't specified INCLUDE_PATH, the Template Toolkit will look in the current working directory.

In addition to a filename, you can pass a reference to text:

```
my $text = "Hello, [% name %]!";
$tt->process(\$text);
```

or you can pass a reference to a filehandle or a typeglob; as in:

```
my $fh = IO::File->new("file.tmpl") or die $!;
$tt->process($fh);
```

or, as in:

```
$tt->process(\*STDIN);
```

Because the Template Toolkit can read from a filehandle, a quick and easy way to pass a template to process is via a reference to the DATA filehandle. (The DATA filehandle contains everything in the current file after the special marker _ _DATA_ _.) This can simplify writing single-usage scripts and tests greatly, as shown in Example 7-23.

Example 7-23. hello.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use Template;

my $tt = Template->new;
$tt->process(\*DATA) or die $tt->error( );

_ _DATA_ _
Hello, world!
```

7.3.1.1.2 Template variables

The second, optional argument to the process method is a reference to a hash defining template variables and corresponding values. The Template Toolkit allows you to bind almost any kind of Perl data to template variables, including scalars, arrays, hashes, subroutines, and objects. The code in Example 7-24 contains examples of all of these.

Example 7-24. Template variables

```
my $vars = {
  name    => 'Arthur Dent',
  planet  => 'Earth',
  friends => [ 'Ford Prefect', 'Slartibartfast' ],
  people  => {
    'Erotica Gallumbits' => {
```

```
            description => 'Triple breasted whore',
            location    => 'Erotican 6',
        },
        'Bugblatter Beast' => {
            description => 'Ravenous (but stupid)',
            location    => 'Traal',
        },
        'Hotblack Desiato' => {
            description => 'Dead (for tax purposes)',
            location    => 'Milliways',
        },
    },
    consult_guide => sub {
        my $arg = shift;
        return "Don't panic, $arg!";
    },
    magrethea => Acme::Planet->new(name  => 'Magrethea',
                                edges => 'Crinkly'),
};

$tt->process($input, $vars)
    || die $tt->error( );
```

Internally, these variables are incorporated into the Template::Stash instance that is made available via the Template::Context object.

7.3.1.1.3 Redirecting template output

The default behavior for the process method is to print the output generated by processing a template to STDOUT. The third argument to the process method can be used to specify an alternate destination for the output.

When a plain string is passed as the third argument, it indicates a filename to which output should be written. The OUTPUT_PATH option must be defined to specify a root directory for generating output files. The file specified will be located relative to this directory (see Example 7-25).

Example 7-25. Redirecting Template output to a file

```
my $tt = Template->new(OUTPUT_PATH => '/tmp');


$tt->process($input, $vars, 'output.html')
    || die $tt->error( );
```

In this example, the output will be written to the */tmp/output.html* file.

A reference to a string can instead be passed as the third argument. In this case, the output will be appended to the string. The process method doesn't clear any existing value that the string has (see Example 7-26).

Example 7-26. Redirecting Template output to a scalar

```
my $output;

$tt->process($input, $vars, \$output)
    || die $tt->error( );

print $output;
```

A reference to an array can also be passed as the third argument. The output will be added as an item to the end of the list, as shown in Example 7-27.

Example 7-27. Redirecting Template output to an array

```
my @output;

for my $file (qw( header body footer )) {
    $tt->process($file, $vars, \@output)
        || die $tt->error( );
}

print @output;
```

Another option is to pass a reference to a filehandle that is open and ready for output, as shown in Example 7-28.

Example 7-28. Redirecting Template output to a filehandle

```
use File::Temp qw(tempfile);
my ($fh) = tempfile( );

$tt->process($input, $vars, $fh)
    || die $tt->error( );
```

Yet another option for the third argument is to pass a reference to a subroutine. The subroutine will be called with the output passed to it as the first argument (see Example 7-29).

Example 7-29. Redirecting Template output to a subroutine

```
sub process_to_db {
    my $content = shift;
    $dbh->do("INSERT INTO content (id, content) VALUES (NULL, ?)",
        undef, $content);
}

$tt->process($input, $vars, \&process_to_db)
    || die $tt->error( );
```

The final option for the third argument is to pass a reference to an object that implements a print method. This includes the Apache::Request object and those derived from IO::Handle, for example. The print method will be called with the output passed as the first argument, as per subroutines (see Example 7-30).

Example 7-30. Redirecting Template output to an object with a print method

```
my $fh = IO::File->new(">$tmpfile");

$tt->process($input, $vars, $fh)

    || die $tt->error( );
```

The OUTPUT configuration option can also be used to set the output destination for the Template module as a whole. It can be set to any of the same values as the third argument to process. When a third argument is passed to process, it will override any value defined in OUTPUT (see Example 7-31).

Example 7-31. Using the OUTPUT configuration option

```
my $tt = Template->new(OUTPUT => \$output);



$tt->process($input, $vars)

    || die $tt->error( );
```

This is functionally equivalent to the code in Example 7-32.

Example 7-32. process equivalent of OUTPUT

```
my $tt = Template->new( );



$tt->process($input, $vars, \$output)

    || die $tt->error( );
```

7.3.1.1.4 Processing options

The fourth argument to process is an optional reference to a hash array of processing options. There's only one option at present, binmode, but there's a chance that others will be added at some later date, and this is where they'll go. Example 7-33 shows the code for setting processing options.

Example 7-33. Setting processing options

```
$tt->process($in, $vars, $out, { binmode => 1 })

    || die $tt->error( );
```

The binmode option is typically used on the Windows platform to ensure that line endings are correctly preserved as \r\n instead of being transformed into \n, which is the standard for Unix and other platforms (except Mac OS, which uses \r just to confuse matters). Example 7-34 shows the code for setting binmode on a filehandle.

Example 7-34. Setting binmode on a filehandle

```
local *FH;

open FH, $filename;

binmode FH;
```

For convenience, you can also specify processing options as a list of arguments, as shown in Example 7-35.

Example 7-35. Setting processing options using a list

```
$tt->process($in, $vars, $out, binmode => 1)

    || die $tt->error( );
```

7.3.1.2 The error method

If the process method returns a false value, the error method can be called to return a reference to a *Template::Exception* object that encapsulates information about the error. The exception object has type and info methods that return a short string identifying the kind of error that occurred (e.g., parse, file, etc.), and a message containing further information, respectively. Example 7-36 shows the code for reporting process errors.

Example 7-36. Reporting process errors

```
unless ($tt->process($input, $vars)) {

    my $error = $tt->error( );

    print "error type: ", $error->type( ), "\n";

    print "error info: ", $error->info( ), "\n";

}
```

The nice thing about this object is that you don't need to do anything special with it. You can just print the object and leave the magical stringification method as_string to generate a printable representation of the error. Hence the idiom should be familiar by now (see Example 7-37).

Example 7-37. Error-reporting idiom

```
$tt->process('no/such/page', $vars)

    || die $tt->error( );
```

The message generated is of the form $type error - $info (see Example 7-38).

Example 7-38. Error example

```
file error - no/such/page not found
```

7.3.2 Template::Config

Template::Config provides a factory method for each major component of the Template Toolkit—context, filters, iterator, parser, plugins, provider, service, stash, and constants (see Example 7-39). The type of object that each method creates is, in turn, controlled by a series of variables in the $Template::Config namespace.

Example 7-39. Template::Config package variables

```
$CONTEXT   = 'Template::Context';

$FILTERS   = 'Template::Filters';

$ITERATOR  = 'Template::Iterator';

$PARSER    = 'Template::Parser';

$PLUGINS   = 'Template::Plugins';

$PROVIDER  = 'Template::Provider';

$SERVICE   = 'Template::Service';

$STASH     = 'Template::Stash';

$CONSTANTS = 'Template::Namespace::Constants';
```

These are set when the Template Toolkit is installed; some of them might differ based on how the installation was performed. For example, the fast XS-based stash (Template::Stash::XS) might have been installed instead of the default stash.

Each method works in basically the same way; Example 7-40 shows provider, by way of example.

Example 7-40. Template::Config::provider

```
sub provider {
    my $class  = shift;
    my $params = defined($_[0]) && UNIVERSAL::isa($_[0], 'HASH')
            ? shift : { @_ };

    return undef unless $class->load($PROVIDER);
    return $PROVIDER->new($params)
        || $class->error("failed to create template provider: ",
                        $PROVIDER->error);
}
```

$PROVIDER, as we just saw, defaults to Template::Provider, but it should be apparent that this can be changed to another class:

```
use Template::Config;
$Template::Config::PROVIDER = 'TTBook::Template::Provider';

my $tt = Template->new( ) || die Template->error( );
```

The provider that gets instantiated is going to be a TTBook::Template::Provider, not a Template::Provider.

7.3.2.1 load

Template::Config provides a general module-loading method, load, which takes a name (such as *TTBook::Template::Config*) and loads the module, using require. It returns undef if there were problems loading the module; the error is available via Template::Config->error.

7.3.2.2 preload

preload will load all of the defined components (based on the contents of the variables $SERVICE, $PROVIDER, etc.), mostly for the benefit of long-running processes, such as mod_perl. For example, it is automatically called by the Template frontend when $ENV{'MOD_PERL'} is set:

```
# Template.pm
# preload all modules if we're running under mod_perl
Template::Config->preload( ) if $ENV{ MOD_PERL };
```

preload can be called with extra module names as well, so it can be used to load custom modules:

```
Template::Config->preload('TTBook::Template::Provider',
                          'TTBook::Template::Plugin::NNTP');
```

7.3.2.3 instdir

This helper method returns the directory in which the optional components were installed, such as /usr/local/tt2 or *C:/Template Toolkit 2*. If the optional components were not installed, instdir returns undef and sets $ERROR.

For example, to add the Spash! templates that come with the Template Toolkit to your INCLUDE_PATH, which are installed in *$instdir/templates/spash*, use this code:

```
my $splash = Template::Config->instdir('templates/splash')
    || die Template::Config->error;

my $tt = Template->new(INCLUDE_PATH => [ $splash ]);
```

7.3.3 Template::Constants

Template::Constants defines the constants used and returned by the other elements of the Template Toolkit. Symbols can be imported into your module in the usual way:

```
use Template::Constants qw( :status );
```

7.3.3.1 :status

The status constants are used to check the results of certain operations. The following symbols are imported as part of :status:

```
STATUS_OK           # ok
STATUS_RETURN       # ok, block ended by RETURN
STATUS_STOP         # ok, stopped by STOP
STATUS_DONE         # ok, iterator done
STATUS_DECLINED     # ok, declined to service request
STATUS_ERROR        # error condition
```

Example 7-41, from the insert method of Template::Context, illustrates how the status codes are used; we are iterating through all available providers until one of them successfully loads the template whose name is stored in $name.

Example 7-41. Using ERROR constants

```
foreach my $provider (@$providers) {
    ($text, $error) = $provider->load($name, $prefix);
    next FILE unless $error;
    if ($error =  = Template::Constants::STATUS_ERROR) {
        $self->throw($text) if ref $text;
        $self->throw(Template::Constants::ERROR_FILE, $text);
    }
}
$self->throw(Template::Constants::ERROR_FILE, "$file: not found");
```

7.3.3.2 :error

The *ERROR_** status codes are primarily used when things go wrong. All Template::Exception objects are instantiated with one of these error codes as the type field.

The error constants are:

```
ERROR_RETURN        # return a status code
ERROR_FILE          # file error: I/O, parse, recursion
ERROR_VIEW          # view error
ERROR_UNDEF         # undefined variable value used
```

```
ERROR_PERL          # error in [% PERL %] block

ERROR_FILTER        # filter error

ERROR_PLUGIN        # plugin error
```

7.3.3.3 :chomp

The :chomp symbol imports the whitespace-related constants *CHOMP_NONE*, *CHOMP_ALL*, and *CHOMP_COLLAPSE*. These can be used when specifying a value for the *PRE_CHOMP* and *POST_CHOMP* configuration options:

```
use Template::Constants qw( :chomp );


my $tt = Template->new(TRIM => CHOMP_COLLAPSE);
```

The chomp constants are:

```
CHOMP_NONE          # do not remove whitespace

CHOMP_ALL           # remove whitespace

CHOMP_COLLAPSE      # collapse whitespace to a single space
```

7.3.3.4 :debug

The *DEBUG_** constants let you debug specific core components and not others. These constants are imported with the :debug tag, and include the following:

```
DEBUG_OFF           # do nothing

DEBUG_ON            # basic debugging flag

DEBUG_UNDEF         # throw undef on undefined variables

DEBUG_VARS          # general variable debugging

DEBUG_DIRS          # directive debugging

DEBUG_STASH         # general stash debugging

DEBUG_CONTEXT       # context debugging

DEBUG_PARSER        # parser debugging

DEBUG_PROVIDER      # provider debugging

DEBUG_PLUGINS       # plugins debugging

DEBUG_FILTERS       # filters debugging

DEBUG_SERVICE       # context debugging

DEBUG_ALL           # everything

DEBUG_CALLER        # add caller file/line info
```

These constants are binary OR-ed together to produce a bitmask that specifies the components to debug. For example, to debug the service, context, and provider, use the code in Example 7-42.

Example 7-42. Using constants from Perl

```
use Template;
use Template::Constants qw( :debug );

my $debug = DEBUG_SERVICE | DEBUG_CONTEXT | DEBUG_PROVIDER;
my $tt = Template->new(DEBUG => $debug);
$tt->process("test.tt2") || die $tt->error( );
```

Processing a simple test template, *test.tt2*, yields debugging information for the service, context, and provider objects, as expected:

```
[Template::Provider] creating cache of unlimited slots for [ . ]
[Template::Service] process(test.tt2, <no params>)
[Template::Context] template(test.tt2)
[Template::Context] looking for block [test.tt2]
[Template::Context] asking providers for [test.tt2] [  ]
[Template::Provider] _fetch_path(test.tt2)
[Template::Provider] searching path: ./test.tt2
[Template::Provider] _load(./test.tt2, test.tt2)
[Template::Provider] _compile(HASH(0x823cf1c), <no compfile>)
[Template::Provider] _store(./test.tt2, Template::Document=HASH(0x829f4a8))
[Template::Provider] adding new cache entry
[Template::Service] PROCESS: Template::Document=HASH(0x829f4a8)
[Template::Context] process([ Template::Document=HASH(0x829f4a8) ], <no params>, <unlocalized>)
[Template::Context] template(Template::Document=HASH(0x829f4a8))
```

Using these DEBUG flags, it is possible to debug individual components. Adding the *DEBUG_CALLER* mask causes the debugging messages to include the filename and line number:

```
my $debug = DEBUG_SERVICE |  DEBUG_CALLER;
my $tt = Template->new(DEBUG => $debug);
$tt->process("test.tt2") || die $tt->error( );


[Template::Provider] creating cache of unlimited slots for [ . ] at /usr/local/lib/perl5/
site_perl/5.6.1/Template/Provider.pm line 350
[Template::Service] process(test.tt2, <no params>)
[Template::Context] template(test.tt2) at /usr/local/lib/perl5/site_perl/5.6.1/Template/
Context.pm line 81
...
```

7.3.4 Template::Base

Template::Base implements a common base class used by almost all of the other Template Toolkit modules. Template::Base implements a few important methods that the other modules inherit, namely new, error, and debug. Template::Base has also made its way to CPAN, with slight variations and enhancements, as Class::Base (http://search.cpan.org/dist/Class-Base/).

7.3.4.1 new

When new is called on an object, it invokes the class's _init method, which is where instance-specific initialization takes place. The new method handles the folding of name => value pairs into a single hash; a reference to this hash is passed to the other modules. This is why objects can be created with either a series of name-value pairs or a hashref:

```
my %opts = (
    INCLUDE_PATH => \@paths,
    ANYCASE => 1,
);
my $tt1 = Template->new(\%opts);
my $tt2 = Template->new(%opts);
```

Both invocations are valid and produce similar instances.

7.3.4.2 error

If something goes wrong, most public methods return undef. When this happens, the error message can be retrieved by calling the error method on the instance:

```
$tt->process($template, \%vars)
    || die $tt->error;
```

The error method behaves analogously for classes as well:

```
my $tt = Template->new(\%opts)
    || die Template->error;
```

If error is called with arguments, these arguments become the current error value, and the call to error returns undef, as shown in Example 7-43.

Example 7-43. TTBook::Template::Plugin::LDAP

```
package TTBook::Template::Plugin::LDAP;


use strict;
use Net::LDAP;


sub new {
    my ($self, $context, $host) = @_;


    return $self->error("Missing required host")
        unless ($host);


    my $ldap = Net::LDAP->new($host)
        || return $self->error("Error connecting to $host: $@");


    $ldap->bind;
    return $ldap;
}
```

This short example implements a basic Net::LDAP plugin, which dies if it is not passed a host to which to connect. It also dies if there is a problem connecting to the host.

7.3.4.3 debug

debug generates a debugging message by concatenating all arguments passed into a string and printing it to STDERR. A prefix is added to indicate the module of the caller. This Template::Context subclass emits debugging information whenever a filter is defined using the context's define_filter method. To use these subclasses of standard modules, remember to set the appropriate $Template::Config variable to the name of the class to be used. In Example 7-44, we're setting $Template::Config::CONTEXT to be TTBook::Template::Context::Debugging.

Example 7-44. TTBook::Template::Context::Debugging

```
package TTBook::Template::Context::Debugging;

use base qw(Template::Context);

sub define_filter {
    my ($self, $name, $filter, $is_dynamic) = @_;

    $self->debug(sprintf "defining %s filter '%s'",
                    $is_dynamic ? "dynamic" : "static",
                    $name);

    return $self->SUPER::define_filter($name, $filter, $is_dynamic);
}
```

Given a simple test template of:[1]

[1] We know that the wrap plugin defines a static filter; see Chapter 8.

```
[% USE wrap %]
```

we get this on STDERR:

```
[Template::Context::Debugging] defining static filter 'wrap'
```

debug itself does not check to see whether the module is currently in debugging mode (as specified by the caller via the *DEBUG* configuration option), but $self->{DEBUG} will be set to a true value if debugging was requested. Our debug call should look like this:

```
$self->debug(sprintf "defining %s filter '%s'",
            $is_dynamic ? "dynamic" : "static",
            $name)
    if $self->{ DEBUG };
```

7.3.5 Template::Context

The Template::Context module defines an object class for representing a runtime context in which templates are processed. It provides an interface to the fundamental operations of the Template Toolkit processing engine through which compiled templates can process templates, load plugins and filters, raise exceptions, and so on.

Plugins and dynamic filters are passed a reference to the current context when they are invoked. This reference can then be used to invoke any of the context's methods, such as define_filter or include.

7.3.5.1 stash

This method returns a reference to the stash (see the sectionSection 7.1.3 earlier in this chapter):

```
my $stash = $context->stash;
```

This reference can then be used to get or set values, which are accessible from templates in the usual way:

```
$stash->set('arp', "with or without is the different");


# In the template:

[% arp %]
```

If you get access to the stash while you are within an *INCLUDE*d template, the stash you get will be the localized one; changes made to this stash will not persist to outer scopes (unless the changes are made to nested structures).

7.3.5.2 insert, include, and process

The context provides methods such as include, process, and insert, which implement the *INCLUDE*, *PROCESS*, and *INSERT* directives. For example, a *PROCESS* directive such as:

```
[% PROCESS box quote = 'A city is like a large, complex, rabbit' %]
```

is translated by the Template::Directive class into something like this:

```
$context->process('box', { 'quote' => 'A city is like a large complex rabbit' });
```

7.3.5.3 template

When a template is specified by name, the context instance queries its internal list of Template::Provider instances, using the template method:

```
my $doc = $context->template($name)

    || die $context->error;
```

$doc will be a Template::Document instance, which, as mentioned earlier, is basically an object wrapper around a compiled subroutine (see the Section 7.3.13, earlier in this chapter). If a template can't be loaded for whatever reason, template returns undef, and the error is available via the error method.

7.3.5.4 plugin and filter

The plugin method uses one or more Template::Plugins objects to load plugins specified by USE, and the filter method uses the Template::Filters objects to fulfill FILTER requests. A simple USE statement, such as:

```
[% USE CGI %]
```

is transformed into something like:

```
$stash->set('CGI', $context->plugin('CGI'));
```

A more complex example, such as:

```
[% USE q = CGI('name=darren&title=JAPH') %]
```

becomes more or less what you would expect:

```
$stash->set('q', $context->plugin('CGI', [ 'name=darren&title=JAPH' ]));
```

Arguments supplied to a plugin are passed as a reference to an array. Named arguments are passed in a hashref, as the last element in the array:

```
[% USE MP3('Got the Time.mp3'

      dir = 'Joe Jackson/Look Sharp!'

      utf8 = 1) %]
```

Reformatted slightly, the resulting Perl code is:

```
$stash->set('MP3',
          $context->plugin('MP3',
                        [ 'Got the Time.mp3',
                          { 'dir' => 'Joe Jackson/Look Sharp!',
                            'utf8' => 1
                          }
                        ]));
```

Note that if a name is not specified to USE, the name of the plugin itelf is used.

Filters are handled similarly. The filter method of the context fetches a filter (using the Template::Filters instance), using the filter method. A simple text string, filtered through upper:

```
[% 'do not leave it is not real' | upper %]
```

turns into this Perl:

```
my $filter = $context->filter('upper')
         || $context->throw($context->error);


$output .= 'do not leave it is not real';


&$filter($output);
```

The upper filter is a static filter, so there isn't much interesting going on there: the filter method calls on the Template::Filters instances to load the filter subroutine. If this fails, the throw method creates a new Template::Exceptions instance and passes it up. Otherwise, the subroutine reference gets assigned to $filter, and we invoke filter on the text waiting to be filtered.

Dynamic filters get passed arguments, which are collected and passed in the same way for filters as they are for plugins:

```
[% FILTER format("%.12f");
    PI = 22 / 7;
    radius = 14.5;
    PI * radius * radius;
   END
%]
```

Arguments are passed as a reference to an array:

```
my $filter = $context->filter('format', [ '%.12f' ])
         || $context->throw($context->error);


$stash->set('PI', 22 / 7);
$stash->set('radius', 14.5);
$output .=  $stash->get('PI') * $stash->get('radius') * $stash->get('radius');


&$filter($output);
```

7.3.5.5 define_filter

Use this method to define a filter:

```
use Term::ANSIColor qw(colored);

$context->define_filter('red', sub { colored($_[0], "red") }, 0);
```

Pass the name of the filter, a reference to the filter sub, and a boolean indicating whether the filter is a dynamic or static filter. This filter becomes available immediately.

7.3.6 Template::Provider

The Template::Provider is used to load, parse, compile, and cache templates. This object may be subclassed to provide more specific facilities for loading or otherwise providing access to templates.

The Template::Context objects maintain a list of Template::Provider objects that are polled in turn (via fetch) to return a requested template. Each may return a compiled template, raise an error, or decline to serve the request, giving subsequent providers a chance to do so.

This is the "Chain of Responsibility" pattern. See *Design Patterns*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addision-Wesley), for further information.

Template::Provider has a few interesting methods, described in the next sections.

7.3.6.1 fetch($name)

fetch returns a compiled template for $name. If the template cannot be found, (undef, STATUS_DECLINED) is returned. If an error occurs (e.g., read error, parse error), ($error, STATUS_ERROR) is returned, where $error is the error message generated. If the TOLERANT flag is set, the method returns (undef, STATUS_DECLINED) instead of returning an error.

Template::Provider can also be used as a general-purpose file loader. Because a normal text file (without delimiters) is a valid template, any file can be loaded via the fetch method (see Example 7-45).

Example 7-45. Using Template::Provider for non-Template Toolkit files

```
my $prov = Template::Provider->new(ABSOLUTE => 1);

my $file = "/etc/passwd";



my ($doc, $error) = $prov->fetch($file);

die "Couldn't load $file" if defined $error;
```

As noted earlier, fetch returns a pair of values: the Template::Document instance and an error string. Only one of the two will be defined: if there was an error loading the file, $error will contain STATUS_DECLINED (from Template::Constants), and $doc will be undefined; if the file was loaded without incident, $error will be undefined and $doc will contain the Template::Document instance, which will have modtime and name methods, at the very least:

```
printf "%s was last modified on %s.\n",

    $doc->name( ), $doc->modtime( );
```

The modtime method returns the number of seconds since the epoch, which can be passed to localtime to get a more meaningful value:

```
printf "%s was last modified on %s.\n",

    $doc->name( ), localtime($doc->modtime( ));
```

More interesting formatting is possible using POSIX::strftime:

```
use POSIX qw(strftime);

my @date = localtime($doc->modtime( ));

printf "%s was last modified on %s.\n",

    $doc->name( ), strftime("%Y/%m/%d", @date);
```

This might return, for example:

```
/etc/passwd was last modified on 2002/10/18.
```

7.3.6.2 store($name, $template)

This method stores the compiled template, $template, in the cache as $name. Subsequent calls to fetch($name) will return this template in preference to any disk-based file.

7.3.6.3 paths

paths expands the object's INCLUDE_PATHS and returns a reference to an array of pathnames. Since Version 2.08 of the Template Toolkit, elements of INCLUDE_PATH can be subroutine references or objects, and paths will correctly call and expand these references.

