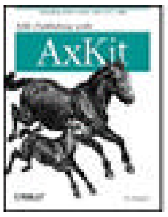


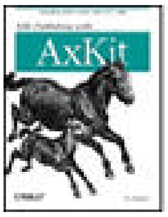
- Table of Contents
- Index
- Reviews
- Reader Reviews
- Errata
- Academic

XML Publishing with AxKit

By Kip Hampton

Publisher: O'Reilly
Pub Date: June 2004
ISBN: 0-596-00216-5
Pages: 216

XML Publishing with AxKit presents web programmers the knowledge they need to master AxKit. The book features a thorough introduction to XSP (extensible Server Pages), which applies the concepts of Server Pages technologies (embedded code, tag libraries, etc) to the XML world, and covers integrating AxKit with other tools such as Template Toolkit, Apache:: Mason, Apache::ASP, and plain CGI. It also includes invaluable reference sections on configuration directives, XPathScript, and XSP.



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

XML Publishing with AxKit

By Kip Hampton

Publisher: O'Reilly
Pub Date: June 2004
ISBN: 0-596-00216-5
Pages: 216

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

Preface

This book introduces Apache AxKit, a *mod_perl*-based extension to the Apache web server that turns Apache into an XML publishing and application environment.

Who Should Read This Book

This book is intended to be useful to any web developer/designer interested in learning about XML publishing, in general, and the practical aspects of XML publishing, specifically with the Apache AxKit XML application and publishing server. While AxKit and its techniques are the obvious focus, many ideas presented can be reused in other XML-based publishing environments. If you do not know XML and dread the thought of consuming a pile of esoteric specifications to understand what is being presented, don't worry—this book takes a fiercely pragmatic approach that will teach you only what you need to know to be productive with AxKit. A quick scan of XML's basic syntax is probably all the XML knowledge you need to get started.

Although AxKit is written in Perl, its users need not know Perl at all to use it to its full effect. However, developers who do know Perl will find that AxKit's modular design allows them to easily write custom extensions to meet specialized requirements. Similarly, AxKit users are not expected to be Apache HTTP server gurus, but those who do know even a bit about how Apache works will find themselves with a valuable head start:

- Web developers will learn XML publishing techniques through a variety of practical, tested examples.
- Perl programmers will see how they can use XML to build on their existing skills.
- Markup professionals will discover how AxKit combines standard XML processing tools with those unique to the Perl programming language to create a flexible, easy-to-use environment that delivers on XML's promise as a publishing technology.

What's Inside

This book is organized into nine chapters and one appendix.

Chapter 1, *XML as a Publishing TEchnology*, puts XML into perspective as a markup language, presents some of the topics commonly associated with XML publishing, and introduces AxKit as an XML application and publishing environment.

Chapter 2, *Installing AxKit*, guides you through the process of installing AxKit, including its dependencies and optional modules. This chapter also covers platform-specific installation tips, how to navigate AxKit's installed documentation, and where to go for additional help.

Chapter 3, *Your First XML Web Site*, guides you through the process of creating and publishing a simple XML-based web site using AxKit. Special attention is paid to the basic principles and techniques common to most projects.

Chapter 4, *Points of Style*, details AxKit's style processing directives. It gives special attention to how to combine various directives to create both simple and complex processing chains, and how to conditionally apply alternate transformations using AxKit's StyleChooser and MediaChooser plug-ins.

Chapter 5, *Transforming XML Content with XSLT*, offers a "quickstart" introduction to XSLT 1.0 and how to use it effectively within AxKit. A Cookbook-style section offers solutions to common development tasks.

Chapter 6, *Transforming XML Content with XPathScript*, introduces AxKit's more Perl-centric alternative to XSLT, XPathScript. The focus is on XPathScript's basic syntax and template options for generating and transforming XML content. The chapter also contains a Cookbook-style section.

Chapter 7, *Serving Dynamic XML Content with XPathScript*, presents a number of tools and techniques that can be used to generate dynamic XML content from within AxKit. The focus is on AxKit's implementation of eXtensible Server Pages (XSP) and on how to create reusable XSP tag libraries that map XML elements to functional code, as well as on how to use Perl's SAWA web-application framework to provide dynamic content to AxKit.

Chapter 8, *Extending AxKit*, introduces AxKit's underlying architecture and offers a detailed view of each of its modular components. The chapter pays special attention to how and why developers may develop custom components for AxKit and provides a detailed API reference for each component class.

Chapter 9, *Integrating AxKit with Other Tools*, shows how to use AxKit in conjunction with other popular web-development technologies, from plain CGI to Mason and the Template Toolkit.

Appendix A, *The AxKit Configuration Directive Reference*, provides a complete list of configuration blocks and directives.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and Unix utilities.

`Constant width`

Indicates commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, XML tags, HTML tags, macros, the contents of files, or the output from commands.

`Constant width italic`

Shows text that should be replaced with user-supplied values.

`Constant width bold`

Shows commands or other text that the user should type literally.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and by quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*XML Publishing with AxKit*, by Kip Hampton. Copyright 2004 O'Reilly Media, Inc., 0-596-00216-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

How to Contact Us

We at O'Reilly have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

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

We have a web page for this book, where we list errata, examples, or any additional information. You can access this page at:

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

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

bookquestions@oreilly.com

You can sign up for one or more of our mailing lists at:

http://elists.oreilly.com

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

http://www.oreilly.com

You may also write directly to the author at khampton@totalcinema.com.

Acknowledgments

I would like to thank my editor, Simon St. Laurent, for his wisdom and feedback, and the good folks at O'Reilly for standing behind this book and seeing it through to completion. Thanks to Matt Sergeant for coding AxKit in the first place and to Matt, Barrie Slaymaker, Ken MacLeod, Michael Rodriguez, Grant McLean, and the many other members of the Perl/XML community for their tireless efforts and general markup processing wizardry. Thanks, and a hearty and heartfelt "DAHUT!" to Robin Berjon, Jörg Walter, Michael Kröll, Steve Willer, Mike Nachbaur, Chris Prather, and the other cryptid denizens of the AxKit cabal. Finally, special thanks go out to my family, especially to my brother, Jason, whose patience, support, and encouragement truly made this book possible.

Chapter 1. XML as a Publishing Technology

In the early days of the commercial Web, otherwise reasonable and intelligent people bought into the notion that simply *having* a publicly