```
package TTBook::Template::Provider::ExpandPaths;


use strict;

use base qw(Template::Provider);


sub paths {

    my $self = shift;

    my $orig_paths = $self->SUPER::paths( );

    my ($path, @paths, %unique);


    for $path (@$orig_paths) {

        my @chunks = split '/', $path;


        while (@chunks) {

            push @paths, join '/', @chunks;

            pop @chunks;

        }

    }


    # Remove duplicates from the list

    @paths = grep { ++$names{$_} =  = 1 } grep { length } @paths;


    return \@paths;

}
```

TTBook::Template::Provider::ExpandPaths will expand each element of @orig_paths into a list consisting of expanded versions of @orig_paths. For example, given an INCLUDE_PATH of /web/www/html:/web/search/html, this provider will return a reference to this array:

```
('/web/www/html',

 '/web/www',

 '/web',

 '/web/search/html',

 '/web/search')
```

Using this provider allows a user to situate templates anywhere along the INCLUDE_PATH, which means that they can be shared. For example, general headers and footers can be located in */web*, while specific subdirectories could implement their own header and/or footer simply by placing a file somewhere along the search path.

7.3.7 Template::Stash

The most common thing that a template needs to do is to access variables. This is where the stash comes in. As we saw earlier, the stash manages the variables that are available to templates and implements the dot (.) operator.

7.3.7.1 get, set

Template variables are stored in and managed by a Template::Stash object. This is a blessed hash array in which template variables are defined:

```
my $stash = Template::Stash->new({
    planet => 'Earth',
    about  => 'Mostly harmless'
 });
```

The object wrapper provides get and set methods that implement all the magical variable features of the Template Toolkit.

Each context object has its own stash, a reference to which can be returned by the appropriately named stash method. So to print the value of some template variable, or, for example, to represent the following source template:

```
<entry>[% planet %]</entry>
<about>
[% about %]
</about>
```

we might have a subroutine definition something like this:

```
sub {
    my $context = shift;
    my $stash   = $context->stash( );
    return '<entry>' . $stash->get('planet') . "</entry>\n"
         . "<about>\n"  . $stash->get('about')  . "\n</about>\n";
}
```

The get method retrieves the variable named by the first parameter:

```
$value = $stash->get('planet');
```

Dotted compound variables can be requested by passing a single list reference to the get method in place of the variable name. Each pair of elements in the list should correspond to the variable name and reference a list of arguments for each dot-delimited element of the variable.

```
[% guide.entry(314159).about %]


$stash->get([ 'guide', 0, 'entry', [ 314159 ], 'about', 0 ]);
```

If there are no arguments for an element, you can specify an empty, zero, or null argument list:

```
[% hitchhiker.name %]


$stash->get([ 'hitchhiker', 0, 'name', 0 ]);
```

The set method works in a similar way. It sets the variable named in the first parameter to the value specified in the second:

```
[% x = 10 %]

$stash->set('x', 10);
```

Dotted compound variables may be specified as per get:

```
[% x.y = 10 %]

$stash->set([ 'x', 0, 'y', 0 ], 10);
```

If the third parameter evaluates to a true value, the variable is set only if it did not have a true value before. This implements the behavior of the DEFAULT directive:

```
$stash->set('about', 'This page intentionally left blank.', 1);
```

7.3.7.2 clone, declone

The stash has clone and declone methods that are used by the template processor to make temporary copies of the stash for localizing changes made to variables. This localization takes place for INCLUDE directives (but not PROCESS). Conceptually, INCLUDE looks like this:

```
$stash = $stash->clone( );
$content->process($template);
$stash = $stash->declone( );
```

The clone method creates and returns a new Template::Stash object that represents a localized copy of the parent stash. Variables can be freely updated in the cloned stash; when declone is called, the original stash is returned with all its members intact and in the same state as they were before clone was called.

For convenience, a hash of parameters may be passed into clone that are used to update any simple variable (i.e., those that don't contain any namespace elements, such as foo and bar but not foo.bar) while cloning the stash. For adding and updating complex variables, the set method should be used after calling clone. This will correctly resolve and/or create any necessary namespace hashes.

The declone method returns the original stash and is used to restore the state of a stash as described earlier.

7.3.8 Template::Filters

The Template::Filters module implements a provider for creating and/or returning subroutines that implement the standard filters. As is done with its brother Template::Provider, the context keeps an array of Template::Filters instances handy for fetching filters. The filter method of the context iterates through these instances and calls the fetch method on them, passing the name of the desired filter, until one of them returns a nonerror value:

```
# Context.pm (simplified)
sub filter {
    my ($self, $name, $args) = @_;
    my ($filter, $error);

    foreach my $provider (@{ $self->{ LOAD_FILTERS } }) {
        ($filter, $error) = $provider->fetch($name, $args, $self);
        last unless $error;
    }

    return $filter;
}
```

7.3.8.1 new

The contructor for Template::Filters receives the *FILTERS* option, which should be a hashref of name => filter sub pairs. These filters become part of the instance, and calls to fetch look in this list of filters in addition to the standard filters.

```
use Text::Soundex qw(soundex);

use Text::Metaphone qw(Metaphone);


my $tf = Template::Filters->new({

    FILTERS => {

        soundex => sub { soundex($_[0]) },

        metaphone => sub { Metaphone($_[0]) },

    }

});
```

The soundex and metaphone filters can now be used like any other filter:

```
[% PROCESS page | metaphone %]
```

7.3.8.2 fetch

The main method that Template::Filters implements is fetch, as illustrated earlier. fetch will be called with three arguments: the name of the filter being requested (which should be either one of the standard filters or a filter defined in the FILTERS option passed to new); a reference to an array of configuration parameters; and the current Template::Context instance.

7.3.8.3 store

Use store to store a new filter:

```
$filters->store('soundex', sub { soundex($_[0]) });
```

This is what is called by the context's define_filter method. You should probably use define_filter if you are installing a new filter because the context will always install the new filter in the right place. If you are creating a replacement for Template::Filters, you might want to implement store differently. For example, the Template::Filters subclass TTBook::Template::Filters::Logging logs when a filter is fetched or stored, as shown in Example 7-46.

Example 7-46. TTBook::Template::Filters::Logging

```
package TTBook::Template::Filters::Logging;


use strict;

use base qw( Template::Filters );


use Template::Filters;


# Store the filter, and store the time

sub store {

    my ($self, $name, $filter) = @_;

    my $now = time;
```

```
    $self->SUPER::store($name, $filter);
    $self->{ FILTER_TIMESTAMPS }->{ $name } = $now;

    $self->debug("store($name => $filter) at $now");

    return 1;
}

# Keeps track of the difference in time between when the filter
# was stored and when it was first used.
sub fetch {
    my ($self, $name, $args, $context) = @_;
    my ($filter_sub, $filter_ts, $now);

    $filter_sub = $self->SUPER::fetch($name, $args, $context);
    $filter_ts = $self->{ FILTER_TIMESTAMPS }->{ $name };
    $now = time;

    $self->debug("fetch($name) at $now");

    return $filter_sub;
}
```

The simple Template::Filters subclass shown in Example 7-47 counts the number of times each filter is fetched.

Example 7-47. TTBook::Template::Filters::Counting

```
package TTBook::Template::Filters::Counting;

use strict;
use base qw( Template::Filters );

sub fetch {
    my ($self, $name, $args, $context) = @_;
    my $count = $self->{ FILTERS_COUNT } ||= { };

    $count->{ $name }++;
    $self->debug("filter $name has been loaded $count->{$name} times.");
```

```
    return $self->SUPER::fetch($name, $args, $context);
}
```

7.3.9 Template::Plugin

The Template::Plugin module provides both an API and a base class for plugins that implement the three basic methods that are required for a plugin to be loaded by the Template::Plugins module: load, new, and error. All the standard plugins inherit from Template::Plugin. By default, a Template::Plugin-based module has no functionality other than to load correctly; subclasses may override these and of course, can implement any other methods they need to perform their duties.

7.3.9.1 load

This method is called when the plugin module is first loaded. It is called as a package method and thus implicitly receives the package name as the first parameter. A reference to the Template::Context object loading the plugin is also passed. The default behavior for the load method is to simply return the class name; the calling context then uses this class name to call the new package method:

```
package MyPlugin;


sub load {              # called as MyPlugin->load($context)
    my ($class, $context) = @_;
    return $class;      # returns 'MyPlugin'
}
```

7.3.9.2 new

This method is called to instantiate a new plugin object for the *USE* directive. It is called as a package method against the class name returned by load. A reference to the Template::Context object creating the plugin is passed, along with any additional parameters specified in the *USE* directive.

```
sub new {               # called as MyPlugin->new($context)
    my ($class, $context, @params) = @_;
    bless {
        _CONTEXT => $context,
        _PARAMS => \@params,
    }, $class;          # returns blessed MyPlugin object
}
```

7.3.9.3 error

This method, inherited from the Template::Base module, is used for reporting and returning errors. It can be called as a package method to set/return the $ERROR package variable, or as an object method to set/return the object's _ERROR member. When called with an argument, it sets the relevant variable and returns undef. When called without an argument, it returns the value of the variable.

```
sub new {
    my ($class, $context, $dsn) = @_;


    return $class->error('No data source specified')
        unless $dsn;
```

```
    bless {
        _DSN => $dsn,
    }, $class;
}

...

my $something = MyModule->new( )
    || die MyModule->error( );

$something->do_something( )
    || die $something->error( );
```

The Template::Plugins object that handles the loading and use of plugins calls the new and error methods against the package name returned by the load method. In pseudocode terms, it looks something like this:

```
$class  = MyPlugin->load($context);       # returns 'MyPlugin'

$object = $class->new($context, @params)  # MyPlugin->new(...)
    || die $class->error( );              # MyPlugin->error( )
```

The load method may alternately return a blessed reference to an object instance. In this case, new and error are then called as *object* methods against that prototype instance. Example 7-48 provides a concrete illustration: this plugin implements a print service.

Example 7-48. TTBook::Template::Plugin::Printer

```
package TTBook::Template::Plugin::Printer;

use strict;
use vars qw($PRINTER $SERVER);
use base qw(Template::Plugin);

use Template::Plugin;
use Template::Exception;
use Net::Printer;

$PRINTER = "jeckyl";
$SERVER = "mr-hyde";

sub load {
    my ($class, $context) = @_;
    my $printer = Net::Printer->new(printer => $PRINTER,
```

```
                         server  => $SERVER);
    my $self = bless {
        _CONTEXT => $context,
        _PRINTER => $printer,
    }, $class;

    return $self;
}

sub new {
    my ($self, $context) = @_;
    return $self;
}

sub print {
    my ($self, $data) = @_;
    my ($printer, $context) = @$self{ qw( _PRINTER _CONTEXT) };

    my $result = $printer->printstring($data);
    $context->throw('printer', $result)
        unless (int($result) =  = 1);

    return "";
}

1;
```

In this example, we have implemented a Singleton plugin. One instance of TTBook::Template::Plugin::Printer gets created when load is called, and it simply returns itself for each call to new.

Because calls to print throw printer exceptions if there is a problem, they should be wrapped in TRY / CATCH blocks, as shown in Example 7-49.

Example 7-49. The Printer plugin

```
[% USE Printer %]
[% TRY %]
    [% Printer.print(data) %]
[% CATCH printer %]
    There was an error printing: [% error %]
[% END %]
```

7.3.10 Template::Plugins

Template::Plugins defines a plugins provider. It is used in almost the same way as Template::Filters and has a similar interface. The Template Toolkit allows multiple plugin providers, again using the "Chain of Responsibility" pattern.

7.3.10.1 new

The new construtor method handles the *PLUGIN* configuration option, which should be a hashref of name => plugin module pairs:

```
my $tp = Template::Plugins->new({
    PLUGINS => {
        'css' => 'TTBook::Template::Plugin::CSS',
        'javascript' => 'TTBook::Template::Plugin::JS',
    },
});
```

These newly defined plugins are stored in the instance, which is where fetch looks first when trying to load plugins. new also stores the *PLUGIN_BASE* and *LOAD_PERL* options, if present. These options affect how fetch finds plugins.

7.3.10.2 fetch

fetch is called by the context's plugin method, in the same way as the filter provider's fetch method gets called from the filter method. fetch is called with the name of the plugin, a reference to an array of parameters, and the current context, and is expected to return a blessed object, which is used in the templates.

The *PLUGIN_BASE* configuration option defines a relative base for loading plugins. If a plugin cannot be loaded by name from *PLUGINS*, each element in *PLUGIN_BASE* (which should be a reference to an array) is prepended to the name, in turn, until the plugin is found or the list exhausted. Template::Plugin is always appended to this list.

The LOAD_PERL configuration option tells the plugin's provider that standard Perl modules can be treated as plugins, after the list of known plugins has been checked and the PLUGIN_BASE search path exhausted. For example, to load the WWW::Wikipedia module, set LOAD_PERL to 1 and use:

```
[% USE wiki = WWW.Wikipedia %]
```

There is no standard WWW.Wikipedia plugin, so the plugins provider will try to load WWW::Wikipedia. Modules loaded this way must have a new method; the result of calling this method is what is returned by the call to fetch.

Given a two-element PLUGIN_BASE and LOAD_PERL:

```
my $tt = Template::Plugins->new({
    PLUGIN_BASE => [ 'TTBook::Template::Plugin',
                     'MyOrg::Template::Plugin' ],
    LOAD_PERL => 1,
});
```

and a simple *USE* statement:

```
[% USE Monitor %]
```

the plugin's provider will look for TTBook::Template::Plugin::Monitor, MyOrg::Template::Plugin::Monitor, Template::Plugin::Monitor, and Monitor; it will throw a plugin exception if none of those is found.

7.3.11 Template::Parser and Template::Grammar

Template::Parser and Template::Grammar are closely related. The parser starts things off by tokenizing the input template, and then refers to the grammar to determine whether the sequence of tokens gleaned from the template makes any sense. Template::Directive is used to generate the Perl code that represents the template.

Template::Parser is the ultimate recipient of all configuration parameters that affect the style of the template, such as TAG_STYLE, START_TAG, END_TAG, ANYCASE, INTERPOLATE, PRE_ and POST_CHOMP, V1DOLLAR, and GRAMMAR (see the Appendix for all the configuration options). The main methods of the parser are new and parse, as shown in Example 7-50.

Example 7-50. Creating and using parser and grammar objects

```
my $parser = Template::Parser->new({
  ANYCASE => 1,
  GRAMMAR => [% namespace %]::Template::Grammar->new( ),
});

my $data = $parser->parse($template_string);
my $doc = Template::Document->new($data);
```

$data is a reference to a hash, which is in the format expected by Template::Document.

In general, there isn't much reason to use Template::Parser or Template::Grammar directly. To get compiled versions of templates, use Template::Provider rather than Template::Parser—the version returned by the parser is in a raw, uncompiled form, used primarily for communication between the parser and the provider. Template::Grammar is generated using the *parser/Parser.yp* source file, which is processed by Parse::Yapp. It consists primarily of the rules and states used by the parser when determining whether the set of tokens created from the input template is valid. If you're interested in how this works, see Chapter 8.

7.3.12 Template::Directive

The Template::Directive module is a Perl factory—it exists only to return strings of valid Perl code, based on input from the parser. Template::Directive interacts closely with Template::Parser and Template::Grammar: the parser tokenizes the input, and the grammar determines which method to call on the factory class to produce the code that implements a directive.

The grammar also determines the arguments that get passed to the factory method, based on the type of directive. For example, an anonymous BLOCK definition, such as [% BLOCK %] Hello! [% END %], receives one argument, which is the contents of the block. (It is possible that this block contains other compiled directives, rather than just plain text, of course; this doesn't affect the generation of the code.) The factory code for anonymous blocks looks like this:

```
#------------------------------------------------------------------------
# anon_block($block)                    [% BLOCK %] ... [% END %]
#------------------------------------------------------------------------

sub anon_block {
    my ($class, $block) = @_;
    $block = pad($block, 2) if $PRETTY;

    return <<EOF;

# BLOCK
$OUTPUT do {
    my \$output  = '';
    my \$error;

    eval { BLOCK: {
$block
    } };
    if (\$@) {
```

```
        \$error = \$context->catch(\$@, \\\$output);
        die \$error unless \$error->type eq 'return';
    }

    \$output;
};
EOF
}
```

It's kind of ugly, primarily because the return value from the method is a string containing Perl, which will be compiled later.

The $block variable contains the results of calling other factory methods (e.g., ident, which handles [% GET foo %] directives). The pad function adds leading spaces to each line in $block if the $PRETTY variable (actually $Template::Directive::PRETTY) is set to a true value to indicate a human will read the generated code.

To control the code that gets written out for a given directive, subclass Template::Directive, and implement the appropriate method or methods. Many of these methods have names that are similar to the directives they implement, such as get, call, insert, and include, but many of the methods have unintuitive names. The easiest way to figure out which methods are called for each directive is to examine the grammar defined in *Parser.yp* (see Chapter 8).

You shouldn't need to touch most of the definitions in this module, but you will need to subclass it to implement any changes to the language you might want to make.

The best way to get a feel for how this module works is to set both $Template::Parser::DEBUG and $Template::Directive::PRETTY to 1, as noted earlier.

Template::Directive sports the following methods:

template($block)

An overall template wrapper.

anon_block($block)

An anonymous block.

block($block)

Any block of template directives.

textblock($text)

A block of text.

text($text)

A single piece of text.

quoted($items)

A quoted string.

ident($ident)

An identifier.

identref($ident)

A reference to an identifier.

assign($var, $val, $default)

An assignment.

args($args)

A list of arguments.

filenames($names)

A filename.

get($expr)

The GET directive.

call($expr)

The CALL directive.

set($setlist)

The SET directive.

default($setlist)

The DEFAULT directive.

insert($nameargs)

The INSERT directive.

include($nameargs)

The INCLUDE directive.

process($nameargs)

The PROCESS directive.

if($expr, $block, $else)

The IF directive.

foreach($target, $list, $args, $block)

The FOREACH directive.

next($nameargs, $block)

The NEXT directive.

wrapper($nameargs, $block)

The WRAPPER directive when specific with a single file.

multi_wrapper($file, $hash, $block)

The WRAPPER directive when specific with multiple files.

while($expr, $block)

The WHILE directive.

switch($expr, $case)

The SWITCH directive.

try($block, $catch)

The TRY directive.

throw($nameargs)

The THROW directive.

return( )

The RETURN directive.

stop( )

The STOP directive.

use($lnameargs)

The USE directive.

view($nameargs, $block, $defblocks)

The VIEW directive.

perl($block)

The PERL directive.

no_perl( )

The PERL directive when EVAL_PERL is disabled.

rawperl($block, $line)

The RAWPERL directive.

filter($lnameargs, $block)

The FILTER directive.

capture($name, $block)

Generates code to capture the output of a directive into a variable.

macro($ident, $block, $args)

The MACRO directive.

debug($nameargs)

The DEBUG directive.

7.3.13 Template::Document

This module defines an object class whose instances represent compiled template documents. The parser module creates a Template::Document instance to encapsulate a template as it is compiled into Perl code.

7.3.13.1 new

new expects a hashref containing BLOCK, DEFBLOCKS, and METADATA items. The BLOCK item should contain a reference to a Perl subroutine or a textual representation of Perl code, as generated by the Template::Parser module, which is then evaluated into a subroutine reference using eval. The DEFBLOCKS item should be a hashref containing further named BLOCKs, which may be defined in the template. The keys represent BLOCK names, and the values should be subroutine references or text strings of Perl code, such as the main BLOCK item. The METADATA item should be a hashref of metadata items relevant to the document.

Though Template::Document instances are usually created by the provider as it receives parsed data from the parser, it is possible to create standalone instances as well:

```
my $doc = Template::Document->new({
    BLOCK => sub { return "Hello!" },
    METADATA => { name => "greeting" },
    DEFBLOCKS => { }
});


print $doc->name( );
```

The only required parameter in the hashref is BLOCK:

```
my $timer = Template::Document->new({
    BLOCK => sub { time },
});
```

7.3.13.2 process

The process method can then be called on the instantiated Template::Document object, passing a reference to a Template::Content object as the first parameter. This will install any locally defined blocks (DEFBLOCKS) in the contexts BLOCKS cache (via a call to visit), so that they may be subsequently resolved by the context. The main BLOCK subroutine is then executed, passing the context reference on as a parameter. The text returned from the template subroutine is then returned by the process method, after calling the context leave method to permit cleanup and de-registration of named BLOCKs previously installed.

7.3.13.3 write_perl_file

The Template::Document module implements the methods necessary to write a compiled template to disk. These methods are as_perl and write_perl_file. If COMPILE_EXT and/or COMPILE_DIR are set, the provider calls write_perl_file, supplying it with a filename.

7.3.13.4 AUTOLOAD

Template::Document has an *AUTOLOAD* method that provides read-only access to the metadata defined for that template.

This includes all items defined in the template with *META*:

```
# thneed.tt2

[% META title = 'You need a thneed!'

       author = 'The Once-ler' %]



# Perl

my $doc = $context->template('thneed.tt2');

print $doc->author;
```

7.3.14 Template::Exception

The Template::Exception module defines an object class for representing exceptions within the template processing life cycle.

Exceptions can be thrown from Perl code in several different ways. The most straightforward way is to call die with a Template::Exception object as the argument. This will then be caught by any enclosing *TRY* blocks from where the code was called:

```
use Template::Exception;

...

die(Template::Exception->new('bad.things',

                             'Bad things happened.'));
```

This can be caught normally in the template:

```
[% USE Something %]

[% TRY %]

   ...

[% CATCH bad.things %]

   "Error: $error" ;

[% END %]
```

which will output:

```
Error: bad.things error - Bad things happened.
```

The info field can also be a reference to another object or data structure, if required:

```
die(Template::Exception->new('bad.things', {

       module => 'foo.pl',

       errors => [ 'bad permissions', 'naughty boy' ],

   }));
```

Later, in a template:

```
[% TRY %]

   ...

[% CATCH bad.things %]

   [% error.info.errors.size or 'no';

      error.info.errors.size =  = 1 ? ' error' : ' errors' %]

   in [% error.info.module %]:

      [% error.info.errors.join(', ') %].

[% END %]
```

it generates this output:

```
2 errors in foo.pl:

  bad permissions, naughty boy.
```

You can also call die with a single string, as is common in much existing Perl code. This will automatically be converted to an exception of the undef type (that's the literal string undef, not the undefined value). If the string isn't terminated with a newline, Perl will append the familiar *at $file line $line* message.

```
sub foo {

    # ... do something ...

    die("I'm sorry, Dave, I can't do that\n");

}
```

Within plugins, which are passed a reference to the context as the second argument, or some extension code that has the current Template::Context in scope, you can also raise an exception by calling the context throw method. You can pass it Template::Exception object reference, a pair of ($type, $info) parameters, or just an $info string to create an exception of undef type:

```
$context->throw($e);            # exception object

$context->throw('Denied');      # 'undef' type

$context->throw('bad.things', 'Bad things happened.');
```

7.3.15 Template::Iterator

The Template::Iterator module provides an easy way to create iterators. Iterator objects can be used within FOREACH loops, and they maintain the magic loop variable available in FOREACH loops.

To create a Template::Iterator instance, pass to the constructor a reference to an array:

```
use Template::Iterator;

my $iter = Template::Iterator->new(\@data);
```

Data is retrieved by calling get_first and then get_next until each item in the original list has been returned.

Iterator instances can be returned by methods designed to be called within *FOREACH* loops:

```
sub results {

    my $self = shift;

    my $iter = Template::Iterator->new($self->{ _RESULTS });



    return $iter;

}
```

From within a template, usage is as you would expect:

```
[% FOREACH result = search.results %]

    ...
```

Template::Iterator automatically provides the size, max, index, count, first, last, prev, and next methods, based on the result set used to initialize the instance. These methods correspond to the methods of the same names that can be called on loop within a FOREACH loop:

```
[% FOREACH result = search.results %]

    Size:   [% loop.size  # $iter->size( ) %]

    Max:    [% loop.max   # $iter->max( ) %]

    Index:  [% loop.index # $iter->index( ) %]

    Count:  [% loop.count # $iter->count( ) %]

    First:  [% loop.first # $iter->first( ) %]
```

```
    Last:   [% loop.last  # $iter->last( ) %]

    Prev:   [% loop.prev  # $iter->prev( ) %]

    Next:   [% loop.next  # $iter->next( ) %]

[% END %]
```

The astute reader will notice the similarity between loop and $iter; they are in fact the same Perl object.

A Template::Iterator instance can be created with a reference to an array of items, as noted earlier, or with an object that implements an as_list method. We can rewrite the preceding example to have as_list:

```
sub as_list {

    my $self = shift;

    return $self->{ _RESULTS };

}



sub results {

    my $self = shift;

    return Template::Iterator->new($self);

}
```

The constructor will also accept a reference to a hash array and will expand it into a list in which each entry is a hash array containing a key and value item, sorted according to the hash keys:

```
my $iter = Template::Iterator->new({

    foo => 'Foo Item',

    bar => 'Bar Item',

});
```

This is equivalent to:

```
my $iter = Template::Iterator->new([

    { key => 'bar', value => 'Bar Item' },

    { key => 'foo', value => 'Foo Item' },

]);
```

Chapter 7. Anatomy of the Template Toolkit

Now that we've spent a great deal of time looking at what you can do with the Template Toolkit, let's take a look inside and get a feel for how it actually works. We'll follow the flow of processing a template from the frontend (such as Template or *ttree*), to getting the file from disk (Template::Provider), to compiling it (Template::Parser, Template::Grammar, and Template::Directive), and to executing it (Template::Context and Template::Document).

We'll be using pseudocode versions of the methods to illustrate the major thrust of each component, mainly to gloss over tedious details of error checking, parameter handling, file opening and closing, and syntax. Feel free to get a copy of each *.pm* file and follow along with the real code; however, the best way to understand any complex system is to look at the innards, and the Template Toolkit is no exception.

8.1 Using and Implementing Noncore Components

Each Template Toolkit module knows about the other modules it needs to do its job, and will create instances of these objects unless one is passed to it explicitly. This means that modules are loaded and instances created on demand. The Template::Config module provides a convenient and centralized place to override core elements of the Template Toolkit, in the form of factory methods for each major component—context, filters, iterator, parser, plugins, provider, service, stash, and constants. The type of object that each method creates is, in turn, controlled by a series of variables in the $Template::Config namespace:

```
$CONTEXT   = 'Template::Context';
$FILTERS   = 'Template::Filters';
$ITERATOR  = 'Template::Iterator';
$PARSER    = 'Template::Parser';
$PLUGINS   = 'Template::Plugins';
$PROVIDER  = 'Template::Provider';
$SERVICE   = 'Template::Service';
$STASH     = 'Template::Stash';
$CONSTANTS = 'Template::Namespace::Constants';
```

These are given default values when the Template Toolkit is installed, and some of them might differ based on how the installation was performed. For example, the fast XS-based Stash (Template::Stash::XS) might have been installed instead of the default Stash.

The hash containing configuration parameters is passed around to each module's constructor. For example, Template::Service creates a Template::Context instance like so:

```
# In Service.pm
sub _init {
   my ($self, $config) = @_;

   # Some other configuration

   $context = $self->{ CONTEXT } = $config->{ CONTEXT }
       || Template::Config->context($config)
       || return $self->error(Template::Config->error);

   return $self;
}
```

In this case, if a Template::Context instance was part of $config, a new one would not be created. This feature is most useful for overriding settings, such as TOLERANT, for specific instances:

```
my $context = Template::Context->new(TOLERANT => 1);
my $tt = Template->new({
   CONTEXT => $context,
   TOLERANT => 0
});
```

To give a feel for implementing core module replacements, we'll illustrate a few simple ones. In most cases, the core modules can serve as a base class, and our subclasses need to override only a few methods.

All of the provider classes—Template::Provider, Template::Plugins, and Template::Filters—are stored as arrays, rather than as single items, specifically so that they can be supplemented by new modules. Simply create your new module and pass it around in the appropriate array when you create your Template object. The PREFIX_MAP gives the context hints as to which provider it should consult, based on the prefix, which looks very similar to the scheme of a URI:

```
[% PROCESS foo:bar/baz %]
```

The preceding code would invoke the provider mapped to foo to resolve the template *foo/bar*:

```
my $tt = Template->new({

    LOAD_TEMPLATES => [

        Template::Provider::Foo->new( ),

        Template::Provider->new( ),

    ],

    PREFIX_MAP => {

        foo     => 1,

        default => 0,

    },

});
```

8.1.1 A Provider That Can Fetch Files over HTTP

A relatively common question on the mailing list is, "Can I fetch templates via HTTP?" The official Template Toolkit FAQ[1] explains that, yes, you can, simply by using Template::Provider::HTTP. The problem, though, is that Template::Provider::HTTP does not exist.

[1] Find it at http://www.template-toolkit.org/faq.html.

Template::Provider already does most of what we want, including caching. Template::Provider::HTTP simply needs to add an LWP::UserAgent instance and customize the fetching process to use URIs rather than filesystem paths:

```
package Template::Provider::HTTP;



use strict;

use vars qw($VERSION);

use base qw(Template::Provider);



$VERSION = 1.00;



use File::Spec;

use HTTP::Request::Common qw(HEAD GET);

use LWP::UserAgent;

use Template::Constants qw(:status);

use Template::Provider;

use URI;

use URI::Escape qw(uri_escape);
```

In addition to Template::Provider and Template::Constants (for the STATUS constants), we need LWP::UserAgent, with which we will do the actual fetching, HTTP::Request::Common to create HTTP::Request objects (the GET and HEAD functions are very convenient shortcuts), and the URI, URI::Escape, and File::Spec modules to manipulate URIs and files.

When a Template::Provider::HTTP object is created, we need to also create an LWP::UserAgent instance.

Template::Provider::_init already handles the caching parameters, so we call it from our own _init:

```
sub _init {

    my ($self, $params) = @_;

    my ($ua, %lwp_args, $lwp_arg);



    $self->SUPER::_init($params);
```

Now we can do the LWP initialization. This list contains all the constructor options that LWP knows about, but for the sake of consistency with the Template Toolkit's native configuration methods, we require all uppercase option names:

```
for $lwp_arg (qw(agent from timeout use_eval parse_head

                max_size cookie_jar conn_cache protocols_allowed

                protocols_forbidden protocols_redirectable)) {

    my $uc_lwp_arg = uc $lwp_arg;

    $lwp_args{ $lwp_arg } = $params->{ $uc_lwp_arg }

        if defined $params->{ $uc_lwp_arg };

}



$self->{ USERAGENT } = $ua = LWP::UserAgent->new(%lwp_args);
```

A busy web site using this provider might want to put a caching proxy between the application server and the server providing the templates (even with the caching, we still need to HEAD the URI to see if it has changed). Setting up LWP's proxy support is simple:

```
if (my $proxy = $params->{ PROXY }) {

    $ua->proxy('http', $proxy);

}



if (my $no_proxy = $params->{ NO_PROXY }) {

    $no_proxy = [ $no_proxy ] unless ref($no_proxy) eq 'ARRAY';

    $ua->no_proxy(@$no_proxy);

}
```

The NO_PROXY option defines domains for which LWP should not use the proxy.

If we're debugging the provider, we can turn on debugging in LWP as well, using LWP::Debug:

```
if ($self->{ DEBUG }) {

    require LWP::Debug;

    LWP::Debug::level('+');

}
```

And, for good measure, we uniquely identify this agent, so it can be specifically picked out by the logs:

```
$ua->agent(sprintf "%s [%s/%.02f]",

    $ua->_agent, ref($self), $VERSION);
```

Because we do not have a base filename to use when contructing paths for compiled versions of the templates, we need to have COMPILE_DIR set if COMPILE_EXT is set (otherwise, the provider will try to create directories in /; we'll see this in more detail when we discuss _fetch).

```
# IF COMPILE_EXT is set, COMPILE_DIR must also be set

my ($cdir, $cext) = @$params{ qw( COMPILE_DIR COMPILE_EXT ) };

if (length($cext) && ! length($cdir)) {
```

```
    return $self->error("COMPILE_DIR must be set if COMPILE_EXT is set");
}

    return $self;
}
```

The main method of our provider, fetch, can be much simpler than the default fetch:

```
sub fetch {
    my ($self, $name) = @_;

    my $uri = URI->new($name, "http");
    $uri->scheme("http");
```

When the context determines which provider to use, based on the PREFIX_MAP, the prefix is stripped off. The URI module will help us put that back in. (The other methods in Template::Provider::HTTP that are expecting URIs will actually be expecting URI objects.)

```
$self->debug("Got request for '$uri'") if $self->{ DEBUG };

    return $self->_fetch($uri);
}
```

Just like Template::Provider, we defer the hard work to the _fetch method. In our case, this is mainly for consistency with the default provider, because fetch is so simple.

_fetch is a little more complicated—it has to be aware of the cache and needs to know how to request a new copy of the template if the one we have is out of date. The LWP::UserAgent module knows how to handle conditional requests, so we can take advantage of that here:

```
sub _fetch {
    my ($self, $uri) = @_;
    my ($data, $error, $compiled, $request, $response);
    my $ua = $self->{ USERAGENT };
    my $now = time;

    $self->debug("_fetch($uri)") if $self->{ DEBUG };
```

_compiled_filename determines what the filename would be if we were writing the Perl versions of the templates to the disk-based cache. There are two reasons we do this: we need to know where to look to see whether we already have a compiled version of the templates, and we need to know where to write compiled versions of the templates.

```
$compiled = $self->_compiled_filename($uri);
```

The HTTP equivalent of stat is to HEAD the URI and check for freshness headers, such as Expires or Last-Modified:

```
# HEAD the URI, to see if we need to refetch it all
$request = HEAD($uri);
$response = $ua->request($request);
```

Once we have the headers for the request, we can check whether it is newer than the compiled version (if we have one):

```
if ($compiled && -f $compiled && $response->is_fresh &&
    (stat($compiled))[9] <= $response->fresh_until) {
    # The compiled version is alright; return it;
```

```
    $data = $self->_load_compiled($compiled);
    $error = defined $data
        ? STATUS_OK
        : $self->{ TOLERANT }
            ? STATUS_DECLINED
            : STATUS_ERROR;
}
```

_load_compiled is a standard Template::Provider method that reads a compiled version of a template from disk, requires it, and returns a compiled subroutine.

If the template fails to load, we need to set $error appropriately. (The context will treat $data as the error message if $error is not undefined.) The TOLERANT flag is a signal from the user that these errors should not be immediately fatal, so we return STATUS_DECLINED if TOLERANT is set, and return STATUS_ERROR otherwise.

```
else {
    # The compiled version either doesn't exist or is out of date
    $request = GET($uri);
    $response = $ua->request($request);

    if ($response->is_success) {
        $data = {
            name => "$uri",
            text => $response->content,
            time => int($response->fresh_until),
            load => time,
        };
        $error = STATUS_OK;

        ($data, $error) = $self->_compile($data, $compiled);
        ($data, $error) = $self->store($compiled, $data);
        $data = $data->{ data }
            unless $error;
    }
    else {
        $data = $response->error_as_HTML( );
        $error = $self->{ TOLERANT } ? STATUS_DECLINED : STATUS_ERROR;

    }
}

    return ($data, $error);
  }
```

}

_compiled_filename is pretty straightforward, and again, we can take advantage of the superclass's version:

```
sub _compiled_filename {

    my ($self, $uri) = @_;



    # This adds '/' to the list of characters not encoded; we want those

    # so that we can make nested directories in which to store cache files.

    $uri = uri_escape($uri->opaque, "^A-Za-z0-9\-_.!~*'( )/");



    return File::Spec->canonpath($self->SUPER::_compiled_filename($uri));

}
```

This method turns an opaque (schemeless) URI such as //templates.tt2.org/config into a filename such as *//templates.tt2.org/config*. Template::Provider::_compiled_filename appends this to the value of COMPILE_DIR, so it ends up somewhere we can write (because you're not running this as the superuser, of course). Finally, File::Spec->canonpath *canonicalizes* the filename, which in this case means removing duplicate forward slash (/) characters. The / character had to be added to the list of characters not escaped by uri_escape, or we would have ended up with a filename such as *%2F%2Ftemplates.tt2.org%2Fconfig*, which is pretty ugly. With the slashes in place, we end up with a nested filesystem structure for our cache directory, which is easily navigable both by the curious developer and the provider as it walks the filesystem looking for compiled files. As a side effect, because we are not doing anything to prevent the escaping of the query string parameters, they become part of the compiled filename—invocations of the same URI but with different query strings will result in different cache files.

Using this new provider is easy:

```
my $http = Template::Provider::HTTP->new( );

my $prov = Template::Provider->new( );



my $tt = Template->new({

    LOAD_TEMPLATES => [

        $prov,

        $http,

    ],

    PREFIX_MAP => {

        http => 1,

        default => 0,

    }

});
```

As mentioned earlier, PREFIX_MAP is necessary to give the context a hint about which provider to use. We use the normal Template::Provider object by default, but for HTTP templates, use the HTTP provider:

```
[%

    PROCESS 'http://use.perl.org/journal.pl?uid=18&content_type=rss' |

        redirect('davorg.xml');

    USE davorg = XML.RSS('davorg.xml');

    FOREACH item IN davorg.items %]

    * [% item.title %]

        * [% item.link;
```

```
    END;
-%]
```

Example 8-1 is the complete Template::Provider::HTTP.

Example 8-1. Template::Provider::HTTP

```
package Template::Provider::HTTP;

use strict;
use vars qw($VERSION);
use base qw(Template::Provider);

$VERSION = 1.00;

use File::Spec;
use HTTP::Request::Common qw(HEAD GET);
use LWP::UserAgent;
use Template::Constants qw(:status);
use Template::Provider;
use URI;
use URI::Escape qw(uri_escape);

# ----------------------------------------------------------------------
# fetch($name)
#
# Retrieve the template identified by $name.  The PREFIX_MAP ensures
# that this gets called only when appropriate.
# ----------------------------------------------------------------------
sub fetch {
    my ($self, $name) = @_;

    # The Context's prefix handling strips out the 'http:', so we
    # need to add it back in.
    my $uri = URI->new($name, "http");
    $uri->scheme("http");

    $self->debug("Got request for '$uri'") if $self->{ DEBUG };

    return $self->_fetch($uri);
```

```
}

# -------------------------------------------------------------------
# fetch($name)
#
# Uses LWP::UserAgent to fetch a template referenced via http://...,
# and then uses standard Template::Provider methods to compile,
# cache, and so on.
# -------------------------------------------------------------------
sub _fetch {
    my ($self, $uri) = @_;
    my ($data, $error, $compiled, $request, $response);
    my $ua = $self->{ USERAGENT };

    $self->debug("_fetch($uri)") if $self->{ DEBUG };

    $compiled = $self->_compiled_filename($uri);

    # HEAD the URI, to see if we need to refetch it all
    $request = HEAD($uri);
    $response = $ua->request($request);

    if ($compiled && -f $compiled && $response->is_fresh &&
        (stat($compiled))[9] <= $response->fresh_until) {
        # The compiled version is alright; return it;

        $data = $self->_load_compiled($compiled);
        $error = defined $data
            ? STATUS_OK
            : $self->{ TOLERANT }
                ? STATUS_DECLINED
                : STATUS_ERROR;
    }
    else {
        # The compiled version either doesn't exist or is out of date
        $request = GET($uri);
        $response = $ua->request($request);

        if ($response->is_success) {
```

```
            $data = {
                name => "$uri",
                text => $response->content,
                time => int($response->fresh_until),
                load => time,
            };
            $error = STATUS_OK;

            ($data, $error) = $self->_compile($data, $compiled);
            ($data, $error) = $self->store($compiled, $data);
            $data = $data->{ data }
                unless $error;
        }
        else {
            $data = $response->error_as_HTML( );
            $error = $self->{ TOLERANT } ? STATUS_DECLINED : STATUS_ERROR;
        }
    }

    return ($data, $error);
}

# ---------------------------------------------------------------------
# _compiled_filename($uri)
#
# Transforms the URI into a filename.
# ---------------------------------------------------------------------
sub _compiled_filename {
    my ($self, $uri) = @_;

    # This adds '/' to the list of characters not encoded; we want those
    # so that we can make nested directories in which to store cache files.
    $uri = uri_escape($uri->opaque, "^A-Za-z0-9\-_.!~*'( )/");

    return File::Spec->canonpath($self->SUPER::_compiled_filename($uri));
}

# ---------------------------------------------------------------------
# _init(\%params)
```

```
#
# This is here primarily to initialize the LWP::UserAgent instance.
# --------------------------------------------------------------------
sub _init {
    my ($self, $params) = @_;
    my ($ua, %lwp_args, $lwp_arg);

    $self->SUPER::_init($params);

    for $lwp_arg (qw(agent from timeout use_eval parse_head
                     max_size cookie_jar conn_cache protocols_allowed
                     protocols_forbidden protocols_redirectable)) {
        my $uc_lwp_arg = uc $lwp_arg;
        $lwp_args{ $lwp_arg } = $params->{ $uc_lwp_arg }
            if defined $params->{ $uc_lwp_arg };
    }

    $self->{ USERAGENT } = $ua = LWP::UserAgent->new(%lwp_args);

    if (my $proxy = $params->{ PROXY }) {
        $ua->proxy('http', $proxy);
    }

    if (my $no_proxy = $params->{ NO_PROXY }) {
        $no_proxy = [ $no_proxy ] unless ref($no_proxy) eq 'ARRAY';
        $ua->no_proxy(@$no_proxy);
    }

    if ($self->{ DEBUG }) {
        require LWP::Debug;
        LWP::Debug::level('+');
    }

    $ua->agent(sprintf "%s [%s/%.02f]",
        $ua->agent, ref($self), $VERSION);

    # IF COMPILE_EXT is set, COMPILE_DIR must also be set
    my ($cdir, $cext) = @$params{ qw( COMPILE_DIR COMPILE_EXT ) };
    if (length($cext) && ! length($cdir)) {
```

```
        return $self->error("COMPILE_DIR must be set if COMPILE_EXT is set");
    }

    return $self;
}

1;
```

8.1.2 Restricting Access to Plugins

By default, all of the Template Toolkit's plugins are available to every template. Sometimes it makes sense to limit the available plugins, such as in a web-hosting or education situation. For these cases, restricting which plugins are available is useful.

Again, we can use the chain of responsibility to our advantage. By creating a Template::Plugins provider that governs access to plugins, we can ensure that only allowed plugins are loaded.

As you recall, the context interacts with the plugin providers by calling its fetch method, which is expected to return a plugin, or (undef, $error) if the plugin could not be loaded. Because the purpose of this plugin is to allow access only to specific plugins, it needs only to implement fetch, and doesn't have to do much more than simply decline to handle requests for allowed plugins by returning STATUS_DECLINED. If a plugin provider declines to handle a request, the context will move on the next provider in line or throw an exception if no more providers are available.

Here is the complete Template::Plugins::Allow:

```
package Template::Plugins::Allow;

use strict;
use Template::Constants qw(:status);

sub new {
    my $class = shift;
    bless { map { ($_, 1) } @_ }, $class;
}

sub fetch {
    my $self = shift;
    my $name = shift;
    return $self->{ $name }
        ? (undef,  STATUS_DECLINED)
        : ("access to $name not allowed", STATUS_ERROR);
}

1;
```

This provider is initialized with the names of the plugins that are allowed. We also need the regular plugins provider, to actually load the allowed plugins:

```
my $allow    = Template::Plugins::Allow->new(qw( Date Table ));
my $plugins  = Template::Plugins->new( );
```

Then we define the LOAD_PLUGINS chain of command with the Allow provider first:

```
my $tt = Template->new({

    LOAD_PLUGINS => [ $allow, $plugins ]

});
```

If the plugin is allowed, the Allow provider returns STATUS_DECLINED and control passes to the regular plugins provider. Otherwise, the Allow provider returns an error.

Here it is in use:

```
[% TRY;

     USE Date;

     "got date\n";

   CATCH;

     "not date: $error\n";

   END;


   TRY;

     USE Table([1, 2, 3]);

     "got table\n";

   CATCH;

     "not table: $error\n";

   END;


   TRY;

     USE Format;

     "got format";

   CATCH;

     "not format: $error\n";

   END;

%]
```

Here's the output:

```
got date

got table

not format: plugin error - access to Format not allowed
```

8.1.3 A chrooted Provider

By default, the Template Toolkit doesn't allow inclusion of files using absolute paths. This is to help disallow malicious or inexperienced users from including potentially sensitive files in output:

```
[% INSERT /etc/aliases %]
```

Sometimes, however, allowing absolute files does make sense. For example, you might want to specify the absolute path to a template to ensure that the INCLUDE_PATH doesn't supply you with a different template that happens to have the same name as the one you want. In these cases, it would be nice to be able to provide a limited directory structure for the templates to access. Normally, an entire process would be run in a *chrooted jail*, which means that the entire process (in this case, the Perl interpreter that is processing the templates via the Template Toolkit) would have a limited view of the underlying filesystem. (chroot is the name of the Unix system call that implements this functionality, and so has become synonymous with the activity.) This can be problematic, however; because everything that the Perl interpreter needs would need to be present in this limited filesystem, including system libraries, this means copying a lot of files around.

However, we can implement a Template::Provider subclass that has a limited view of the filesystem, by superficially emulating what chroot does: we can simply prepend a specific root (we'll call it CHROOT_BASE) to every absolute filename passed to INCLUDE, PROCESS, and INSERT. Then, a request such as:

```
[% INSERT /etc/aliases %]
```

would be translated into a request for */var/www/etc/aliases* (assuming a CHROOT_BASE of */var/www*).

We can build upon Template::Provider—we are modifying the default behavior only slightly. File::Spec::Functions provides a clean, function-oriented interface to File::Spec, while still preserving File::Spec's "cross-platform-y" goodness:

```
package Template::Provider::Chroot;


use strict;

use vars qw($VERSION);

use base qw(Template::Provider);


$VERSION = 1.00;


use File::Spec::Functions qw(canonpath catfile file_name_is_absolute);
```

We'll pull the CHROOT_BASE parameter out of the configuration, and then let Template::Provider::_init take over handling the rest of the parameters:

```
sub _init {

    my ($self, $params) = @_;

    $self->{ CHROOT_BASE } = $params->{ CHROOT_BASE } || "";


    return $self->SUPER::_init($params);

}
```

We need to override only the fetch method, and even then we need to do something only when the requested template is an absolute filename:

```
sub fetch {

    my ($self, $name) = @_;

    my $chroot = $self->{ CHROOT_BASE };

    my $newname = $name;


    if ($chroot && file_name_is_absolute($name)) {

        $newname = canonpath(catfile($chroot, $name));

        $self->debug("Using path of '$newname' instead of '$name'")

            if $self->{ DEBUG };

    }


    return $self->SUPER::fetch($newname);

}
```

One happy side effect of the method this provider uses is that if a template cannot be found, the error that the context emits references the original template name, not the adjusted filename.

Because this provider falls through to the behavior of the default provider, we don't need to use an array of providers or set up a PREFIX_MAP. We can simply tell Template::Config to use our new class instead of the default provider:

```
use Template;
use Template::Config;

$Template::Config::PROVIDER = 'Template::Provider::Chroot';
```

and continue as normal.

Example 8-2 shows the complete Template::Provider::Chroot.

Example 8-2. Template::Provider::Chroot

```
package Template::Provider::Chroot;

use strict;
use base qw(Template::Provider);

use File::Spec::Functions qw(canonpath catfile file_name_is_absolute);
use Template::Provider;

sub fetch {
    my ($self, $name) = @_;
    my $chroot = $self->{ CHROOT_BASE };
    my $newname = $name;

    if ($chroot && file_name_is_absolute($name)) {
        $newname = canonpath(catfile($chroot, $name));
        $self->debug("Using path of '$newname' instead of '$name'")
            if $self->{ DEBUG };
    }

    return $self->SUPER::fetch($newname);
}

sub _init {
    my ($self, $params) = @_;

    $self->{ CHROOT_BASE } = $params->{ CHROOT_BASE } || "";
```

```
    return $self->SUPER::_init($params);

}



1;
```

These few simple examples should be enough to get you started extending the Template Toolkit to do your bidding.

8.2 Creating Filters

Chapter 5 introduced Template Toolkit filters. This section explains how to write your own filters.

There are two types of filters: *static* and *dynamic*. A static filter is one that always operates the same way, and a dynamic filter is one that can be configured differently for each invocation. From within templates, they are invoked almost identically, with the exception that dynamic filters can take arguments, while static filters cannot.

8.2.1 Static Filters

Internally, filters are implemented as references to subroutines; when invoked, these subroutines are passed the text to be filtered as a string, and are expected to return a string. Defining a static filter is as simple as creating a subroutine and declaring it in the FILTERS configuration option (it can also be installed into the context with the define_filter method). All invocations of a static filter will use the same subroutine reference, which won't be passed any parameters other than the text to be filtered. Standard filters such as html and lower are examples of static filters.

Here is a simple Perl subroutine, designed to be used as a static filter, which rot13s text:[2]

> [2] rot13 is a simple, well-known substitution cipher, in which each character in a string of text is replaced by the character 13 positions away. For example, a becomes n, b becomes o, and so on. Passing a string through rot13 two times restores the original string.

```
sub rot13 {

    my $text = shift;

    $text =~ tr/a-zA-Z/n-za-mN-ZA-M/;

    return $text;

}
```

Once our rot13 subroutine has been defined, it can be installed in the processing context by passing a subroutine reference to the Template constructor:

```
my $tt = Template->new({

    FILTERS => {

        'rot13' => \&rot13,

    },

});
```

Using our rot13 filter is easy:

```
[% FILTER rot13 %]

Gur juvgr mbar vf sbe ybnqvat naq haybnqvat bayl.

[% END %]
```

The preceding code produces, naturally:

```
The white zone is for loading and unloading only.
```

And that's most of what there is to static filters: define a subroutine that expects one text argument, munges that argument in some way, and returns the output. The processing can be arbitrarily complex, and of course the text returned can be anything at all, or even nothing.

8.2.2 Dynamic Filters

The FILTER directive is expecting a reference to a subroutine that will be invoked with its text. For static filters, this subroutine reference was installed by the FILTERS or LOAD_FILTERS options when the Template instance was created. However, because the parameters of a dynamic filter might not be known until runtime, they must be treated differently. Dynamic filters are installed differently than static filters (via the FILTERS call), and the context knows to invoke them differently. Installing a dynamic filter at constructor time looks like this:

```
my $tt = Template->new({
    FILTERS => {
        'rot13'  => \&rot13, # our trusty static filter
        'censor' => [ \&censor_factory, 1 ], # our dynamic filter
    },
});
```

As you can see, dynamic filters are installed as array references, where the first element is a code reference and the second is a flag: 1 for dynamic, 0 for static. Analogously, static filters can be installed as:

```
FILTERS => {
    'rot13' => [ \&rot13, 0 ],
},
```

which explicitly marks it as a static filter.

When a dynamic filter is fetched, it is expected to return a reference to a subroutine, which is what the FILTER directive is expecting. The subroutine that is called and expected to return another subroutine to FILTER is called a *factory*.

Let's look at censor_factory, referred to earlier.

```
sub censor_factory {
    my ($context, $letter) = @_;

    return sub {
        my $text = shift;
        $text =~ s/($letter)/"*" x length($1)/eg;
        return $text;
    }
}
```

When called as:

```
[% text FILTER censor("a") %]
```

each a in $text will be replaced with *. When called as:

```
[% text FILTER censor("lemon") %]
```

each lemon in $text will be replaced with *****, and so on. Note that the arguments to censor—a and lemon—need to be given to censor_factory, which uses them to create a closure. This closure is then passed to FILTER, which invokes the subroutine and then discards it. If the dynamic filter is going to be reused, with the same arguments, it can be assigned to a variable:

```
[% text | no_lemons = censor("lemon") %]

[% more_text | no_lemons %]
```

The second invocation of no_lemons behaves identically to the first.

censor_factory is invoked with the Template::Context object as its first argument, and any other arguments as the rest of @_. Named parameters are folded into a hash reference and passed as the last argument, as is usual for invoked subroutines within templates. The factory subroutine should take into account the number and type of arguments it is expecting. Filters are free to ignore any or all of these arguments, of course.

We can redefine censor_factory to accept configuration parameters this way:

```
sub censor_factory {
    my ($context, @args) = @_;
    my $args = ref($args[-1]) eq 'HASH' ? pop @args : { };
```

```
    my $repl = $args->{'replacement'} || "*";


    return sub {
        my ($text, $letter) = @_;
        $text =~ s/($letter)/$repl x length($1)/eg;
        return $text;
    }
}
```

The key is @args: if there are any named parameters, they will be collected and passed, as a reference to a hash, as the last element of @_. These are popped off @args and assigned to hash references $args, from which we extract the replacement key (or a default of *, to make it backward compatible with our earlier version of censor_factory).

Now, we can call censor with a configurable replacement character:

```
[% text | censor("lemon", replacement = "#") %]
```

And each occurrence of the string lemon will be replaced with #####. Because the Template Toolkit rearranges named parameters to be passed last, our filter can be called with replacement replacement anywhere in the argument list, with identical results:

```
[% text | censor(replacement = "#", "lemon") %]
```

It it possible to pass arguments to static filters, but they are ignored:

```
[% FILTER rot13(all_caps = 1) %]
Gur juvgr mbar vf sbe ybnqvat naq haybnqvat bayl.
[% END %]


The white zone is for loading and unloading only.
```

The Template Toolkit ignores parameters passed to items that are not expecting them: because the presentation language is implementation neutral, a template has no way of knowing whether this filter can take arguments.

8.2.3 Template::Plugin::Filter

The Template::Plugin::Filter module, which allows for filters to be written and treated as plugins, is a bit of an odd beast—it is actually a plugin, but is designed to be used as a filter:

```
[% USE myfilt = MyFilter %]
[% FILTER $myfilt %]
    ...
[% END %]
```

Using Template::Plugin::Filter to write filters is more akin to writing plugins than to writing filters, with one major difference: when the variable is used as a filter, a method named filter is invoked. All of our filter examples can be turned into Template::Plugin::Filter objects by renaming the subroutine to filter and putting it into its own class, which inherits from Template::Plugin::Filter:

```
package TTBook::Template::Plugin::Rot13;


use strict;
use base qw(Template::Plugin::Filter);


sub filter {
    my $text = shift;
```

```
    $text =~ tr/a-zA-Z/n-za-mN-ZA-M/;

    return $text;

}
```

Now our rot13 filter can be used like so:

```
[% USE encryptor = Rot13 %]

[% text | $encryptor %]
```

Note that you must explicitly dereference the plugin filter using the $encryptor format; this is key!

8.2.4 Writing New Filters

As we have seen, a filter is a subroutine reference that can be invoked from within the processing context. There are many mature and full-featured modules on CPAN that filter text. Often, you will need the functionality of one of these modules within your templates, and filters are the easiest way to glue the two together. We cover some of these modules next.

8.2.4.1 Digest::MD5

The Digest::MD5 module creates a *message digest* of text or files. According to the manpage:

> *The "Digest::MD5" module allows you to use the RSA Data Security Inc. MD5 Message Digest algorithm from within Perl programs. The algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input.*

This makes a good candidate for a filter. We could use the MD5 filter from within *ttree* to generate our checksum files:

```
[% USE dir = Directory(".");

   FOREACH file = dir.files;

    checksum = INSERT $file.name | md5 %]

    * [% file.name %] = [% checksum %]

[% END %]
```

Digest::MD5 exports a function called md5_hex that does exactly what we are looking for. Our md5 static filter is simple:

```
use Digest::MD5 qw(md5_hex);

sub md5 {

    my $text = shift;

    return md5_hex($text);

}
```

This static filter is so simple that it is possible to inline it with almost no loss of clarity:

```
use Digest::MD5 qw(md5_hex);


my $tt = Template->new(

    FILTERS => {

        "md5" => sub { my $text = shift; return md5_hex($text); },

    },

);
```

8.2.4.2 Text::Bastardize

Text::Bastardize is a great little module for manipulating text. It has methods for transformations to pig Latin, numerical abbreviation, and k3wlt0k, among others.

Using Text::Bastardize is simple:

```
use Text::Bastardize;

my $bastard = Text::Bastardize->new;
$tb->charge($data);

print $tb->rev;
```

The various methods return arrays, which in general is appropriate when dealing with text, but we'll need strings; join is our friend:

```
print join "", $tb->rev;
```

The methods Text::Bastardize makes available include the following:

rdct

"Reduce" text:

```
$tb->charge("The white zone is for loading and unloading only.");
$tb->rdct( );
# the whte z1 is fr ladng nd unladng only.
```

pig

Transform text into pig Latin:

```
$tb->charge("You need a thneed!");
$tb->pig( );
# youay eednay away eedthnay!
```

rot13

Hey, this looks familiar:

```
$tb->charge("with or without is the different.");
$tb->rot13( );
# jvgu be jvgubhg vf gur qvssrerag
```

k3wlt0k

Transforms your text into its "elite" form:

```
$tb->charge("You'll love it, it's a way of life");
$tb->k3wlt0k( );
# JUR11 10V4 17, 17Z 3 W3Y 0F 11F4
```

rev

Reverses your text:

```
$tb->charge("A thing of beauty is a joy forever.")
$tb->rev( );
# .reverof yoj a si ytuaeb fo gniht A
```

n20e

Replaces long words (more than six characters) with numeric equivalents:

```
$tb->charge("Every nonzero finite dimensional inner " .
            "product space has an orthonormal basis."
$tb->n20e( );
# Every n5o finite d9l inner p5t space has an o9l basis.
```

Turning these Text::Bastardize methods into filters is relatively straightforward:

```
use Template;

my $tt = Template->new(
  FILTERS => {
    "rdct" => \&rdct,
    "n20e" => \&n20e,
  },
);

sub rdct {
  my $text = shift;

  my $tb = Text::Bastardize->new;
  $tb->charge($text);

  return join "", $tb->rdct;
}

sub n20e {
  my $text = shift;

  my $tb = Text::Bastardize->new;
  $tb->charge($text);

  return join "", $tb->rdct;
}
```

And so on. Each Text::Bastardize method follows the same general pattern:

```
my $tb = Text::Bastardize->new;
$tb->charge($data);

return join "", $tb->METHOD;
```

This means that we can produce these subroutines automatically, with a factory function:

```
sub bastardize_factory {
    my $type = shift || "rot13";

    return sub {
        my $text = shift;

        my $tb = Text::Bastardize->new;
        $tb->charge($text);

        return join "", $tb->$type( );
    };
}

my $tt = Template->new(
    FILTERS => {
        "rdct" => bastardize_factory("rdct"),
        "n20e" => bastardize_factory("n20e"),
    },
);
```

This is exactly what is needed to create dynamic filters; we can make bastardize available to our templates as a dynamic filter:

```
my $tt = Template->new(
    FILTERS => {
        "bastardize" => [ \&bastardize_factory, 1 ]
    },
);
```

The bastardize dynamic filter would be used with an argument:

```
[% FILTER bastardize("n20e") %]
Numeric abbreviation.
[% END %]
```

The filter subroutine created by calling bastardize(TYPE) can be captured for later use, by assigning it to a variable:

```
[% FILTER rot13 = bastardize("rot13") %]

Grzcyngr Gbbyxvg Ehyrf

[% END %]


[% text | rot13 %]
```

As you will recall, dynamic filters get called with a Template::Context instance as their first argument. bastardize_factory needs to deal with this:

```
sub bastardize_factory {

    shift( ) if ref $_[0];
```

If the first argument is a reference, it is not the type that we are expecting; therefore, we can shift it away. bastardize_factory, in its entirety, is pretty simple:

```
sub bastardize_factory {

    shift if ref $_[0];

    my $type = shift;

    my $tb = Text::Bastardize->new;



    return sub {

        my $text = shift;



        $tb->charge($text);

        return join "", $tb->$type;

    };

}
```

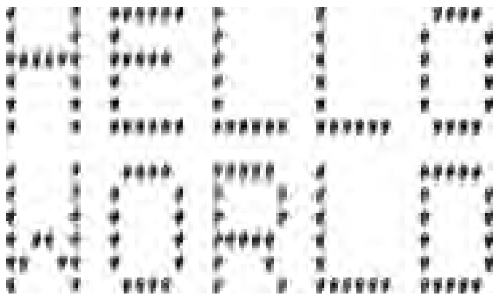
And, of course, we can have both the static and dynamic versions of our bastardize filters in our Template::Filters instance:

```
my $tt = Template->new(

    FILTERS => {

        rdct       => [   bastardize_factory("rdct"),    0 ],

        pig        => [   bastardize_factory("pig"),     0 ],

        k3wlt0k    => [   bastardize_factory("k3wlt0k"), 0 ],

        rot13      => [   bastardize_factory("rot13"),   0 ],

        rev        => [   bastardize_factory("rev"),     0 ],

        n20e       => [   bastardize_factory("n20e"),    0 ],

        bastardize => [ \&bastardize_factory,            1 ],

    },

);
```

8.2.4.3 Text::FIGlet

FIGlet is a program for making large letters out of ordinary, unexpecting text, and Text::FIGlet (http://www.figlet.org/) is a Perl implementation. *FIGlet* is akin to the Unix program banner, which formats a message for printing on a line printer (see Figure 8-1).

Figure 8-1. "Hello world" created by the Unix program banner



FIGlet does something similar, but adds font capability kerning, and the ability to make your text face in the correct direction. The default font looks like Figure 8-2.

Figure 8-2. "Hello world" created by FIGlet (using the default font)

But there are hundreds of other fonts, such as *rozzo* (see Figure 8-3).

Figure 8-3. The rozzo font in FIGlet

The possibilities here are staggering, of course.

Using Text::FIGlet is easy:

```
use Text::FIGlet;

my $figgy = Text::FIGlet->new(-f => $fontname);



print $figgy->figify(-A => $text);
```

Turning this into a dynamic filter is straighforward: we need to handle the various -X constructor parameters, one of which is a scalar containing the text to be figified. Hey, we have one of those:

```
sub figify_filter_factory {

    my ($context, @args) = @_;

    my $args = ref($args[-1]) eq 'HASH' ? pop @args : { };

    my $figgy = Text::FIGlet->new(%$args);



    return sub {

        my $text = shift;

        $figgy->figify(-A => $text);
```

```
    }
}
```

Using this figify filter feels a little unnatural, however, mainly due to the strange-looking format of the constructor parameters:

```
[% FILTER figify("-f" => "acrobatic") %]
Hello, world!
[% END %]
```

We can provide intuitive mappings for these in our implementation:

```
# some nice aliases...
my %fig_params = (
    "german"        => "-D",
    "fontdir"       => "-d",
    "fontfile"      => "-f",
    "smushmode"     => "-m",
    "direction"     => "-X",
    "justification" => "-x",
    "width"         => "-w",
);
# ...and some even nicer aliases
$fig_params{'font'} = $fig_params{'fontfile'};
$fig_params{'dir'} = $fig_params{'fontdir'};

sub figify_filter_factory {
    my ($context, @args) = @_;
    my $args = ref($args[-1]) eq 'HASH' ? pop @args : { };
    my %cons_args;

    for my $a (%$args) {
        my $p = $fig_params{ $a };
        $cons_args{ $p } = $args->{ $a } if defined $p;
    }

    my $figgy = Text::FIGlet->new(%cons_args);

    return sub {
        my $text = shift;
        $figgy->figify(-A => $text);
    }
}
```

Now our figified templates look a little more like other templates:

```
[% FILTER figify(font => "cosmic") %]

Hello, world!

[% END %]
```

The output is shown in Figure 8-4.

Figure 8-4. "Hello world" using a dynamic filter in FIGlet

8.2.4.4 Normalizing HTML: HTML::Clean

The HTML::Clean module encapsulates a number of common techniques for minimizing the size of HTML output: removing unnecessary whitespace, comments, and META tags; replacing longer tags with shorter ones; and removing empty unnecessary tags. HTML::Clean normally operates in filter mode, which makes it an ideal filter. HTML::Clean is available from http://search.cpan.org/dist/HTML-Clean/.

The "clean level" and types of cleaning that HTML::Clean does are controlled by options passed to strip, so HTML::Clean is a good candidate for a dynamic filter:

```
use HTML::Clean;


sub clean {

    my ($context, @args) = @_;

    my $config = ref($args[-1]) eq 'HASH' ? pop @args : { };


    return sub {

        my $text = shift;


        my $h = HTML::Clean->new(\$text);


        $h->level($config->{'level'})

            if (defined $config->{'level'});


        $h->strip($config);


        return ${ $h->data };

    };

}


my $tt = Template->new(FILTERS => { clean => [ \&clean, 1 ] });
```

This makes a good overall filter:

```
[% BLOCK page %]
[% FILTER clean(level = 9) %]
<html>
  <head>
    <title>[% template.title %]
  </head>
  <body>
[% content %]
  </body>
</html>
[% END %]
[% END %]

[% WRAPPER page %]
   ...
```

Using Subroutine References as Filters

Because filters are "just" subroutine references, and the Template Toolkit allows for subroutine references to be passed as values in the second parameter to process, you might be thinking that we should be able to rephrase our filter examples as:

```
my %filters = (
    'rot13' => \&rot13,
    'censor' => \&censor_factory,
);
my $t = Template->new( );
$t->process($file, \%filters);
```

The answer, of course, is, yes, there's more than one way to do it. However, this method requires that your filters be called as:

```
[% rot13(text) %]
[% censor(text) %]
```

Because "real" filters can be called using the FILTER or | notation, you lose the ability to pipe PROCESS and INCLUDE calls through your subroutine:

```
[% rot13(INCLUDE encrypted.txt) %]
```

Therefore, the previous code produces a parser error. Using an intermediate variable is an option, of course:

```
[% enc = INCLUDE encrypted.txt; rot13(enc); %]
```

But that's no fun.

These examples, by the way, produce something like:

```
Gur juvgr mbar vf sbe ybnqvat naq haybnqvat bayl.
CODE(0x83a85c4)
```

which, in the second case, is not what we wanted. Dynamic filter factories, which return subroutine references, need to be handled differently:

```
$filters{'censor_a'} = censor_factory("a");

$filters{'censor_b'} = censor_factory("b");
```

And so on, which has obvious ramifications in the template. In these cases, dynamic filters have to be rewritten to return text, and not a code reference:

```
sub censor {

    my ($text, $letter) = @_;

    $text =~ s/($letter)/"*" x length($1)/eg;

    return $text;

}
```

8.3 Creating Plugins

As we saw in Chapter 6, a plugin is implemented as an object-oriented Perl module. This module must implement a few basic methods in order for the context to load it correctly, and all of these methods can be inherited from the Template::Plugin module; otherwise, a plugin can be very free form.

8.3.1 The Template::Plugin Module

The Template::Plugin module both defines the plugin API and serves as a base class for plugin implementations. By default, a Template::Plugin instance has almost no functionality, other than to load correctly.

Template::Plugin defines three methods: load, new, and error. Subclasses are free to override any of these methods, or implement any others they might need to perform their duties.

load($context)

This method is called by Template Toolkit when the plugin module is first loaded. It is called as a package method and thus implicitly receives the package name as the first parameter. A reference to the Template::Context object loading the plugin is also passed. The default behavior for the load method is to simply return the class name; the calling context then uses this class name to call the new package method:

```
package MyPlugin;

sub load {              # called as MyPlugin->load($context)

    my ($class, $context) = @_;

    return $class;      # returns 'MyPlugin'

}
```

new($context, @params)

This method is called to instantiate a new plugin object for the USE directive. It is called as a package method against the class name returned by load. A reference to the Template::Context object creating the plugin is passed, along with any additional parameters specified in the USE directive:

```
sub new {               # called as MyPlugin->new($context)

    my ($class, $context, @params) = @_;

    bless {

        _CONTEXT => $context,

        _PARAMS => \@params,

    }, $class;          # returns blessed MyPlugin object

}
```

error($error)

This method, inherited from the Template::Base module, is used for reporting and returning errors. It can be called as a package method to set/return the $ERROR package variable, or as an object method to set/return the object's _ERROR member. When called with an argument, it sets the relevant variable and returns undef. When called without an argument, it returns the value of the variable.

```
sub new {

    my ($class, $context, $dsn) = @_;

    return $class->error('No data source specified')

        unless $dsn;

    bless {
```

```
            _DSN => $dsn,
        }, $class;
    }
    ...
    my $something = MyModule->new( )
        || die MyModule->error( ), "\n";
    $something->do_something( )
        || die $something->error( ), "\n";
```

The Template::Context object that handles the loading and use of plugins calls the new and error methods against the package name returned by the load method. In pseudocode terms, it might look something like this:

```
$class  = MyPlugin->load($context);       # returns 'MyPlugin'


$object = $class->new($context, @params)  # MyPlugin->new(...)
    || die $class->error( );              # MyPlugin->error( )
```

The load method may alternately return a blessed reference to an object instance. In this case, new and error are then called as *object* methods against that prototype instance.

Example 8-3 is the complete TTBook::Template::Plugin::Printer plugin, which implements a print service.

Example 8-3. TTTBook::Template::Plugin::Printer

```
package TTBook::Template::Plugin::Printer;


use strict;
use vars qw($PRINTER $SERVER);
use base qw(Template::Plugin);


use Template::Plugin;
use Template::Exception;
use Net::Printer;


$PRINTER = "jeckyl";
$SERVER = "mr-hyde";


sub load {
    my ($class, $context) = @_;
    my $printer = Net::Printer->new(printer => $PRINTER,
                                    server  => $SERVER);
    my $self = bless {
        _CONTEXT => $context,
        _PRINTER => $printer,
```

```
    }, $class;

    return $self;
}

sub new {
    my ($self, $context) = @_;
    return $self;
}

sub print {
    my ($self, $data) = @_;
    my ($printer, $context) = @$self{ qw( _PRINTER _CONTEXT) };

    my $result = $printer->printstring($data);
    $context->throw('printer', $result)
        unless (int($result) =  = 1);

    return "";
}

1;
```

In this example, we implemented a *Singleton* plugin. One object gets created when load is called; the object simply returns itself for each call to new.

When the plugin is loaded, a TTBook::Template::Plugin::Printer instance is created; each call to new is called against this object, which instantiates and returns that same instance.

Because calls to print throw printer exceptions if there is a problem, they should be wrapped in TRY / CATCH blocks:

```
[% USE Printer %]
[% TRY %]
    [% Printer.print(data) %]
[% CATCH printer %]
    There was an error printing: [% error %]
[% END %]
```

print explicitly returns an empty string so that there is no unwanted output in the template.

8.3.2 Installing Functions into the Stash from Within a Plugin

While plugins are implemented as object-oriented modules, there is no reason that every plugin has to be used in an object-oriented way. Because a plugin is invoked with $context as an argument, a plugin writer can elect to install functions in the stash in addition to returning an object designed to be used:

```
package TTBook::Template::Plugin::Red;

use strict;
use base qw(Template::Plugin);

sub new {
    my ($class, $context) = @_;
    my $stash = $context->stash;

    $stash->set('red', \&make_red);

    return sub { make_red(@_) };
}

sub make_red {
    my $text = shift;
    return qq|<font color="red">$text</font>|;
}

1;
```

The plugin still needs to return a blessed object, but it will probably be ignored. This plugin would be used like this:

```
[% USE Red %]

Hello, [% red('World') %]
```

However, because we've chosen to return a subroutine reference, the plugin name can also be used, to the same effect:

```
[% USE colorizer = Red %]

Hello, [% red('red world!') %]
I am [% colorizer('also red') %].
```

This example, while silly, illustrates two important points. First, once a plugin has a reference to the stash, arbitrary functionality can be added to your templates. Second, a plugin need merely return something that Perl considers true—it doesn't have to be a blessed object.

Instead of make_red, we could have created an incrementing counter:

```
my $count = 0;
$stash->set('counter' => sub { ++$count });
```

Each time counter is invoked, it returns the next number:

```
[% FOREACH [ 1 .. 10 ] %]
    * [% counter %]
[% END %]
```

As such, the previous code returns:

```
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
```

By making new( ) accept an argument, we can seed the counter:

```
sub new {
    my ($class, $context, $start) = @_;
    my $stash = $context->stash;

    my $count = int($start || 0);
    $stash->set('counter' => sub { ++$count });

    bless { } => $class;
}
```

This counter will start where we tell it to:

```
[% USE Counter(100) %]
[% counter %]
```

As such the previous code yields:

```
101
```

Example 8-4 is the complete TTBook::Template::Plugin::Counter.

Example 8-4. TTBook::Template::Plugin::Counter

```
package TTBook::Template::Plugin::Counter;

use strict;
use vars qw($VERSION);
use base qw(Template::Plugin);

sub new {
    my ($class, $context, $start) = @_;
    my $stash = $context->stash;

    my $count = int $start;
```

```
    $stash->set("counter" => sub { ++$count });


    bless { } => $class;
}


1;
```

8.3.3 Defining Filters from Within a Plugin

Earlier, we saw how the define_filter( ) method can be called against the $context object to define new filters. Let's look at a plugin that does this.

Let's revisit our Digest::MD5 filter and install it from within a plugin. Recall that the body of the filter was a very simple subroutine:

```
use Digest::MD5 qw(md5_hex);

sub md5 {

    my $text = shift;

    return md5_hex($text);

}
```

Installing a plugin into the current stash is something that should be done when the module is loaded, so load is an ideal place for it:

```
sub load {

    my ($class, $context) = @_;

    $context->define_filter('md5', \&md5);

    return $class;

}
```

Example 8-5 is the complete $namespace::Template::Plugin::MD5.

Example 8-5. $namespace::Template::Plugin::MD5

```
package TTBook::Template::Plugin::MD5;


use strict;

use vars qw($VERSION);

use base qw(Template::Plugin);


use Template::Plugin;

use Digest::MD5 qw(md5_hex);


$VERSION = 1.01;


sub md5 {

    my $text = shift;
```

```
    return md5_hex($text);
}

sub load {
    my ($class, $context) = @_;
    $context->define_filter("md5", \&md5);
    return $class;
}

1;
```

The Printer plugin shown earlier is another good example of a plugin that could also work as a filter:

```
[% USE Printer %]
[% text | print %]
```

Modifying load to do what we intend is simple:

```
sub load {
    my ($class, $context) = @_;
    my $printer = Net::Printer->new(printer => $PRINTER,
                                    server  => $SERVER);
    my $self = bless {
        _CONTEXT => $context,
        _PRINTER => $printer,
    }, $class;

    $context->define_filter('print', sub { $self->print(@_) });

    return $self;
}
```

We need to pass a closure to define_filter because print needs access to $self (the plugin object) when it is invoked, unlike the MD5 filter, where md5 was simple enough to stand on its own.

8.3.4 Defining New Virtual Methods from Within a Plugin

Virtual methods are defined within Template::Stash, and are implemented as subroutine references attached to package-scoped hashes within the Template::Stash namespace: $Template::Stash::SCALAR_OPS for scalar vmethods, $Template::Stash::LIST_OPS for list vmethods, and $Template::Stash::HASH_OPS for hash vmethods. Creating a new vmethod is as simple as assigning a subroutine reference to the appropriate package variable.

To get a feel for creating vmethods, let's add a few. Graham Barr's *List::Util* package (shipped with Perl as of 5.8.0, available from http://search.cpan.org/dist/List-Util/ for versions before 5.8.0) provides several very useful functions that operate on arrays, such as shuffle, which will randomize an array, and max, which will return the largest numeric value in an array:

```
use Template::Stash;
use List::Util;


my $l_ops = $Template::Stash::LIST_OPS;


$l_ops->{'shuffle'} = \&List::Util::shuffle;
$l_ops->{'max'} = \&List::Util::max;
```

These new virtual methods can now be used like any other virtual methods:

```
[% list = [ 1 2 3 4 5 ];
  shuflist = list.shuffle;
%]
```

Note that because of how virtual methods are implemented, once a subroutine is installed as a vmethod, it is global, and available to all templates.

8.3.5 Writing New Plugins

To help you get a feel for the real-world issues that crop up when you build plugins, let's look closely at three sample plugins, building from a simple wrapper to one that searches Google.

8.3.5.1 A simple wrapper plugin

One of the simplest types of plugins is one that acts as a factory for another object-oriented module, such as CGI or Apache. In a case such as this, the entire plugin can be implemented by having the plugin's new( ) method defer to the modules constructor. A good example is the standard CGI plugin, the entirety of which is Example 8-6.

Example 8-6. Standard CGI plugin

```
package Template::Plugin::CGI;


use strict;
use base qw( Template::Plugin );
use Template::Plugin;
use CGI;


sub new {
    my $class   = shift;
    my $context = shift;
    CGI->new(@_);
}


1;


_ _END_ _
```

Most of the time, however, plugins require a little more work. Under mod_perl, the Apache module provides a way to directly access the current requested object and manipulate the request. An Apache plugin, to be used in a template running under mod_perl, might look like Example 8-7.

Example 8-7. Apache plugin

```
package TTBook::Plugin::Apache;

use strict;
use vars qw($VERSION);

$VERSION = 1.00;

use Apache;
use base qw(Template::Plugin);

sub new {
    return Apache->request;
}
```

In the case of the Apache class, the constructor is named request( ), which returns a reference to the current Apache request object. This plugin would be used like this:

```
[% USE r = Apache %]

<p>Query parameters are: '[% r.args %]'.</p>
<p>You are using [% r.header_in('User-Agent') %].</p>
```

Of course, most plugins are not this simple, including this one. Because this module delegates to a regular Apache instance, we can still call standard Apache methods against it, including the print method, which can have unpredictable results when invoked within a template. Because we're dealing with a plugin, and plugins are basically regular Perl modules, we can inherit from the Apache module, implement a Template Toolkit-friendly version of the print method, and return a reference to our subclass. The Apache module makes special allowances for subclasses: an object that is not an Apache instance is checked to see whether it is a hash, and whether it contains an Apache instance or subclass as a data member named _r. Using this information, we can rewrite our plugin to be a little more interesting. The rewritten plugin is shown in Example 8-8.

Example 8-8. Rewritten Apache plugin

```
package TTBook::Template::Plugin::Apache;

use Apache;
use base qw( Template::Plugin Apache );
use vars qw($VERSION);

$VERSION = 1.01;

sub new {
    my ($class, $context) = @_;
```

```
    bless {
        '_r' => Apache->request,
    } => $class;
}

sub print {
    my ($self, @data) = @_;
    my ($str, $output);

    for $str (@data) {
        if (ref $str eq 'SCALAR') {
            $output .= $$str;
        } else {
            $output .= $str;
        }
    }

    return $output;
}
```

We've added a print method that accumulates output and returns it to the context. (Apache's print method allows scalar references to be passed, for efficiency; our method defeats this efficiency at the cost of working correctly.) Now, calls to the instance's print( ) method Do the Right Thing:

```
[% r.print('foo') %]
```

The preceding code is the same as:

```
[% foo %]
```

which isn't all that useful, in and of itself, except that it prevents unforeseen errors.

Something similar has to be done with the send_http_header( ) method, but in this case, we can discard the call, assuming that something else will be sending the headers. Apache's send_http_header( ) takes an optional $content_type, which is used to set the Content-Type header (this is generally optional, as the TypeHandler usually has already set the content type). Our send_http_header( ) can call the content_type( ) method to set the content type if one is provided:

```
sub send_http_header {
    my $r = shift;

    if (my $content_type = shift) {
        $r->content_type($content_type);
    }

    return '';
}
```

send_http_header( ) explicitly returns an empty string, so we don't get any unexpected output.

We can make this plugin available to our templates using the PLUGIN configuration parameter:

```
my $t = Template->new({
    PLUGINS => {
        'apache' => 'TTBook::Template::Plugin::Apache',
    }
});
```

Example 8-9 is the complete TTBook::Template::Plugin::Apache.

Example 8-9. TTBook::Template::Plugin::Apache

```
package TTBook::Template::Plugin::Apache;

use strict;
use vars qw($VERSION);

use Apache;
use base qw(Template::Plugin Apache);

$VERSION = 1.02;

sub new {
    my ($class, $context) = @_;

    bless {
        '_r' => Apache->request,
    } => $class;
}

sub print {
    my ($self, @data) = @_;
    my ($str, $output);

    for $str (@data) {
        if (ref $str eq 'SCALAR') {
            $output .= $$str;
        } else {
            $output .= $str;
        }
    }

    return $output;
```

```
}

sub send_http_header {
    my $r = shift;

    if (my $content_type = shift) {
        $r->content_type($content_type);
    }

    return "";
}

1;
```

8.3.5.2 A more complex wrapper plugin

The next type of plugin is one that is based on an object-oriented module, but that needs configuration or runtime translation; a good example is LWP. LWP provides a web useragent in the LWP::UserAgent class, and a host of supporting modules, representing HTTP requests and responses, server messages, and even robots; using these powerful modules can be complex. We will develop a simple, easy-to-use plugin frontend for LWP::UserAgent; most of the work that we need to do will involve translating data that the Template Toolkit wraps up into hashrefs back into the hashes that the LWP::UserAgent methods are expecting:

```
package TTBook::Template::Plugin::LWP;


use strict;
use base qw(Template::Plugin);


use HTTP::Request;
use LWP::UserAgent;
use Template::Plugin;
```

We would like it to be useable in standard plugin style:

```
[% USE lwp %]
```

perhaps with some specified parameters to indicate the name of the useragent:

```
[% USE lwp(agent => 'TTBook bot/1.0') %]
```

or proxy information:

```
[% USE ua = lwp(env_proxy => 1) %]
```

or all:

```
[% USE lwp(agent     => 'TTBook bot/1.0',
           env_proxy => 1,
           timeout   => 60) %]
```

The constructor for LWP::UserAgent expects a hash of (name, value) pairs, rather than the hashref that the Template Toolkit passes to plugin constructors, which means that we will need to do a little translation. The new( ) method for our plugin, therefore, looks like this:

```
sub new {
    my ($class, $context, $plugin_params) = @_;
    my ($self, $ua, %lwp_params);

    %lwp_params = %$plugin_params;
    $ua = LWP::UserAgent->new(%lwp_params);

    return bless {
        _CONTEXT => $context,
        _UA      =>   $ua,
    } => $class;
}
```

Using the plugin should be simple, too; LWP::UserAgent supports GET, POST, and HEAD requests in the form of the get( ), post( ), and head( ) methods, so our plugin will inherit these, but they will require some parameter mapping to make their calling sequence seem more natural to plugin users. These methods take, as parameters, the request URI and then (name, value) pairs that specify headers; the special header named Content will be used to set the content of the request (for POST and PUT requests), rather than to create a header. Our plugin interface will maintain this split, but, just like the constructor, will need to map from hashref to hash.

These methods can be accessed simply as:

```
[% use.perl.org = lwp.get('http://use.perl.org/') %]
```

The URL plugin can be of great assistance here:

```
[% USE url('http://use.perl.org/journal.pl', light = 1) %]

[% use.perl.org = lwp.get(url(uid = 18)) %]
```

Our plugin doesn't have to do anything to get the benefits of this; url has been dereferenced by the Template Toolkit before, and our method is passed a string.

Our get, post, and head wrappers would look like this:

```
sub get {
    my ($self, $url, $query_params) = @_;
    my %get_params = %$query_params;
    my $ua = $self->{ _UA };

    return $ua->get($url, %get_params);
}


sub head {
    my ($self, $url, $query_params) = @_;
    my %head_params = %$query_params;
    my $ua = $self->{ _UA };
```

```
    return $ua->head($url, %head_params);
}


sub post {
    my ($self, $url, $query_params) = @_;
    my %post_params = %$query_params;
    my $ua = $self->{ _UA };


    return $ua->post($url, %post_params);
}
```

We can use these pretty simply:

```
[% lwp.post(url, 'Content' = my_text) %]
```

We have often wished that there was a general-purpose download method in the LWP::UserAgent class, so let's create one. The request method of the LWP::UserAgent class will write the requested content to a disk file when passed a string as a second argument, so we can begin there:

```
sub download {
    my ($self, $uri, $filename) = @_;
    my ($ua, $context, $request);


    $ua      = $self->{ _UA };
    $context = $self->{ _CONTEXT };
```

We can't just defer to the get method of LWP::UserAgent here; we'll need to use HTTP::Request directly:

```
$request = HTTP::Request->new(GET => $uri);
```

(We assume a GET request; implementing download for other request types is left as an exercise for the reader.)

```
$ua->request($request, $filename)
    || $context->throw('file', "Can't write $filename: $!");
```

Because this method is writing to the filesystem, there is the possibility that it can fail; this needs to be checked for success. If the write fails, we throw a file exception using $context.

Finally, we return the content of the response:

```
return $response->content;
}
```

Making our LWP plugin available to templates can be achieved by passing it as an element of the PLUGINS hash:

```
my $t = Template->new({
    PLUGINS => {
        'lwp'   => 'TTBook::Template::Plugin::LWP',
    }
});
```

Example 8-10 is the complete TTBook::Template::Plugin::LWP.

Example 8-10. TTBook::Template::Plugin::LWP

```
package TTBook::Template::Plugin::LWP;

use strict;
use vars qw($VERSION);
use base qw(Template::Plugin);

use HTTP::Request;
use LWP::UserAgent;
use Template::Plugin;

$VERSION = 1.00;

sub new {
    my ($class, $context, $plugin_params) = @_;
    my ($self, $ua, %lwp_params);

    %lwp_params = %$plugin_params;
    $ua = LWP::UserAgent->new(%lwp_params);

    return bless {
        _CONTEXT => $context,
        _UA      =>  $ua,
    } => $class;
}

sub get {
    my ($self, $url, $query_params) = @_;
    my %get_params = %$query_params;
    my $ua = $self->{ '_UA' };

    return $ua->get($url, %get_params);
}

sub head {
    my ($self, $url, $query_params) = @_;
    my %head_params = %$query_params;
    my $ua = $self->{ _UA };
```

```
    return $ua->head($url, %head_params);
}

sub post {
    my ($self, $url, $query_params) = @_;
    my %post_params = %$query_params;
    my $ua = $self->{ _UA };

    return $ua->post($url, %post_params);
}

sub download {
    my ($self, $uri, $filename) = @_;
    my ($ua, $context, $request);

    $ua      = $self->{ _UA };
    $context = $self->{ _CONTEXT };
    $request = HTTP::Request->new(GET => $uri);

    $ua->request($request, $filename)
        || $context->throw('file', "Can't write $filename: $!");

    return $response->content;
}

1;
```

8.3.5.3 A plugin that sends mail

Sending mail is such a common thing to do with the Template Toolkit, it is surprising that there is no standard plugin to handle it. Many mail-related Perl modules are on CPAN, but the simplest is Mail::Sendmail, which exports a single subroutine (sendmail) that takes a hash of arguments. We can use this as the basis for our Mail plugin.

A mail plugin would need to have methods to get and set the To, From, Cc, Bcc, Subject, and Body fields:

```
[% Mail.To('you@yourhost.com') %]
[% Mail.From('me@myhost.com') %]
[% Mail.Subject('Re: your mail') %]

[% body = BLOCK %]
Hello, friend!
[% END %]

[% Mail.Body(body) %]
```

```
[% Mail.Body(body) %]
```

Additionally, it would be nice to be able to reuse the plugin instance—in a loop, for example:

```
[% addresses = [ 'one@addr.ess'
                 'two@addr.ess'
                 'three@addr.ess'
                 'four@addr.ess'
               ];

  message_content = 'The system will be down blah blah blah.';

  USE Mail from    => 'Administrator <admin@addr.ess>',
           subject => 'Scheduled downtime',
           body    => message_content;

  FOREACH address = addresses;
      Mail.send(to => address);
      Mail.reset;
  END;
%]
```

Our plugin begins fairly predictably:

```
package TTBook::Template::Plugin::Mail;

use strict;
use base qw(Template::Plugin);
use vars qw($VERSION $AUTOLOAD);

use Mail::Sendmail;
use Net::Domain qw(hostfqdn);
use Template::Exception;
use Template::Plugin;

$VERSION = 1.00;
$AUTOLOAD = undef;
```

We'll be using Template::Exception to propagate errors, so they can be caught and handled appropriately. Net::Domain gives us hostfqdn, which will help us generate a Message-ID header. We'll need $VERSION and $AUTOLOAD later, so we declare them now.

Because we want the user to be able to invoke our plugin not only as:

```
[% USE Mail %]
```

but also with default arguments:

```
[% USE Mail subject = 'Testing, testing, testing'
            from = 'admin@template-toolkit.org' %]
```

we can write new to accept parameters:

```
sub new {
    my ($class, $context, $params) = @_;
    my $self;
```

As you recall, named parameters are passed to subroutines as the last element in @_, as a reference to a hash; any parameters that the user specifies in the USE line will be there.

```
$params->{ server } = 'mailhost'
    unless defined $params->{ server };
```

Mail::Sendmail requires the name of the SMTP relay to be specified as one of its arguments, but we'll take that responsibility out of the user's hands and use a reasonable default. Savvy users can still specify a server to use, for example:

```
[% USE Mail server => 'localhost' %]
```

In order to reuse our plugin, we'll need to keep the default configuration values separate from values set later. To do this, we will use two data members for parameters:

```
$self = bless {
    _CONTEXT       => $context,
    _ORIG_PARAMS   => $params,
    _PARAMS        => { },
    _LOGMESSAGE    => '',
} => $class;
```

_ORIG_PARAMS is the configuration parameters that were specified at instance creation time and that will be used as our defaults. We finish our new( ) method with:

```
$self->reset( );
return $self;
    }
```

The reset( ) method is responsible for copying the elements of _ORIG_PARAMS into _PARAMS:

```
sub reset {
    my $self = shift;
    delete $self->{ _ORIG_PARAMS }->{ 'message-id' };
    %{ $self->{ _PARAMS } } = %{ $self->{ _ORIG_PARAMS } };
    $self->{ _LOGMESSAGE } = '';
    return $self;
}
```

reset( ) takes the precaution of deleting the Message-ID key: because this must be unique for each outgoing email, we don't take the chance that the user hasn't specified it manually. We also reset the _LOGMESSAGE string, which will contain a transcript of the conversation with the server.

The most important method, send, is very straightforward. It is used like this:

```
[% Mail.send(params) %]
```

This is our last chance to specify parameters—they will be mixed in with _PARAMS. Mail::Sendmail provides a transcript of its communications with the server in the $Mail::Sendmail::log variable; we'll store this in the _LOGMESSAGE instance variable.

```
sub send {
    my $self = shift;
    my ($params, $context) = @$self{ qw( _PARAMS _CONTEXT) };
    my $mail = ref($_[-1]) eq 'HASH' ? pop @_ : { };
```

```
    %$mail = ('X-Mailer' => join('/', ref $self, $VERSION),
              %$params,
              %$mail);

    $mail->{'message-id'} = $self->generate_mid( )
        unless defined $mail->{'message-id'};

    sendmail(%$mail)
        or $context->throw('mail', $Mail::Sendmail::error);

    $self->{ _LOGMESSAGE } = $Mail::Sendmail::log;

    return '';
}
```

Both $params and $mail are hash references, so they can be dereferenced sequentially to produce one hash. Because $mail is dereferenced after $params, any keys defined in $mail supercede those in $params—which is to say that parameters specified in send override those set earlier. Finally, we add a vanity header (X-Mailer), which also can be overridden by either $params or $mail:

```
[% Mail.send('X-Mailer' => 'Micros~1 Outlook 6.6.6') %]
```

The send method returns an empty string so that there is no unintentional output when it is invoked.

We need to explicitly create a Message-ID header if one hasn't been provided by the user. Most MTAs will add a Message-ID header if it isn't present, but many will not, so we cannot rely on it. The Message-ID header will be used to uniquely identify a message in space and time; ideally, it should consist of enough information to identify the message without giving away too much information about the user. The generate_mid method creates a Message-ID based on the time, domain name, and eight characters of randomness ($junk):

```
sub generate_mid {
    my $self = shift;
    my @time = localtime;
    my $junk = join '', map { ('a'..'z', 'A'..'Z')[rand 52] } (0..8);

    my $mid = sprintf '<%d%02d%02d.%s@%s>',
        $time[5] + 1900, $time[4], $time[3], $junk, hostfqdn( );

    return $mid;
}
```

We can access the transcript using the logmessage( ) method:

```
sub logmessage {
    my $self = shift;
    return $self->{ _LOGMESSAGE };
}
```

Finally, the other methods can be handled by an AUTOLOAD method:

```
my %multi = map { $_ => 1 } qw(to cc bcc);
```

```
sub AUTOLOAD {
    my $self = shift;
    my ($method, $item);

    $method = $AUTOLOAD;
    $method =~ s/.*:://;
    $method = ucfirst lc $AUTOLOAD;
    $method =~ s/_(\w)/-\u$1/g;

    # Make an alias
    $item = \$self->{ _PARAMS }->{ $method };

    if (@_) {
        if (defined $multi{ $method }) {
            my @addrs;
            if (ref $_[0] eq 'ARRAY') {
                @addrs = @{$_[0]};
            } else {
                @addrs = @_;
            }
            $$item = join ', ', @addrs;
        } else {
            $$item = shift @_;
        }
        return '';
    }

    return $$item;
}
```

Perl's AUTOLOAD facility catches calls for methods that do not exist (which makes it perfect as a catchall method for this plugin). Mail::Sendmail will pass on any parameters passed to the sendmail( ) function as headers; we can combine these two facts to let Perl write the rest of our methods for us. When AUTOLOAD is invoked, the name of the invoked method is in the variable $AUTOLOAD, with the fully qualified package name. Mail::Sendmail takes header names in any case, but we normalize it (by lowercasing) to keep from storing duplicates in _PARAMS. Using this AUTOLOAD, we can set any arbitrary header, not just the ones mentioned earlier:

```
[% Mail.message_id('20030811-093159@localhost') %]

[% Mail.x_pgp_fingerprint(pgp_f) %]
```

To, Cc, and Bcc can be multivalued elements (as defined in %multi), so we accept a list of elements. This allows us to do this:

```
[% Mail.To(address1, address2, address3) %]
```

We also explicitly check to see whether $_[0] is an array reference, and dereference it if it is. This is because if we pass a list created in our template, it will be an array reference:

```
[% addresses = [ 'one@addr.ess',
                 'two@addr.ess',
                 'three@addr.ess'
               ];
 Mail.To(addresses) %]
```

If we are setting a value, we explicitly return the empty string, so there are no side effects.

Because send throws an exception if it cannot contact the mail server, or if something else goes wrong, we need to wrap calls to send in a TRY...CATCH block:

```
[% TRY %]
   [% Mail.send %]
[% CATCH mail %]
   Error: [% error %]
[% END %]
```

The last thing to do is to make the plugin available to our templates:

```
my $t = Template->new({
    PLUGINS => {
        'mail' => 'TTBook::Template::Plugin::Mail',
    }
});
```

Example 8-11 is the complete TTBook::Template::Plugin::Mail.

Example 8-11. TTBook::Template::Plugin::Mail

```
package TTBook::Template::Plugin::Mail;

use strict;
use base qw(Template::Plugin);
use vars qw($VERSION $AUTOLOAD);

use Mail::Sendmail;
use Net::Domain qw(hostfqdn);
use Template::Exception;
use Template::Plugin;

$VERSION = 1.00;
$AUTOLOAD = undef;

sub new {
    my ($class, $context, $params) = @_;
    my $self;
```

```
    $params->{ server } = 'mailhost'
        unless defined $params->{ server };

    $self = bless {
        _CONTEXT        => $context,
        _ORIG_PARAMS    => $params,
        _PARAMS         => { },
        _LOGMESSAGE     => '',
    } => $class;

    $self->reset( );
    return $self;
}

sub reset {
    my $self = shift;
    delete $self->{ _ORIG_PARAMS }->{ 'message-id' };
    %{ $self->{ _PARAMS } } = %{ $self->{ _ORIG_PARAMS } };
    $self->{ _LOGMESSAGE } = '';
    return $self;
}

sub send {
    my $self = shift;
    my ($params, $context) = @$self{ qw( _PARAMS _CONTEXT) };
    my $mail = ref($_[-1]) eq 'HASH' ? pop @_ : { };

    %$mail = ('X-Mailer' => join('/', ref $self, $VERSION),
              %$params,
              %$mail);

    $mail->{'message-id'} = $self->generate_mid( )
        unless defined $mail->{'message-id'};

    sendmail(%$mail)
        or $context->throw('mail', $Mail::Sendmail::error);

    $self->{ _LOGMESSAGE } = $Mail::Sendmail::log;
```

```
    return '';
}

sub generate_mid {
    my $self = shift;
    my @time = localtime;
    my $junk = join '', map { ('a'..'z', 'A'..'Z')[rand 52] } (0..8);

    my $mid = sprintf '<%d%02d%02d.%s@%s>',
        $time[5] + 1900, $time[4], $time[3], $junk, hostfqdn( );

    return $mid;
}

sub logmessage {
    my $self = shift;
    return $self->{ _LOGMESSAGE };
}

my %multi = map { $_ => 1 } qw(to cc bcc);
sub AUTOLOAD {
    my $self = shift;
    my ($method, $item);

    $method = $AUTOLOAD;
    $method =~ s/.*:://;
    $method = ucfirst lc $method;
    $method =~ s/_(\w)/-\u$1/g;

    # Make an alias
    $item = \$self->{ _PARAMS }->{ $method };

    if (@_) {
        if (defined $multi{ $method }) {
            my @addrs;
            if (ref $_[0] eq 'ARRAY') {
                @addrs = @{$_[0]};
            } else {
```

```
                @addrs = @_;
            }
            $$item = join ', ', @addrs;
        } else {
            $$item = shift @_;
        }
        return '';
    }

    return $$item;
}

1;
```

8.3.5.4 GoogleSearch

Everybody loves Google, right? Since the advent of the Google API, everybody can write their own custom search interface. Aaron Straup Cope's Net::Google provides a nice, simple Perl interface to the Google SOAP API.

In order to use this plugin, you'll need to register with Google; you can do so at http://api.google.com/.

Using the GoogleSearch plugin should be straightforward:

```
[% USE g = GoogleSearch('Template Toolkit') %]
[% num = g.num_results %]

[% FOREACH result = g.results %]
    [% result.title %]
    [% result.URL %]
[% END %]
```

The plugin starts with the usual prologue:

```
package TTBook::Template::Plugin::GoogleSearch;

use strict;
use vars qw($VERSION $KEY);
use base qw(Template::Plugin);

use Net::Google;
use Template::Exception;
use Template::Iterator;
use Template::Plugin;

$VERSION = 1.00;
$KEY = 'cc42973b5c5f292a7be146e1b444379e';
```

$KEY is your Google key. Don't use the one in the preceding code because it isn't real (it's the MD5 hash of the string Template Toolkit).

Net::Google works by creating and reusing a Net::Google instance, which acts as a factory for Net::Google::Search instances. The best way to represent this is by using the singleton plugin pattern described earlier:

```
sub load {
    my ($class, $context) = @_;
    my $google = Net::Google->new(key => $KEY);

    bless {
        _CONTEXT => $context,
        _GOOGLE => $google,
    } => $class;
}
```

We will need $context for throwing exceptions.

new( ) is where we create the Net::Google::Search instance:

```
sub new {
    my ($self, $context, @args) = @_;
    my ($params, $google, $search, $p);

    $params = ref $args[-1] eq 'HASH' ? pop @args : { };

    $google = $self->{ _GOOGLE };
    $search = $self->{ _SEARCH } = $google->search( );

    for $p (qw/ lr ie oe starts_at
               max_results safe filter /) {
        $search->$p($params->{$p})
            if defined $params->{$p};
    }

    $search->query(join ' ', @args);

    return $self;
}
```

Search terms are provided as positional arguments, while other elements of the search are provided as named arguments:

```
[% USE g = GoogleSearch max_results = 50
                      lr = [ 'de' 'es' ]
                      'perl'
                      '"templating languages"' %]
```

This search, for *perl* and *templating languages*, will return up to 50 results (instead of the default 10) and will search

German and Spanish pages only. (See the Net::Google::Search manpage for what the available parameters actually are.)

Our result set will be wrapped in a Template::Iterator instance:

```
sub results {
    my $self = shift;
    my ($search, @results, $iter);

    $search = $self->{ _SEARCH } || return Template::Iterator->new([  ]);
    @results = @{$search->results( )};
    $iter = Template::Iterator->new(\@results);

    return $iter;
}
```

Each element in the iterator is a Result object (created by the Net::Google::Response object), and has methods useable to access the elements of the result:

```
[% FOREACH result = g.results %]
    blah blah blah
```

Example 8-12 is the complete TTBook::Template::Plugin::GoogleSearch.

Example 8-12. TTBook::Template::Plugin::GoogleSearch

```
package TTBook::Template::Plugin::GoogleSearch;

use strict;
use vars qw($VERSION $KEY);
use base qw(Template::Plugin);

use Net::Google;
use Template::Exception;
use Template::Iterator;
use Template::Plugin;

$VERSION = 1.00;
$KEY = "cc42973b5c5f292a7be146e1b444379e";

sub load {
    my ($class, $context) = @_;
    my $google = Net::Google->new(key => $KEY);

    bless {
        _CONTEXT => $context,
        _GOOGLE => $google,
```

```
    } => $class;
}

sub new {
    my ($self, $context, @args) = @_;
    my ($params, $google, $search, $p);

    $params = ref $args[-1] eq 'HASH' ? pop @args : { };

    $google = $self->{ _GOOGLE };
    $search = $self->{ _SEARCH } = $google->search( );

    for $p (qw/ lr ie oe starts_at
               max_results safe filter /) {
        $search->$p($params->{$p})
            if defined $params->{$p};
    }

    $search->query(join " ", @args);

    return $self;
}

sub results {
    my $self = shift;
    my ($search, @results, $iter);

    $search = $self->{ _SEARCH } ||
        return Template::Iterator->new([  ]);
    @results = @{$search->results( )};
    $iter = Template::Iterator->new(\@results);

    return $iter;
}

1;
```

8.3.5.5 Normalizing URLs

For some reason, many organizations find it difficult to keep their URLs consistent. This plugin might be helpful: given a relative URL, it will return the canonical version of it, relative to either the main host, or to the graphics host if the link looks like it might be an image. For example:

```
[% USE Link www_host = 'www.example.com' %]

<a href="[% link('/foo/bar.html') %]">...</a>
```

will produce:

```
<a href="http://www.example.com/foo/bar.html">...</a>
```

This Link plugin accepts a few arguments: www_host, graphics_host, and opaque. graphics_host will be used for things that appear to be images, and www_host will be used for everything else. If opaque is specified, the resulting URL will not have a scheme; this is most useful for templates that might be served under multiple protocols—for example, *http* and *https*. The client will assume the current scheme if one is not provided, so the server does not have to check whether the current page is secure.

```
[% USE Link www_host      = 'www.tt2.org',

           graphics_host = 'graphics.tt2.org',

           opaque        = 1

%]

<a href="[% link('/index.html') %]">

   <img src="[% link('ttpower.gif') %]">

</a>
```

Calls to link( ) would expand to full URIs:

```
<a href="//www.tt2.org/index.html">

   <img src="//graphics.tt2.org/ttpower.gif">

</a>
```

The URI referring to an image was detected, and the host was set to the graphics server.

It would be straightforward to modify this plugin to treat arguments to link as keywords rather than filenames.

Example 8-13 is the complete TTBook::Template::Plugin::Link.

Example 8-13. TTBook::Template::Plugin::Link

```
package TTBook::Template::Plugin::Link;


use strict;

use vars qw($VERSION $DEFAULT_WWW_HOST $DEFAULT_GRAPHICS_HOST $DEFAULT_OPAQUE);

use base qw(Template::Plugin);


use LWP::MediaTypes qw(guess_media_type);

use URI;


$VERSION       = 1.00;

$DEFAULT_WWW_HOST      = "www.example.com";

$DEFAULT_GRAPHICS_HOST = "graphics.example.com";

$DEFAULT_OPAQUE        = 0;


sub load {
```

```
    my ($class, $context, @args) = @_;
    my $params = ref $args[-1] eq 'HASH' ? pop @args : { };

    $context->stash->set("link", link_factory($params));

    bless { } => $class;
}

# Nominal new; can't inherit from Template::Plugin
sub new { return shift }

sub link_factory {
    my $params        = shift;
    my $www_host      = sprintf "http://%s/", $params->{ www_host }
                                || $DEFAULT_WWW_HOST;
    my $graphics_host = sprintf "http://%s/", $params->{ graphics_host }
                                || $DEFAULT_GRAPHICS_HOST;
    my $opaque        = $params->{'opaque'} || $DEFAULT_OPAQUE;

    return sub {
        my $url = shift || return;

        my $link = URI->new($url);

        # This will be the case for URIs such as "/foo", which
        # URI will decide are of type "URI::_generic"
        $link = URI->new($link, "http")->abs($www_host)
            unless ($link->can("host"));

        $link->host($graphics_host)
            if (guess_media_type($url) =~ /^image/);

        return $opaque ? $link->opaque( ) : $link->canonical( );
    };
}

1;
```

8.4 Building a New Frontend

The Template module is the default frontend to the Template Toolkit, but there are others. The Apache::Template module, available from CPAN, is one, as are the familiar *tpage* and *ttree*. Here is a description of these default frontends:

Template

The Template module is the frontend that most users are familiar with. Template provides the familiar process method:

```
$tt->process($input, $vars, $output)
    || die $tt->error( );
```

Template uses the underlying Template::Service instance internally to process $input, and then redirect that output appropriately, based on the third argument to process( ) (see Chapter 7 for details).

Apache::Template

The Apache::Template module provides a simple interface to the Template Toolkit from Apache/mod_perl. Apache::Template allows configuration to be handled in an Apache-specific manner, using directives in Apache's *httpd.conf* configuration file.

Apache::Template is covered in Chapter 12. The Appendix lists valid Apache::Template-related *httpd.conf* configuration directives.

tpage and ttree

We've already met *tpage* and *ttree* in Chapter 1 and Chapter 2; these two scripts are also Template Toolkit frontends.

A Template Toolkit frontend manages the Template::Service instance, and, generally, manages input and output. In this section, we look at these standard frontends and how to build a custom frontend for email.

8.4.1 Mail::Template

Because email is basically text, and generating text is so simple using the Template Toolkit, why isn't there a dedicated mail frontend? Well, there could be; let's develop one.

Our Template Toolkit frontend module needs two user-facing methods, new and process. The Template::Base module implements most of the common functionality of the modules that ship with the Template Toolkit, so we can start there:

```
package Mail::Template;


use strict;
use vars qw($VERSION $MAILHOST $MAILPORT);
use base qw(Template::Base);


use Mail::Sendmail qw(sendmail);
use Template::Base;


$VERSION = 1.00;
$MAILHOST = "mailhost" unless defined $MAILHOST;
$MAILPORT = 25 unless defined $MAILPORT;
```

The Mail::Sendmail module provides the sendmail function, which, well, sends mail. $MAILHOST and $MAILPORT are defined as package variables so that the defaults can be overridden in client code:

```
use Mail::Template;

$Mail::Template::MAILHOST = "smtp.example.com";
```

The new method inherited from Template::Base calls the _init method, which Mail::Template can use to handle specific constructor details. _init is called with a reference to a hash containing the parameters passed to new.

```
sub _init {

    my ($self, $config) = @_;



    $self->{ _MAILHOST } = $config->{ MAILHOST } || $MAILHOST;



    if (not defined $config->{ MAILPORT }) {

        if ($self->{ _MAILHOST } =~ s/:(\d+)$//) {

            $self->{ _MAILPORT } = $1;

        }

        else {

            $self->{ _MAILPORT } = $MAILPORT;

        }

    }



    # Setup a Template::Service instance

    $self->{ SERVICE } = $config->{ SERVICE }

        || Template::Config->service($config)

        || return $self->error(Template::Config->error);



    return $self;

}
```

Mail::Template looks for two unique parameters: MAILHOST and MAILPORT, both of which are assigned reasonable defaults (mailhost and 25, respectively). We can use an alternate port or host by passing them specifically, or the two can be joined with a colon as MAILHOST:

```
my $config = { MAILHOST => "smtp-server:2525" };

my $mt = Mail::Template->new($config);
```

The Template::Service instance is created as an idiom that occurs in many places throughout the Template Toolkit. The error method, which is inherited from Template::Base, does double-duty: if called without an argument, it returns the most recent error message, but if called with an argument, it sets the error data field and returns undef. The Template::Config class defines methods for instantiating all of the major components of the Template Toolkit in one easy-to-use, easy-to-override place. Any other parameters specified to the Mail::Template constructor will be passed on to the objects that the Template::Service instance creates.

The format of the process method is modeled after Template::process:

```
sub process {

    my ($self, $input, $vars, $addrs, @opts) = @_;

    my ($output, $error);

    my $service = $self->{ SERVICE };

    my $options = (@opts =  = 1) && ref($opts[0]) eq 'HASH'
```

```
        ? shift(@opts) : { @opts };
    $addrs = ref($addrs) eq 'ARRAY' ? $addrs : [ $addrs ];

    return $self->error("No recipients specified")
        unless @$addrs;

    $output = $service->process($input, $vars);

    if (defined $output) {
        $options->{ To      }   = $addrs;
        $options->{ Message }   = $output;
        $options->{ Server  } ||= $self->{ MAILHOST };
        $options->{ Port    } ||= $self->{ MAILPORT };

        if (sendmail(%$options)) {
            return 1;
        }
        else {
            return $self->error($Mail::Sendmail::error);
        }
    }
    else {
        return $self->error($service->error);
    }
}
```

Just like Template::process, Mail::Template::process can take up to four arguments: the template to be processed; a reference to a hash of parameters; a reference to a list of addresses; and a reference to a hash of mail options, which will be used to set mail-specific headers, such as Subject and From:

```
my $friends = [ qw(abw@cpan.org dave@dave.org.uk) ];
my $options = {
    Subject => "Testing Mail::Template",
    From => "Darren Chamberlain <darren@cpan.org>",
};

$mt->process($input, $vars, $friends, $options)
    || die $tt->error;
```

The processing of the template is handled by the Template::Service instance, which was created in _init. This leaves only the sending of the mail for process to handle (we farm that out to Mail::Sendmail).

Example 8-14 is the complete Mail::Template.

Example 8-14. Mail::Template

```
package Mail::Template;

use strict;
use vars qw($VERSION $MAILHOST $MAILPORT);
use base qw(Template::Base);

use Mail::Sendmail qw(sendmail);
use Template::Base;

$VERSION = 1.00;
$MAILHOST = "mailhost" unless defined $MAILHOST;
$MAILPORT = 25 unless defined $MAILPORT;

sub _init {
    my ($self, $config) = @_;

    $self->{ _MAILHOST } = $config->{ MAILHOST } || $MAILHOST;

    if (not defined $config->{ MAILPORT }) {
        if ($self->{ _MAILHOST } =~ s/:(\d+)$//) {
            $self->{ _MAILPORT } = $1;
        }
        else {
            $self->{ _MAILPORT } = $MAILPORT;
        }
    }

    # Set up a Template::Service instance
    $self->{ SERVICE } = $config->{ SERVICE }
        || Template::Config->service($config)
        || return $self->error(Template::Config->error);

    return $self;
}

sub process {
    my ($self, $input, $vars, $addrs, @opts) = @_;
    my ($output, $error);
```

```
    my $service = $self->{ SERVICE };
    my $options = (@opts =  = 1) && ref($opts[0]) eq 'HASH'
        ? shift(@opts) : { @opts };
    $addrs = ref($addrs) eq 'ARRAY' ? $addrs : [ $addrs ];

    return $self->error("No recipients specified")
        unless @$addrs;

    $output = $service->process($input, $vars);

    if (defined $output) {
        $options->{ To      }   = $addrs;
        $options->{ Message }   = $output;
        $options->{ Server  } ||= $self->{ MAILHOST };
        $options->{ Port    } ||= $self->{ MAILPORT };

        if (sendmail(%$options)) {
            return 1;
        }
        else {
            return $self->error($Mail::Sendmail::error);
        }
    }
    else {
        return $self->error($service->error);
    }
}

1;
```

8.4.2 Custom Apache Handlers

In many ways, writing a mod_perl-based frontend is easier than writing other types of frontends because it doesn't need to be as flexible. There is only one way that your handler will be called, and you know exactly what arguments will be provided. There are a few things to keep in mind when writing this frontend, though; a primary goal should be to avoid recreating Template Toolkit components whenever possible, especially expensive objects such as the parser. Providing full access to the request object and the metadata associated with it, such as cookies and form parameters, is also very important.

The differences between Apache 1.3 and Apache 2.0 make themselves known only in the machinery needed to make the handler work; the Template Toolkit aspects are identical. Let's take a look at a simple Apache 1.3/mod_perl 1.x handler:

```
package TTBook::ApacheHandler;

use strict;
use vars qw($VERSION);

$VERSION = 1.00;    # Apache 1.3.x handler

use Apache;
use Apache::Constants qw(OK SERVER_ERROR);
use Template::Config;
use URI::Escape qw(uri_unescape);

# Preload all Template Toolkit modules
Template::Config->preload( );

my $tt;
```

We'll need the OK, DECLINED, and SERVER_ERROR constants—OK for when there are no problems, SERVER_ERROR for when there are, and DECLINED so that we can specifically decline to handle requests for files that don't exist (or requests for things that aren't files, such as directories). Using DECLINED like this means that Apache's normal error handlers can be used for 404's and the like.

Using Template::Config and getting a service instance through Template::Config->service means that we can use a custom subclass without having to change our handler code. The Template Toolkit will defer loading modules until they are needed, but calling Template::Config->preload will force all of them to be loaded immediately. Under mod_perl, this is important because modules compiled in the parent process will reside in the segment of memory shared among all the child processes, which can result in memory savings.

We use a package-scoped lexical variable, $tt, to store our service instance so that it can be shared between multiple requests by the same child:

```
sub handler {
    my $r = shift;
    my ($filename, $docroot, %vars, $template, $content);

    $filename = $r->filename;
    $docroot = $r->docroot;

    return DECLINED unless -f $filename;
```

If this is the first time the current child process has been called up to handle a template, $tt will not be defined. We define it here, and check for errors:

```
$tt ||= do {
    Template::Config->service({
        INCLUDE_PATH => [ $docroot ],
    });
```

```
};

unless (defined $tt) {
    # Catch errors here, and return SERVER_ERROR
    my $mod = $Template::Config::SERVICE;
    $r->log_error("Can't create $mod instance: ",
                  Template::Config->error);
    return SERVER_ERROR;
}
```

If creating a Template::Service instance fails, we need to report it. A well-behaved mod_perl script will write to Apache's error_log and the best way to do that is to use the Apache object's log_error method. We feed it the error according to Template::Config.

We can make query parameters available as top-level variables so that a request for /news/2003/08/11?article=34293 makes a variable called article available within the templates:

```
[% article %]
```

In list context, both $r->args and $r->content return a hash of variables, which is, conveniently enough, what we will need to pass to process:

```
%vars = $r->method eq 'POST' ? $r->content : $r->args;
```

Apache doesn't make the parsed cookies available, but they can be pulled out pretty easily:

```
my @cookies = split /;\s*/, $r->header_in('cookie');

for my $cookie (@cookies) {
    my ($name, $value) = map { uri_unescape($_) } split /=/, $cookie;
    $vars{$name} = $value;
}
```

This makes cookies available as top-level variables, just like query parameters.

The service instance uses the DocumentRoot for its INCLUDE_PATH, so we need to strip it from the filename. A request for something like /news/2003/08/11 will be resolved to a filename such as */var/www/news/2003/08/11*, which we then turn into *news/2003/08/11*:

```
($template = $filename) =~ s,^\Q$docroot\E/?,,;
```

We pass $template to the service instance to process and check for errors. Again, we return SERVER_ERROR if something goes wrong. A more robust implementation might check whether TOLERANT was set, and return DECLINED so that the next content handler in line gets a shot (which might be Apache's default-handler):

```
$content = $tt->process($template, \%vars) || do {
    $r->log_error("$template returned no content: ",
                  $tt->error);
    return SERVER_ERROR;
};
```

At this point, $content contains the results of processing our template, and control is returned to our handler. We can add some extra header fields to the response (such as Content-Length) call $r->print($content) to tell Apache to send the data to the client, and return OK to tell Apache that we handled the request successfully:

```
$r->content_type('text/html');
$r->headers_out->add('Content-Length', length($content));
$r->send_http_header;

$r->print($content);

    return OK;
}

1;
```

You might have noticed that this handler makes no attempt to account for virtual hosts. A reasonable way to use this module—or one like it—with virtual hosts is to store the service instances in a hash keyed by `$r->server_name`; then each virtual host will have its own set of template objects.

Setting up `TTBook::ApacheHandler` within *httpd.conf* is very similar to setting up `Apache::Template`:

```
<Files *.html>
    SetHandler perl-script
    PerlHandler TTBook::ApacheHandler
</Files>
```

Example 8-15 is the complete `TTBook::ApacheHandler`.

Example 8-15. TTBook::ApacheHandler

```
package TTBook::ApacheHandler;

use strict;
use vars qw($VERSION);

$VERSION = 1.00;    # Apache 1.3.x handler

use Apache;
use Apache::Constants qw(OK SERVER_ERROR);
use Template::Config;
use URI::Escape qw(uri_unescape);

# Preload all Template Toolkit modules
Template::Config->preload( );

my $tt;

sub handler {
    my $r = shift;
```

```
my ($filename, $docroot, %vars, $template, $content);

$filename = $r->filename;
$docroot = $r->docroot;

return DECLINED unless -f $filename;

$tt ||= do {
    Template::Config->service({
        INCLUDE_PATH => [ $docroot ],
    });
};

unless (defined $tt) {
    # Catch errors here, and return SERVER_ERROR
    my $mod = $Template::Config::SERVICE;
    $r->log_error("Can't create $mod instance: ",
                  Template::Config->error);
    return SERVER_ERROR;
}

%vars = $r->method eq 'POST' ? $r->content : $r->args;

my @cookies = split /;\s*/, $r->header_in('cookie');
for my $cookie (@cookies) {
    my ($name, $value) = map { uri_unescape($_) } split /=/, $cookie;
    $vars{$name} = $value;
}

($template = $filename) =~ s,^\Q$docroot\E/?,,;

$content = $tt->process($template, \%vars) || do {
    $r->log_error("$template returned no content: ",
                  $tt->error);
    return SERVER_ERROR;
};

$r->content_type('text/html');
$r->headers_out->add('Content-Length', length($content));
```

```
    $r->send_http_header;

    $r->print($content);

    return OK;
}

1;
```

8.5 Changing the Language

The grammar for the Template Toolkit language is generated using a YACC-like parser generator written in Perl called Parse::Yapp (http://search.cpan.org/dist/Parse-Yapp/). Parse::Yapp is not distributed with or required by the Template Toolkit, but you will need it if you want to regenerate the grammar. Yapp is identical to YACC in all the important ways; for a good general introduction to YACC, see *lex & yacc*, Second Edition, by John R. Levine, Tony Mason, and Doug Brown (O'Reilly), which gives a good introduction to the principles of an LALR parser and how to define grammars in YACC. See also the Parse::Yapp documentation and the comments in Template::Parser for more information. For an in-depth study of parser and compiler theory, consult *Compilers: Principles, Techniques and Tools* (a.k.a., the "Dragon Book") by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman (Addison Wesley).

The Template Toolkit source distribution includes the subdirectory *parser*, which contains a few files, most notably one called *Parser.yp*. This is the one you will be modifying to extend the language.[3] The parser grammar is compiled by *yapp*, the frontend script to Parse::Yapp, based on the grammar skeleton *Grammar.pm.skel*, which is also in the *parser* directory.

[3] Be sure to have a backup of the file handy while you are modifying the grammar!

Changing the grammar is a simple process, in theory at least, if you're familiar with Yapp/YACC. In practice, it also requires some insight into the inner workings of the Template Toolkit.

8.5.1 Building the Grammar

The Template Toolkit distribution includes a helper script called *yc*, which builds the grammar. It is a thin wrapper around yapp that sets the appropriate options to compile, emit, and save the Perl code for the grammar. Here it is in its entirety:

```
#!/bin/sh



: ${GRAMMAR:="Parser.yp"}

: ${OUTPUT:="../lib/Template/Grammar.pm"}

: ${TEMPLATE:="Grammar.pm.skel"}



echo "Compiling parser grammar (${GRAMMAR} -> ${OUTPUT})"



yapp -v -s -o ${OUTPUT} -t ${TEMPLATE} ${GRAMMAR}
```

yc takes the grammar defined in *Parser.yp* and plugs it into the skeleton module file, *Grammar.pm.skel*. The output is written to *lib/Template/Grammar.pm*, clobbering anything that was there before. A report detailing the status of the compilation process is written to *Parser.output*:

```
$ ./yc

Compiling parser grammar (Parser.yp -> ../lib/Template/Grammar.pm)
```

yc writes the output to the *../lib/Template/Grammar.pm* file by default, so you'll need to modify the script accordingly (or set the OUTPUT environment variable) if you want to compile your own grammar module with it.

Be prepared to become intimately familiar with the (rather verbose) output in the *Parser.output* file if you're planning on writing your own grammar or making major changes to the existing grammar. Often *yapp* will refuse to compile grammar, or raise warnings about *conflicts* that indicate ambiguities in the grammar that it can't automatically resolve. In these cases, you'll need to carefully inspect the error report in *Parser.output* and trace through the rules and states listed to try and figure out where you went wrong. A good compiler reference book will be invaluable at this stage.

8.5.2 Extending the Existing Grammar

In most cases, you will be modifying the grammar because you have a specific feature or syntax element in mind that you want to be part of the core language, or your version of it. Many things can be done with plugins or filters, but you are still bound by the syntax of the language.

The Template Toolkit display language is very rich, and lacks very few control structures or directives. But occasionally, something will stand out as particularly expressive or helpful. With that in mind, let's add a feature to the language: UNTIL. UNTIL is logically equivalent to WHILE NOT, but can make for cleaner templates:

```
[% UNTIL count =  = 100 %]

    [% do.something.to(count) %]

[% END %]
```

Because UNTIL is a variation of WHILE, we can probably get away with mimicking the WHILE implementation, and simply negating the condition test. This simple implementation will give us a chance to poke around the grammar a bit.

We'll start in *parser/Parser.yp*. Download a fresh tarball (or get a new CVS checkout) of the Template Toolkit sources, and let's begin.

Parse::Yapp

As mentioned earlier, Parse::Yapp is very similar to *yacc*, and the format of the grammar file is also very similar. It consists of three main sections, divided by %%; the first section is the *preamble*, the last section is the *postamble*, and the middle section consists of sets of *rules* that define the structure of the language being represented. These rules are in the form:

```
rule: production1 | production2 | production3 ;
```

A *production* consists of two parts: a series of tokens that defines what the production looks like, and an optional action, enclosed in { and }. Productions are defined in terms of other rules and *terminals*. A terminal is a token that cannot be reduced any further— i.e., one that doesn't match any other rules.

For example, the grammar for Template::Simple defines this simple rule, chunk:

```
chunk:     TEXT            { $factory->textblock($_[1]) }

      |   statement ';'
```

The rule is chunk, and there are two productions: TEXT { ... } and statement ';' (the | indicates alternates). This means that the chunk rule is defined as either TEXT or whatever statement expands to (followed by a literal ;). The { ... } block attached to the TEXT subrule will be emitted literally into the grammar, and is assumed to be syntactically correct Perl code (it will become part of live code when the resulting grammar is actually used). The statement rule is assumed to have its own code block. The parser will pass the matching tokens to the statement as @_, with the parser as $_[0].

The parser will continue to reduce parsing until there are no expandable rules left in the input stream. At this point, the data is in its final parsed form.

The first thing to do is to modify the grammar, which means editing *parser/Parser.yp*. Because UNTIL will be based on WHILE, we can duplicate the WHILE implementation. The grammar defines WHILE as a type of loop; the definition for loop looks like this:

```
loop:      FOR loopvar ';'        { $_[0]->{ INFOR }++                   }

               block END          { $_[0]->{ INFOR }--;

                               $factory->foreach(@{$_[2]}, $_[5])  }

       |   atomexpr FOR loopvar    { $factory->foreach(@{$_[3]}, $_[1])  }

       |   WHILE expr ';'         { $_[0]->{ INWHILE }++                 }

               block END          { $_[0]->{ INWHILE }--;

                               $factory->while(@_[2, 5])          }

       |   atomexpr WHILE expr     { $factory->while(@_[3, 1])          }

;
```

We see that two types of loops are defined in the language—FOR and WHILE—and that each has a side-effect variant (e.g., atomexpr FOR loopvar).

The WHILE actions increment and decrement the INWHILE member of $_[0] (we'll see $_[0] in a moment); a quick search through the file reveals that INWHILE is used to implement the LAST and NEXT directives (these are *atomic directives*, which the grammar calls atomdir). If we are in a WHILE or FOR loop, these directives jump to the next or last occurrence of the LOOP label. Otherwise, they simply jump to the end of the current block:

```
atomdir:    GET expr      { $factory->get($_[2])                }
        ...
        |   LAST          { $_[0]->{ INFOR } || $_[0]->{ INWHILE }
                            ? 'last LOOP;'
                            : 'last;'                           }
        |   NEXT          { $_[0]->{ INFOR }
                            ? $factory->next( )
                            : ($_[0]->{ INWHILE }
                              ? 'next LOOP;'
                              : 'next;')                        }
    ...
    ;
```

So we'll need to keep INWHILE for UNTIL.

The action for WHILE calls $factory->while(@_[2, 5]). We know that $factory is a Template::Directive instance—this is what its while method looks like:

```
sub while {
    my ($class, $expr, $block) = @_;
    $block = pad($block, 2) if $PRETTY;

    return <<EOF;

# WHILE
do {
    my \$failsafe = $WHILE_MAX;
LOOP:
    while (--\$failsafe && ($expr)) {
$block
    }
    die "WHILE loop terminated (> $WHILE_MAX iterations)\\n"
        unless \$failsafe;
};
EOF
}
```

This production produces a series of five tokens: WHILE, the expansion of expr, ;, the expansion of the block, and END. These five elements, along with the parser object itself, are passed to the code block as @_. The factory's while is only interested in expr and block (which is reasonable because the other tokens are static strings):

```
|   WHILE expr ';'        { $_[0]->{ INWHILE }++                }
      block END           { $_[0]->{ INWHILE }--;
                            $factory->while(@_[2, 5])           }
```

$_[0] is the parser itself, and each token in the subrule becomes another element in the @_ array passed to the action subroutine. The parser invokes actions for subrules recursively, so $_[2], which is expr, has already been passed through the expr rule:

```
expr:    expr BINOP expr      { "$_[1] $_[2] $_[3]"            }
       | expr '/' expr        { "$_[1] $_[2] $_[3]"            }
       | expr '+' expr        { "$_[1] $_[2] $_[3]"            }
       | expr DIV expr        { "int($_[1] / $_[3])"           }
       | expr MOD expr        { "$_[1] % $_[3]"                }
       | expr CMPOP expr      { "$_[1] $CMPOP{ $_[2] } $_[3]"  }
       | expr CAT expr        { "$_[1]  . $_[3]"               }
       | expr AND expr        { "$_[1] && $_[3]"               }
       | expr OR expr         { "$_[1] || $_[3]"               }
       | NOT expr             { "! $_[2]"                      }
       | expr '?' expr ':' expr { "$_[1] ? $_[3] : $_[5]"      }
       | '(' assign ')'       { $factory->assign(@{$_[2]})     }
       | '(' expr ')'         { "($_[2])"                      }
       | term
;
```

So $_[2] contains a string of Perl code as generated by the expr rule when the action for WHILE gets to it. Most of these rules are defined in terms of themselves, except for term:

```
term:    lterm
       | sterm
;

lterm:   '[' list  ']'        { "[ $_[2] ]"                    }
       | '[' range ']'        { "[ $_[2] ]"                    }
       | '['       ']'        { "[ ]"                          }
       | '{' hash  '}'        { "{ $_[2]  }"                   }
;

sterm:   ident                { $factory->ident($_[1])         }
       | REF ident            { $factory->identref($_[2])      }
       | '"' quoted '"'       { $factory->quoted($_[2])        }
       | LITERAL
       | NUMBER
;
```

term eventually settles itself down to be a dotted identified (ident), a quoted string (quoted), a literal (LITERAL), or a number (NUMBER), or a list, hash, or range of those things.

Similarly, $_[5] contains a string of Perl code as determined by the block rule, which is one of the core building blocks of the grammar.

We want UNTIL to call a method with the same signature that WHILE calls, so we can duplicate the appropriate lines in the loop rule:

```
loop:    FOR loopvar ';'      { $_[0]->{ INFOR }++             }
           block END          { $_[0]->{ INFOR }--;
```

```
                              $factory->foreach(@{$_[2]}, $_[5])  }
        |   atomexpr FOR loopvar    { $factory->foreach(@{$_[3]}, $_[1])  }
        |   WHILE expr ';'          { $_[0]->{ INWHILE }++               }
              block END             { $_[0]->{ INWHILE }--;
                              $factory->while(@_[2, 5])          }
        |   atomexpr WHILE expr     { $factory->while(@_[3, 1])          }
        |   UNTIL expr ';'          { $_[0]->{ INWHILE }++               }
              block END             { $_[0]->{ INWHILE }--;
                              $factory->until(@_[2, 5])          }
        |   atomexpr UNTIL expr     { $factory->until(@_[3, 1])          }
;
```

This points to the currently nonexistent until method of Template::Directive; let's add it. Open *lib/Template/Directive.pm* and find the while method. Because UNTIL is logically equivalent to WHILE NOT, while is where we need to start looking, and in fact, we can duplicate it almost in its entirety:

```
sub until {
    my ($class, $expr, $block) = @_;
    $block = pad($block, 2) if $PRETTY;

    return <<EOF;

# UNTIL
do {
    my \$failsafe = $WHILE_MAX;
LOOP:
    while (--\$failsafe && !($expr)) {
$block
    }
    die "UNTIL loop terminated (> $WHILE_MAX iterations)\\n"
        unless \$failsafe;
};
EOF
}
```

We can copy the while method, and change the name of the subroutine and the name of the directive (in case anyone looks at the generated code), as well as modify the loop expression, from:

```
while (--\$failsafe && ($expr)) {
```

to:

```
while (--\$failsafe && !($expr)) {
```

And we're finished inside Directive.pm.

The last change is one of the most important—we need to tell the grammar that UNTIL is now a reserved word. In *parser/Grammar.pm.skel*, add UNTIL to the @RESERVED array:

```
@RESERVED = qw(
        GET CALL SET DEFAULT INSERT INCLUDE PROCESS WRAPPER BLOCK END
```

```
        USE PLUGIN FILTER MACRO PERL RAWPERL TO STEP AND OR NOT DIV MOD
        IF UNLESS ELSE ELSIF FOR NEXT WHILE SWITCH CASE META IN
        TRY THROW CATCH FINAL LAST RETURN STOP CLEAR VIEW DEBUG
        UNTIL
    );
```

Now, we're ready to re-create the grammar, and start testing!

```
$ ./yc
Compiling parser grammar (Parser.yp -> ../lib/Template/Grammar.pm)
```

When making any changes to the grammar, it is important to go back to the root of the distribution and run `make test`, to ensure that your changes didn't accidentally break anything else. It is also a good idea to write some new tests to both illustrate and test your new functionality.

8.5.3 Replacing the Default Grammar

It is possible to completely replace the existing grammar with something radically different. Generally, this requires not only the appropriate *Grammar.pm* file, but also a `Template::Directive`-style factory class that knows how to emit the code to implement your new language.

8.5.3.1 Template::Simple

The `Template::Simple` module implements a simple template language for use with the Template Toolkit.[4] It really is simple compared to the regular Template Toolkit language. It allows you to access variables and nothing else. No directives. No `INCLUDE`, no `IF`, no `FOREACH`. Nothing.

> [4] `Template::Simple` is available via anonymous CVS at *cvs -d:pserver:cvs@tt2.org:/Template-Simple co Template-Simple*.

However, all of the functionality for accessing variables is available. You can use scalars, lists, hash arrays, subroutines, and objects, and you can call virtual methods. There is no `SET` directive, either implicit or explicit, so you cannot update or create new variables.

```
# simple vars
[% name %] is an inhabitant of [% planet %].


# complex vars
[% friends.0 %] and [% friends.1 %] are his friends.


# virtual methods
[% friends.join(' and ') %] are still his friends.
```

You can emulate existing directives by binding subroutines to variables that make the appropriate calls to the `Template::Context` object:

```
my $ts = Template::Simple->new( );
my $tc = $ts->context( );


my $vars = {
    name    => 'Arthur Dent',
    planet  => 'Earth',
    friends => [ 'Ford Prefect', 'Slartibartfast' ],
```

```
    include => sub { $tc->include(@_) },
};
```

Then you access the subroutine via the include variable, passing the template name and local variables as arguments:

```
[% include( 'person/summary',
            name   = 'Slartibartfast'
            planet = 'Magrethea' )
%]
```

The Template::Simple module is a very thin wrapper around the Template module. All it does is set the GRAMMAR configuration option to Template::Simple::Grammar. Most of the other Template Toolkit options can be passed to the Template::Simple constructor. However, any options that relate to directives that are no longer implemented will be ignored (e.g., PLUGINS, FILTERS, etc.).

8.5.3.2 The Template::Simple grammar

The heart of Template::Simple is the grammar, which is built from *Parser.yp*. Template::Simple's full grammar is relatively simple, and consists of a small set of core tokens (TEXT, IDENT, COMMA, LITERAL, NUMBER, DOT, ASSIGN) and a few more complex rules built up from these tokens.

Example 8-16 is the complete Template::Simple grammar. To read the grammar, start at the top—the first rule is the implicit "start" rule, from which the parser commences. Thus, the main rule in this grammar is template. $factory is the Perl factory, Template::Directive by default, that is used to generate Perl code that will eventually be transformed into the Template::Document instance (refer to Chapter 7 for all the details).

Example 8-16. Template::Simple grammar

```
%%

template:   block           { $factory->template($_[1]) }
;

block:      chunks          { $factory->block($_[1]) }
        |   /* NULL */      { $factory->block( )     }
;

chunks:     chunks chunk    { push(@{$_[1]}, $_[2])
                                if defined $_[2];
                              $_[1]
                            }
        |   chunk           { defined $_[1]
                                ? [ $_[1] ]
                                : [ ]
                            }
;

chunk:      TEXT            { $factory->textblock($_[1]) }
        |   statement ';'
```

```
;

statement:  term            { $factory->get($_[1]) }
        |   /* empty */
;

term:       ident           { $factory->ident($_[1])  }
        |   '"' quoted '"'  { $factory->quoted($_[2]) }
        |   LITERAL
        |   NUMBER
;

ident:      ident DOT node  { push(@{$_[1]}, @{$_[3]});
                              $_[1]
                            }
        |   ident DOT NUMBER  { push( @{ $_[1] },
                                    map { ($_, 0) }
                                    split(/\./, $_[3]) );
                              $_[1]
                            }
        |   node
;

node:       item            { [ $_[1], 0 ]                    }
        |   item '(' args ')' { [ $_[1], $factory->args($_[3]) ] }
;

item:       IDENT           { "'$_[1]'"                       }
        |   '${' term '}'   { $_[2]                           }
        |   '$' IDENT       { $factory->ident(["'$_[2]'", 0])  }
;

args:       args term       { push(@{$_[1]}, $_[2]);
                              $_[1]
                            }
        |   args param      { push(@{$_[1]->[0]}, $_[2]);
                              $_[1]
                            }
        |   args COMMA      { $_[1]                           }
```

```
        |  /* init */       { [ [ ] ]                   }
;

quoted:     quoted quotable   { push(@{$_[1]}, $_[2])
                                    if defined $_[2];
                                  $_[1]
                                }
        |  /* NULL */       { [ ] }
;

quotable:   ident            { $factory->ident($_[1]) }
        |  TEXT             { $factory->text($_[1])  }
        |  ';'              { undef                  }
;

param:      LITERAL ASSIGN term  { "$_[1] => $_[3]" }
        |  item ASSIGN term     { "$_[1] => $_[3]" }
;

%%
```

Chapter 8. Extending the Template Toolkit

Most of the customization you are likely to perform will fall under one of two categories: creating new frontends and writing filters and plugins. However, some things cannot be handled with a new frontend or by writing a plugin, such as modifying how the provider finds templates to process or limiting access to certain plugins. Luckily, the Template Toolkit makes it easy to replace or extend any of the core components; its modular design makes replacing individual components simple. Chapter 7 gives public API details for each component.

9.1 Using the DBI Plugin

The DBI plugin provides direct access to the Perl DBI. The DBI provides a generic way of connecting to a database, and is the standard for using databases within Perl. The DBI plugin is a thin wrapper around DBI, with some Template Toolkit-specific modifications.

9.1.1 Simple Database Access with the DBI Plugin

In our first example of using the DBI plugin, we'll pull some data out of a MySQL database that contains details of a company's product range. Example 9-1 shows the template that we will use.

Example 9-1. Listing products

```
[% USE DBI('dbi:mysql:products', 'username', 'password') -%]
  Code  Name        Price   Stock
  [% FOREACH product = DBI.query('select ProductID, Name, Price, Stock from products') -%]
  [% product.ProductID | format('%05d') %] [% product.Name -%]
  $[% product.Price | format('%6.2f') %] [% product.Stock | format('%5d') %]
[% END -%]
```

The first thing to notice is the USE DBI directive, which is used to load the DBI plugin and connect to the database. The USE DBI directive takes a number of arguments. In this case, we pass it a string that identifies the data source that we want to connect to, together with the username and password that are required to make the connection.

The exact syntax of the data source identifier will vary depending on the type of the data source, but it will always start with the string dbi followed by a colon and the name of the connection type. In this case, as we are connecting to a MySQL database, we give it the string mysql followed by the name of the database that we wish to connect to (products). This usage assumes that the database is on the same server as the template processor. If it is on a different server, we can define that here by adding the hostname to the end of the data source identifier—for example, *dbi:mysql:products:db.company.com* would attempt to connect to the products database on the server *db.company.com*.

Having connected to the database, we can start to execute queries to access the required data. In this example, we will use the query function, which executes an SQL select query and returns the data a row at a time in a hash. The keys of the hash are the names of the columns selected. We assign each row in turn to the variable product, and can use that variable to access various parts of the returned row. Here are the results of processing the template in Example 9-1:

```
Code  Name         Price   Stock
00050 Basic Widget $ 49.99  2500
00051 Cheap Widget $ 29.99  5000
00101 Super Widget $ 99.99  1000
00102 Ultra Widget $149.99   500
```

Example 9-2 adds another level of complexity. Each product comes from a supplier; in this second report, we want to produce a list of each supplier followed by a sublist of the products that we get from the supplier.

Example 9-2. Listing products by supplier

```
[% USE DBI('dbi:mysql:products', 'username', 'password')
  suppliers = DBI.prepare('select SupplierID, Name from suppliers')
  products = DBI.prepare('select ProductID, Name, Price, Stock
                  from   products
                  where  SupplierID = ?')
```

```
-%]

[% FOREACH supplier = suppliers.execute -%]

[% supplier.Name %]

  [% FOREACH product = products.execute(supplier.SupplierID) -%]

  [% product.Name %]

  [% END %]

[% END -%]
```

For this, we will need two SQL queries to be active—one to list the suppliers and one to list the products. Additionally, the product query will need to take a parameter so that it returns only the products from the current supplier. To do this, we use the `prepare` method to precompile the two queries. Notice that the product query contains a clause, `where SupplierID = ?`. The `?` character marks a placeholder that will be filled in when we execute the query.

We then execute the suppliers query and process each returned row. As part of that processing, we execute the products query. The call to `products.execute` is passed the `SupplierID` for the current supplier record. Any arguments to `execute` are used as values to fill in the placeholders in the original SQL.

Here are the results of processing the template in Example 9-2:

```
Costcutter Widgets Inc.

    Basic Widget

    Cheap Widget



Quality Widgets Inc.

    Super Widget

    Ultra Widget
```

9.1.2 A More Complex Example: Web Access Logs

Having taken a look at a couple of simple templates that use the DBI plugin, it's now time to look at a more complex example. For this section, we will be using a table generated from a web server's access log (in Common Log Format). For simplicity, our examples will use `DBD::SQLite`—SQLite is a small, fast, embeddable, typeless RDBMS that implements most of SQL92, and includes advanced features such as transactions, triggers, and views. See http://www.hwaci.com/sw/sqlite/ for details about SQLite, and http://search.cpan.org/dist/DBD-SQLite/ for details about `DBD::SQLite`.

We will be using the following table definition:

access_log.sql

```
CREATE TABLE access_log (

    id INTEGER PRIMARY KEY,

    hostaddr VARCHAR,

    hostname VARCHAR,

    logname VARCHAR,

    req_time VARCHAR,

    request VARCHAR,

    uri VARCHAR,

    method VARCHAR,

    http_version VARCHAR,
```

```
    status VARCHAR,
    bytes_sent VARCHAR
);
```

The hostname field is generated by doing a DNS lookup of the hostaddr field (if it doesn't look like an IP address), and the uri, method, and http_version fields are parsed from the request field.

Example 9-3 shows the script that we used to get our file-based data into the database.

Example 9-3. Parsing log file entries

```
#!/usr/bin/perl -w


use strict;
$|++;


use DBI;
use Net::Nslookup qw(nslookup);
use Regexp::Common qw(net);


my $dsn = shift;
my $dbh = DBI->connect($dsn)
    || die "Can't connect to '$dsn': $DBI::err\n";


my $count = 0;
my $INSERT =<<'SQL';
    INSERT INTO access_log
        (hostaddr, hostname, logname, req_time, request,
         uri, method, http_version, status, bytes_sent)
    VALUES
        (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
SQL


while (<>) {
    my ($hostaddr, $logname, $remote_user, $req_time,
        $request, $status, $bytes_sent) = /^(\S+)     # host address
                                             \s+
                                            (\S+)     # remote logname
                                             \s+
                                            (\S+)     # username
                                             \s+
                                           \[(.+?)\]   # request time
                                             \s+
```

```
                      "(.+?)"    # request
                        \s+
                      ([\d-]+)   # status
                        \s+
                      ([\d-]+)   # bytes sent
                     /x;

    next unless $hostaddr;

    my ($method, $uri, $http_version) = split /\s+/, $request;

    my $hostname;
    if ($hostaddr =~ /$RE{net}{IPv4}/o) {
        $hostname = nslookup(host => $hostaddr, type => 'PTR');
    }
    else {
        $hostname = $hostaddr;
    }

    $dbh->do($INSERT, undef, $hostaddr, $hostname, $logname,
             $req_time, $request, $uri, $method, $http_version,
             $status, $bytes_sent)
        or warn "Error inserting line $.: " . $dbh->errstr;

    $count++;
    print '.' if (($count % 10) =  = 0);
    print "\n" if (($count % 700) =  = 0);
}

$dbh->commit;   # commit any outstanding lines
$dbh->disconnect;
```

Run the script with the DSN as the first argument, and an *access_log* on standard input:

```
$ logparse.pl dbi:SQLite:dbname=access_log < /home/www/logs/access_log
```

The script emits a dot character (.) for each 10 lines it inserts, breaking the output lines at 70 characters, mainly as a visual indication that it is still running (inserting thousands of entries can take a long time, after all).

With that out of the way, we can start using the DBI plugin. To connect to a database, pass the DSN to the USE DBI line in the template:

```
[% USE DBI('dbi:SQLite:dbname=access_log') %]
```

Or use the connect( ) method on a DBI object:

```
[% USE DBI %]

[% DBI.connect('dbi:SQLite:dbname=access_log') %]
```

Once we have a DBI object, we can use it to issue SQL statements:

```
[% log_entries = DBI.query('SELECT * FROM access_log') %]
```

The query method takes an SQL statement, which it issues against the underlying database, and returns an iterator that we can use to manipulate the data (see Example 9-4).

Example 9-4. Counting visitors

```
[% # Get a count of visits per address

  visitors = { };

  FOREACH log_entry IN log_entries;

    visitors.${log_entry.hostaddr} =

      visitors.${log_entry.hostaddr} + 1;

  END



  MACRO times(count)

    # "1 time" or "2 times"

    IF count =  = 1;

      "$count time";

    ELSE;

      "$count times";

    END

-%]

[% FOREACH visitor IN visitors.keys %]

  [% visitor %] visited [% times(visitors.$visitor) -%]

[% END %]
```

The simple template in Example 9-4 might give us something like the following:

```
134.174.141.2 visited 4 times

128.103.1.1 visited 1 time

206.33.106.134 visited 2 times

4.2.2.1 visited 3 times
```

Once we have the data, we can use one of the graph-generating plugins—for example, GD.Graph.pie, to generate a nice graph (see Example 9-5).

Example 9-5. Generating graphs

```
[%  USE graph = GD.Graph.pie(400, 300);

  FILTER null;

    data = [

      [ ]     # Array of addresses

      [ ]     # Array of visits
```

```
   ];

   FOREACH visitor IN visitors.keys;
     data.0.push(visitor);
     data.1.push(visitors.$visitor);
   END;

   dclrs = [ 'green' 'blue' 'red' 'cyan' ];
   graph.set(
     title   = 'Visits per address'
     transparent = 0,

     cycle_clrs  = 1
     dclrs       = dclrs
   );

   # plot data as a PNG, and send it to stdout
   # (recall the argument to the stdout filter
   # indicates that bindmode should be set).
   graph.plot(data).png | stdout(1);
 END;
-%]
```

Because the DBI plugin passes through to the underlying RDBMS, we can use any functions or stored procedures that database offers, as shown in Example 9-6.

Example 9-6. Using RDBMS-specific functions

```
[% query = DBI.query('SELECT sum(bytes_sent) as bytes_sent,
                     hostaddr FROM access_log group by
                     hostaddr'); %]
[% FOREACH hb = query %]
  We sent [% hb.bytes_sent %] bytes to [% hb.hostaddr %].
[% END %]
```

The query method returns an iterator that is similar, though not identical, to what is created within a FOREACH loop (the loop variable). This means that we have access to some of loop's methods, such as size, index, and max (see Example 9-7).

Example 9-7. Counting results

```
[% log_entries = DBI.query('SELECT hostaddr
                            FROM access_log
                            GROUP BY hostaddr');
-%]
There are [% log_entries.size %] unique addresses in the log.
```

Business folk like to have reports in CSV format so that they can manipulate the data in a spreadsheet program such as Excel or Gnumeric; producing a CSV file is pretty straightforward, as shown in Example 9-8:

Example 9-8. Producing a report as a CSV file

```
[% log_entries = DBI.query('SELECT * FROM access_log');
  FOREACH entry IN log_entries;
    FOREACH field IN entry.keys;
      field = entry.$field;
      field.match('[, ]') ? "\"$field\"" : field;
      "," UNLESS loop.last;
    END;
    "\n";
  END
-%]
```

If the field contains a comma (,) or a space, we quote it, using double quotes. Otherwise, it can be emitted as is.

Generating the report in XML format is similar, as shown in Example 9-9.

Example 9-9. Producing a report as XML

```
<?xml version="1.0" standalone="yes"?>
<access-log>
  [% log_entries = DBI.query('SELECT * FROM access_log') %]
  [% FOREACH entry IN log_entries %]
    <log-entry>
      [% FOREACH field IN entry.keys %]
        <[% field %]>[% entry.$field | html %]</[% field %]>
      [% END %]
    </log-entry>
  [% END %]
</access-log>
```

9.2 Using Class::DBI

Class::DBI is a convenient, easy-to-use database abstraction layer. It automates all the repetitive code that accompanies every database wrapper—accessors, mutators, constructors, search interfaces—and enforces efficient use of the DBI as well.

When using Class::DBI-based objects with the Template Toolkit, most of the work takes place in the Perl module that implements the class; once that part is written, templates can treat the object like any other variable. With Class::DBI taking care of most of the details of the database, fully functional modules can be implemented with very little actual code.

To illustrate using Class::DBI with the Template Toolkit, here is a very simple realestate application, based around a few tables:

```
CREATE TABLE listing (
    listing INTEGER PRIMARY KEY,
    location INTEGER,
    realtor INTEGER,
    price INTEGER,
    rooms INTEGER,
    bedrooms INTEGER,
    baths INTEGER,
    body VARCHAR
);

CREATE TABLE realtor (
    realtor INTEGER PRIMARY KEY,
    name VARCHAR,
    phone VARCHAR,
    url VARCHAR
);

CREATE TABLE location (
    location INTEGER PRIMARY KEY,
    city VARCHAR,
    state VARCHAR,
    postalcode VARCHAR
);
```

While this schema leaves obvious room for improvements, it will suffice for our needs. To illustrate how simple it is to integrate Class::DBI and the Template Toolkit, we'll start with a Class::DBI base class, as suggested in Example 9-10.

Example 9-10. Class::DBI

```
package TTBook::RealEstate::DBI;


use strict;

use vars qw($VERSION);

use base qw(Class::DBI);


TTBook::RealEstate::DBI->set_db('Main', 'dbi:SQLite:dbname=realestate.db');
```

This very simple module will be used as the base class by the other modules in our real-estate application. We set the main DSN here (the Main table)—it will be inherited by our subclasses.

The modules that sit on top of the listing, realtor, and location tables are almost as simple; they just need to declare the table upon which they sit, and list the columns in that table:

```
package TTBook::RealEstate::Listing;


use strict;

use base qw(TTBook::RealEstate::DBI);


# DB Table

TTBook::RealEstate::Listing->table('listing');


# Column groups

TTBook::RealEstate::Listing->columns(All =>

    qw(listing rooms body price bedrooms baths location realtor));


# Relationships with other objects

TTBook::RealEstate::Listing->has_a(location => 'TTBook::RealEstate::Location');

TTBook::RealEstate::Listing->has_a(realtor => 'TTBook::RealEstate::Realtor');
```

The TTBook::RealEstate::Listing table has relationships with data in other tables, and we indicate this with the has_a method. The TTBook::RealEstate::Realtor and TTBook::RealEstate::Location tables are very simple, and as a consequence can be represented very simply:

```
package TTBook::RealEstate::Realtor;


use strict;

use base qw(TTBook::RealEstate::DBI);


# DB Table

TTBook::RealEstate::Realtor->table('realtor');


# Columns

TTBook::RealEstate::Realtor->columns(All => qw(realtor name phone));
```

```
package TTBook::RealEstate::Location;


use strict;

use base qw(TTBook::RealEstate::DBI);


# DB Table

TTBook::RealEstate::Location->table('location');


# Columns

TTBook::RealEstate::Location->columns(All => qw(location city state postalcode));
```

Notice that these modules consist almost entirely of configuration, and not code. Such is the power of Class::DBI—only extraordinary situations require special-purpose code.

Using our new classes is simple. The simple CGI script in Example 9-11 either processes *listing.tt2* (if invoked with a listing_id parameter) or presents a search form, which will presumably call itself with a listing_id parameter.

Example 9-11. listing.cgi

```
#!/usr/bin/perl


use strict;

use warnings;


use CGI;

use Template;

use TTBook::RealEstate::Listing;


my $q = CGI->new( );

my $listing_id = $q->param('listing_id');

my $template = $listing_id ? 'listing.tt2' : 'form.tt2';


my $tt = Template->new( ) || die Template->error;


my $listing = TTBook::RealEstate::Listing->retrieve($listing_id);

$template = 'notfound.tt2' unless $listing;


my $vars = {

    'listing' => $listing,
```

```
};

print $q->header('text/html');

$tt->process($template, $vars)
    || die $tt->error;
```

Within *listing.tt2*, we can access methods of the listing variable (which is an instance of our Class::DBI subclass, TTBook::RealEstate::Listing) directly, as shown in Example 9-12.

Example 9-12. listing.tt2

```
[% USE wrap;
  realtor = listing.realtor;
  location = listing.location;
-%]
<h1>Look at this beautiful home in [% location.city %]!</h1>

[% PROCESS summary.tt2
  price       = listing.price
  rooms       = listing.rooms
  bedrooms    = listing.bedrooms
  baths       = listing.baths
%]

<p>
  [% listing.body | wrap %]
</p>

<p>
  For more information, contact [% realtor.name %] at
  [% realtor.phone %].
</p>
```

The *summary.tt2* template shown in Example 9-13 creates a simple table of attributes (price and number of rooms, bedrooms, and bathrooms). We can use the Template::Plugin::Number::Format plugin from CPAN,[1] to format the price nicely.

[1] You can find this plugin at http://search.cpan.org/dist/Template-Plugin-Number-Format/.

Example 9-13. summary.tt2

```
[% USE Number.Format %]

<table>

  <tr>

    <th>Price</th><td>[% price | format_price(0) %]</td>

  </tr>

  <tr>

    <th>Rooms</th><td>[% rooms %]</td>

  </tr>

  <tr>

    <th>Bedrooms</th><td>[% bedrooms %]</td>

  </tr>

  <tr>

    <th>Baths</th><td>[% baths %]</td>

  </tr>

</table>
```

The format_price filter takes a precision, which in this case we will set to 0—we probably don't need to see fractions of a quid when dealing with house prices.

It so happens that we can simplify our implementations even more. Because we are using SQLite for a database, our TTBook::RealEstate::DBI base class can subclass Class::DBI::SQLite instead of Class::DBI. Class::DBI::SQLite knows how to query the underlying SQLite database to get the schema for the appropriate tables automatically:

```
package TTBook::RealEstate::DBI;



use strict;

use vars qw($VERSION);

use base qw(Class::DBI::SQLite);



TTBook::RealEstate::DBI->set_db('Main', 'dbi:SQLite:dbname=realestate.db');
```

Using Class::DBI::SQLite enables us to simplify all of our subclasses, using the set_up_table method.[2] For example:

[2] This feature isn't specific to Class::DBI::SQLite; there are also versions for Oracle, Postgres, and MySQL.

```
package TTBook::RealEstate::Listing;



use strict;

use base qw(TTBook::RealEstate::DBI);



TTBook::RealEstate::Listing->set_up_table('listing');



# Relationships with other objects

TTBook::RealEstate::Listing->has_a(location => 'TTBook::RealEstate::Location');

TTBook::RealEstate::Listing->has_a(realtor => 'TTBook::RealEstate::Realtor');
```

9.3 Using DBIx::Table2Hash

The DBIx::Table2Hash module provides a simple way to turn a database table into a hash, turning SQL statements into simple lookups in a prepopulated table. DBIx::Table2Hash has methods to make this data available in a nested form as well as in a one-dimensional lookup table. While it doesn't allow for updates, it provides fast, convenient access to the data of a static table, such as a table containing postal codes and the cities to which they map. For this example, assume a simple table that looks like this (using SQLite again):

```
CREATE TABLE postal_code (
    code VARCHAR PRIMARY KEY,
    city VARCHAR
);
```

Using DBIx::Table2Hash, we can get a hash of our access_log data from within Perl like so:

```
my %args = ( dbh          => $dbh,
             table_name   => 'postal_codes',
             key_column   => 'city',
             value_column => 'code' );


my $t2h = DBIx::Table2Hash->new(%args)
my $data = $t2h->select;
```

Let's see how we can utilize this data. DBIx::Table2Hash expects to be passed a hash of items, including a connected database handle. Here's an example, adapted from the DBIx::Table2Hash documentation:

```
[% args = { dbh          = dbh
            table_name   = 'postal_code'
            key_column   = 'city'
            value_column = 'code' };


   USE t2h = Table2Hash(args);
   codes = t2h.select %]
```

The Template Toolkit will pass those hash values as a hashref, so we'll need to wrap this in a plugin.

Once we USE the plugin, we can call select, select_hashref, or select_tree to get our data. select returns a hash reference in which each element is a key_column => value_column pair (key_column and value_column are specified in arguments given to the constructor).

```
The postal code for Plymouth is [% codes.Plymouth %].
```

Keys with spaces in their names must be used indirectly:

```
[% ey = "East Yarmouth" -%]
The postal code for East Yarmouth is [% codes.$ey %].
```

select_hashref returns a hash of hashrefs, keyed by key_column:

```
[% codes = Table2Hash.select_hashref %]
[% FOREACH city = codes.keys %]
    [% city %] has postal code [% codes.$city.code %].
[% END %]
```

We've been ignoring where the dbh in this example comes from. There are several options here; for example, we could add code to TTBook::Template::Plugin::Table2Hash to accommodate a missing dbh parameter. Even simpler would be to use the DBI plugin:

```
[% USE DBI('dbi:SQLite:dbname=postal_codes.db');
   USE Table2Hash(dbh          = DBI.dbh
                  table_name   = 'postal_code'
                  key_column   = 'city'
                  value_column = 'code');
   codes = Table2Hash.select;
%]
```

The complete TTBook::Template::Plugin::Table2Hash: is shown in Example 9-14.

Example 9-14. TTBook::Template::Plugin::Table2Hash

```
package TTBook::Template::Plugin::Table2Hash;

use strict;
use vars qw($VERSION);
use base qw(Template::Plugin);

use DBIx::Table2Hash;

$VERSION = 1.00;

sub new {
    my ($class, $context, $args) = @_;
    my $dbix = DBIx::Table2Hash->new(%$args);

    return bless {
        _CONTEXT => $context,
        _T2H => $dbix,
        _ARGS => $args,
    } => $class;
}

sub select {
    my ($self, $args) = shift;
    return $self->{_T2H}->select(%$args);
}

sub select_hashref {
    my ($self, $args) = shift;
```

```
    return $self->{_T2H}->select_hashref(%$args);

}


sub select_tree {

    my ($self, $args) = shift;

    return $self->{_T2H}->select_tree(%$args);

}



1;
```

9.3.1 Writing Your Own Database Abstraction Layer

When all else fails, you can always write your own abstraction layer. Sometimes, this is the only alternative that makes sense. When dealing with content developers who have no understanding of SQL, it can be easier to provide them with a foolproof method of retrieving dynamic data from a database. Creating an abstraction layer to handle query generation also means that you can change the underlying database—for example, from SQLite to Postgres—without anyone having to know, and without any of the templates that access it having to be changed.

One of the most basic elements of a database abstraction layer is figuring out how to turn a collection of data into SQL. Luckily, several modules are on CPAN that do exactly that. My favorite is Nathan Wiger's SQL::Abstract (http://search.cpan.org/dist/SQL-Abstract/). This powerful module takes search critera as a hash, and transforms it into a WHERE clause.

We can create a search interface for the access_log database we defined earlier. Recall our access_log table:

```
CREATE TABLE access_log (

    id INTEGER PRIMARY KEY,

    hostaddr VARCHAR,

    hostname VARCHAR,

    logname VARCHAR,

    req_time VARCHAR,

    request VARCHAR,

    uri VARCHAR,

    method VARCHAR,

    http_version VARCHAR,

    status VARCHAR,

    bytes_sent VARCHAR

);
```

The key to creating a useable database query module is making it simple to use—you can't get much more powerful than DBI, but it is unintuitive for people who don't already know both SQL and Perl. SQL::Abstract is a small, powerful module with methods designed to generate SQL from a hash of parameters, such as those that might come in via a CGI form submission.

Ideally, we'll be able to provide a robust search interface, using only a few simple constructs in the template (see Example 9-15).

Example 9-15. Searching with the AccessLogSearch plugin

```
[%  # Our search plugin is called AccessLogSearch

  USE als = AccessLogSearch('dbi:SQLite:dbname=access_log');
```

```
  search.terms = {
    uri    = '*/index.htm?'
    status = 404,
    };

    fields = [ 'hostname' 'uri' 'status' ];

    results = als.query(fields, search.terms);
%]

Found [% results.size %] results for your search terms!
[% FOREACH result IN results %]
  ...
[% END %]
```

Given these search terms, results would contain all requests for *index.htm* or *index.html* pages that generated a status of 404 (Not Found). Note the * and ? wildcards, which make globbing simpler for users who might not know that % and _ are the SQL wildcard characters. More importantly, it abstracts the implementation; if we change the underlying data source to a different database, or to something other than database, the user-facing interface isn't coupled to an irrelevant wildcard convention.

We begin by subclassing the DBI plugin because it does almost all of what we want. Specifically, it handles connecting to the database and creating an efficient iterator object so that we don't have to read all of our results into memory.

```
package TTBook::Template::Plugin::AccessLogSearch;

use strict;
use vars qw($VERSION $DEBUG);
use base qw(Template::Plugin::DBI);

$VERSION = 1.00;
$DEBUG = 0 unless defined $DEBUG;

use SQL::Abstract;
use Template::Plugin::DBI;
```

The new method defers to the DBI plugin's new method, but also needs to create a SQL::Abstract instance:

```
sub new {
    my $class = shift;
    my $self = $class->SUPER::new(@_);
    my $sql = SQL::Abstract->new;
```

```
$self->{ _SQL } = $sql;


    return $self;
}
```

The AccessLogSearch plugin keeps a similar interface to the DBI plugin, but adds a little syntactic sugar to the query method:

```
[% # How many hits from Harvard's medical library this month?
   results = als.query('hostname' 'status' 'uri'
                       hostaddr = '134.174.151.*'
                       req_time = '%Aug%2003%);
%]
```

The new query method handles these criteria easily: name => value pairs are search parameters, and any other values are the fields to be selected:

```
sub query {
    my ($self, @fields) = @_;
    my $terms = ref($fields[-1]) eq 'HASH' ? pop(@fields) : {  };
    my ($sql, @bind, $sth, $result, @results);
```

We can specify the fields that we want back, such as hostname, uri, and status, but if fields is empty, we use *, which means to select all fields. If the user passes in an array from the template, it will come to our method as an arrayref, so we dereference it here.

```
@fields = ('*') unless @fields;
@fields = @{$fields[0]} if ref($fields[0]) eq 'ARRAY';


$self->expand($terms);
($sql, @bind) = $self->{ _SQL }->select('access_log', \@fields, $terms);
```

If we are in $DEBUG mode—for example, during development—we emit the compiled SQL statement to the standard error stream, via the debug method (inherited from Template::Base, by way of Template::Plugin::DBI). Because SQL::Abstract generates SQL with placeholders, we need to fill them into the debugging string:

```
if ($DEBUG) {
    my @local_bind = @bind;
   (my $local_sql = $sql) =~ s/\?/"" . shift(@local_bind) . ""/eg;
    $self->debug("Generated SQL: '$local_sql'")
}
```

Now that we've generated the SQL, we can pass that to the DBI plugin's query method, which does the right thing—executes the query and returns a reference to an Iterator:

```
return $self->SUPER::query($sql, @bind);
    }
```

The expand method is responsible for turning * and ? into the SQL wildcards % and _ as shown here:

```
sub expand {
    my ($self, $terms) = @_;
```

```
    for my $term (keys %$terms) {
        my $like = 0;

        for ($terms->{$term}) {
            s/*/%/g && $like++;
            s/\?/_/g && $like++;
        }

        $terms->{$term} = $like ? { 'LIKE' => $terms->{$term} }
                                : { '='    => $terms->{$term} }
    }

    return $terms;
}
```

SQL::Abstract also knows how to deal with wildcard SQL, as long as we tell it to emit LIKE instead of =, so we count occurrences of the wildcard characters and use that to determine the appropriate test to use.

The complete TTBook::Template::Plugin::AccessLogSearch is shown in Example 9-16.

Example 9-16. TTBook::Template::Plugin::AccessLogSearch

```
package TTBook::Template::Plugin::AccessLogSearch;

use strict;
use vars qw($VERSION $DEBUG);
use base qw(Template::Plugin::DBI);

$VERSION = 1.00;
$DEBUG = 0 unless defined $DEBUG;

use SQL::Abstract;
use Template::Plugin::DBI;

# ----------------------------------------------------------------------
# new($context, @args)
#
# Pass @args directly to the superclass.
# ----------------------------------------------------------------------
sub new {
    my $class = shift;
    my $self = $class->SUPER::new(@_);
```

```
    my $sql = SQL::Abstract->new;

    $self->{ _SQL } = $sql;

    return $self;
}

# ----------------------------------------------------------------------
# query(@fields, \%terms)
# ----------------------------------------------------------------------
sub query {
    my ($self, @fields) = @_;
    my $terms = ref($fields[-1]) eq 'HASH' ? pop(@fields) : {  };
    my ($sql, @bind, $sth, $result, @results);

    @fields = ('*') unless @fields;
    @fields = @{$fields[0]} if ref($fields[0]) eq 'ARRAY';

    $self->expand($terms);
    ($sql, @bind) = $self->{ _SQL }->select('access_log', \@fields, $terms);

    if ($DEBUG) {
        my @local_bind = @bind;
       (my $local_sql = $sql) =~ s/\?/"'" . shift(@local_bind) . "'"/eg;
        $self->debug("Generated SQL: '$local_sql'")
    }

    return $self->SUPER::query($sql, @bind);
}

# ----------------------------------------------------------------------
# expand(\%terms)
#
# Expand * and ? wildcards into SQL wildcards % and _.  Expects a
# reference to a hash, and operates on each value.  If a value is
# expanded, use LIKE instead of =.
# ----------------------------------------------------------------------
sub expand {
    my ($self, $terms) = @_;
```

```
    for my $term (keys %$terms) {
        my $like = 0;

        for ($terms->{$term}) {
            s/*/%/g && $like++;
            s/\?/_/g && $like++;
        }

        $terms->{$term} = $like ? { 'LIKE' => $terms->{$term} }
                                : { '='    => $terms->{$term} }
    }

    return $terms;
}

1;
```

Chapter 9. Accessing Databases

In many ways, the integration of a templating system and a database is natural. From e-commerce sites to Microsoft Word's MailMerge, database-backed template processing is very common. Indeed, this integration is one of the primary selling points of many systems, such as ASP and PHP.

You can integrate the Template Toolkit with a database in several ways. The most straightforward way is to simply use the DBI plugin. The DBI plugin is part of the standard Template Toolkit distribution, and provides a template-facing way to utilize Perl's DBI module (see *Programming the Perl DBI: Database Programming with Perl*, by Alligator Descartes and Tim Bunce (O'Reilly), for details about the DBI).

In addition to DBI, several database-related modules are on CPAN, such as Class::DBI and DBIx::SearchBuilder, that can be used to abstract the database layer out of code. Using these modules from within the Template Toolkit is the same as using them in Perl programs.

Writing your own abstraction layer is always an option as well. Many people like to keep SQL out of application code, for the same reasons that people prefer to keep business logic out of presentation templates; this is the primary purpose of a database abstraction layer. Many SQL-related helper modules are on CPAN, such as SQL::Abstract, SQL::OrderBy, SQL::QueryBuilder::Simple, and SQL::AnchoredWildcards, that can be used to help provide a non-SQL interface to a database.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *Perl Template Toolkit* is a badger. The badger (*Meles meles*) is the largest member of the weasel family, and is found extensively throughout the northern hemisphere. Badgers are the best diggers of all carnivores, and can be found in the sandy or clay soils of dry open fields, parklands, and pastures where there are few large tree roots in their way as they digs.

Badgers are completely covered in gray or black fur except for on the head, where white stripes (or badges) run from the nose to the shoulders. Adult males can weigh as much as 26 pounds in autumn as, to prepare for winter, badgers tend to consume large amounts of food. Although they do not hibernate, badgers sleep in their burrows during winter and live off of their body fat.

The bones and muscles are large for an animal of the badger's size. The forefeet are armed with long, wide claws for digging. The claws on the hind legs are short and shovel-like for scooping away dirt. The flattened body easily slips into small burrows. A badger can dig itself into a hole in a few minutes.

Badgers are nocturnal, foraging for food at night. They eat everything from earthworms, insects, fruits, and berries to squirrels, mice, rabbits, and snakes. If attacked by a person or coyote--its main enemies--the badger acts quickly. The badger digs itself into a hole, throwing dirt and dust into its attacker's face. The badger turns with its powerful claws and terrible bite to face its enemy. The badger then starts to fill the hole in front of it with loose dirt to hide itself. Coyotes usually leave to find less dangerous prey. Few other animals will attack a badger.

Often hunted for their pelts, many countries now have laws protecting badgers. Badgers have been known to live for up to 14 years in the wild, but are likely to die or be killed before they reach this age.

Darren Kelly was the production editor, Audrey Doyle was the copyeditor, and Mary Brady was the proofreader for *Perl Template Toolkit*. Mary Anne Weeks Mayo and Colleen Gorman provided quality control. Tom Dinse wrote the index. Jamie Peppard, Matt Hutchinson, and Mary Agner provided production assistance.

Emma Colby designed the cover of this book, based on a series design by Edie Freedman. The cover image is an original engraving from the 19th century. Emma produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted by Joe Wizda to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Darren Kelly.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Ellie Cutler) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, Ellie Cutler, and Jeff Liggett.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Audience

This book should be useful to anyone building and maintaining web sites or other complex content systems. No prior knowledge of Perl, the Template Toolkit, or HTML is required to apply the basic techniques taught in this book. Some of the more advanced topics require some degree of familiarity with the Perl programming language. Readers who understand the basic language constructs and idioms of Perl and who already know how to install and use Perl modules will have no trouble integrating the Template Toolkit into their existing or new projects. Some chapters talk about more specific application areas: HTML, web programming, XML, and SQL, for example. Experience in these areas will make the benefits of the Template Toolkit more readily apparent, but isn't required.

About this Book

This book is divided into 12 chapters and 1 appendix.

Chapter 1, *Getting Started with the Template Toolkit*, provides an introduction to the concepts of template processing in general and to the Template Toolkit in particular. It also covers how to install the Template Toolkit on your system and gives a brief tutorial on its use so that you can check that installation is successful. In case it isn't, the chapter also includes pointers to other sources of information on the Template Toolkit.

Chapter 2, *Building a Complete Web Site Using the Template Toolkit*, is a tutorial on building a web site using the Template Toolkit. It gives a brief overview of many of the features of the Template Toolkit that are covered in more detail later in the book.

Chapter 3, *The Template Language*, begins our detailed look at the Template Toolkit. In this chapter, we look at the syntax of the Template Toolkit's presentation language.

Chapter 4, *Template Directives*, covers the syntax and use of the many templating directives that can be used from the Template Toolkit.

Chapter 5, *Filters*, takes a look at filters. These are extensions to the Template Toolkit that allow you to filter your data in various ways before presenting it to your users. This chapter includes a guide to the various standard filters that are included with the Template Toolkit distribution.

Chapter 6, *Plugins*, looks at the Template Toolkit plugins. Plugins are another way to extend the Template Toolkit by giving your templates access to powerful external modules. This chapter includes a guide to the various standard plugins that are included with the Template Toolkit distribution.

Chapter 7, *Anatomy of the Template Toolkit*, looks under the covers of the Template Toolkit and examines in some detail how it all works from the inside.

Chapter 8, *Extending the Template Toolkit*, covers ways to extend the Template Toolkit by writing your own filters and plugins.

Chapter 9, *Accessing Databases*, looks in detail at writing templates that access data held in various different types of databases.

Chapter 10, *XML*, looks at using the Template Toolkit to generate XML. It also covers reading XML documents and using their contents from within your templates.

Chapter 11, *Advanced Static Web Page Techniques*, starts to put together everything we've covered in the previous chapters and shows how to build a static web site using the Template Toolkit.

Chapter 12, *Dynamic Web Content and Web Applications*, extends the example of the previous chapter to add dynamic content to your web site.

Appendix A, describes the configuration options for the Template Toolkit and Apache::Template.

Conventions Used in This Book

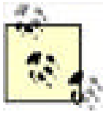
The following typographical conventions are used throughout this book:

`Constant width`

Used for Perl code, Template Toolkit directives, HTML, and code examples.

Italic

Used for filenames, URLs, hostnames, first use of terms, and emphasis.



Indicates a tip, suggestion, or general note.

Indicates a warning or caution.

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/perltt

To comment on or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, software, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

http://www.oreilly.com

Acknowledgments

This book would not be possible without the contribution and support of many individuals, including friends, family, and the hard-working folks at O'Reilly & Associates, Inc. All three of us wish to thank our production team and, in particular, our editor, Nathan Torkington, for his fine word wrangling and masterful cat herding. We would also like to thank our technical reviewers Chris Devers, Mark Fowler, Andrew Langmead, Martin Portman, and Simon Matthews for their detailed and insightful comments.

Andy Wardley

I'd like to start by thanking Dave, Darren, Nat, and the production team at O'Reilly for turning a bunch of words into a book. I would also like to thank Dom Millar for suggesting a badger for the front cover, and the design team for accommodating us with this beautiful animal.

The Template Toolkit has long since ceased to be a product of my work alone, if indeed it ever was. It owes its success to the dedicated efforts of an extended team of developers, testers, documenters, and users. At the time of this writing, the Template Toolkit documentation lists over sixty contributors who have donated their time and effort in different ways. Our collective thanks go to each of them: Chuck Adams, Stephen Adkins, Ivan Adzhubey, Mark Anderson, Bradley Baetz, Thierry-Michel Barral, Craig Barratt, Stas Bekman, Tony Bowden, Neil Bowers, Leon Brocard, Lyle Brooks, David Cantrell, Piers Cawley, Darren Chamberlain, Eric Cholet, Dave Cross, Chris Dean, Francois Desarmenien, Horst Dumcke, Mark Fowler, Michael Fowler, Axel Gerstmair, Dylan William Hardison, Perrin Harkins, Bryce Harrington, Dave Hodgkinson, Harald Joerg, Colin Johnson, Vivek Khera, Rafael Kitover, Ivan Kurmanov, Hans von Lengerke, Jonas Liljegren, Simon Luff, Paul Makepeace, Gervase Markham, Simon Matthews, Robert McArthur, Craig McLane, Leslie Michael Orchard, Eugene Miretskiy, Tatsuhiko Miyagawa, Keith G. Murphy, Chris Nandor, Briac Pilpré, Martin Portman, Slaven Rezic, Christian Schaffner, Randal L. Schwartz, Paul Sharpe, Ville Skyttä, Doug Steinwand, Michael Stevens, Drew Taylor, Swen Thuemmler, Richard Tietjen, Stathy G. Touloumis, Jim Vaughan, Simon Wilcox, and Chris Winters.

Special thanks are due to Simon Matthews, who has been using and abusing the Template Toolkit and its predecessors from the very start. Countless pints of Guinness have been consumed through long evenings spent discussing the design, development, and general direction of the project. I would also like to thank Martin Portman for the many enjoyable hours we have spent at the whiteboard, engaged in animated conversation and frantic scribbling. Many of the important TT design decisions have been thrashed out in the company of Simon and Martin. Their efforts and input continue to be gratefully received.

I would also like to thank all my other friends and colleagues of past and present at Knowledge Pool, Canon Research Centre Europe, and Fotango, many of whom are listed above. Each of these organizations and the people within them have played important roles in the evolution of the Template Toolkit.

Finally I would like to thank my wife, Sheila, and son, Ben, for their love, patience, and understanding. Writing this book ate up far too much of the time that should have been spent with you.

Darren Chamberlain

I'd like to thank my wife and kids for their help and support, and for being so understanding of the time I've spent writing instead of mowing the lawn or playing. This wouldn't have been possible for me otherwise, and I appreciate it more than they know.

Thanks to *Boston.com* for having the incredibly sane policy of using the best tool for the job, which means letting me use the Template Toolkit for so many things; to Andrew Langmead, Chris Devers, and Mike Melillo for proofreading, fact-checking, and putting up with me in general; and to Marc Lavallee, for introducing me to TT in first place.

Thanks to Andy for writing the Template Toolkit, which is as fine and versatile a piece of software as I've seen in a long time. Andy, Dave, and Nat have all been great—I hope I get to work them again.

And, of course, thanks to everyone who buys the book and keeps O'Reilly (and their fine authors!) afloat.

David Cross

I'd like to thank Andy for developing the Template Toolkit and both Darren and Andy for making the process of writing this book as much fun as it was.

Thanks to the members of the London.pm/TT cabal for first introducing me to the Template Toolkit and convincing me that it was the only templating system that I needed to look at.

Thanks to the various clients and employers who have put up with me leaving the office on time to get on with writing the book. Particular thanks should go to the people at Bibliotech who took pity on me trying to write and work simultaneously and resolved the situation by making me redundant.

Most of this book has been written while listening to music. I've found that I write best when listening to either Billy Bragg or any combination of the Waterson/Carthy clan, so thanks to them.

Thank you to Joss Whedon for cancelling "Buffy the Vampire Slayer" while I was working on this book and giving me one less reason to avoid writing.

Thank you to the various friends and family who have ensured that I still have a social life despite my seeming to do my utmost to avoid it.

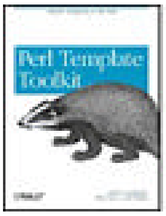
Thanks, of course, to my parents Jean and John, and to my wife Gill. Their love and support make it all much easier.

Preface

Perl Template Toolkit is an introduction to the Template Toolkit. The Template Toolkit is most often used in the creation of web sites, but it can be used as a general text manipulation tool. It is a presentation management system that allows you to separate aspects of presentation from the rest of an application, in the same way that a database allows you to separate storage concerns.

The information in this book is based on Version 2.10 of the Template Toolkit, released in July 2003. The Template Toolkit will continue to evolve. Apart from bug fixes and minor updates, the Version 2.* branch will remain pretty much the same as it is now.

Version 3, expected sometime in 2004, will include new features and some changes to the internal architecture. However, it is an important requirement that new versions of the Template Toolkit are backward-compatible with previous versions wherever possible. Although the Template Toolkit may change in some subtle ways, the basic principles, syntax, and style are here to stay.



- Table of Contents
- Index
- Reviews
- Examples
- Reader Reviews
- Errata
- Academic

Perl Template Toolkit

By Darren Chamberlain, David Cross, Andy Wardley

Publisher: O'Reilly
Pub Date: December 2003
ISBN: 0-596-00476-1
Pages: 576
Slots: 1.0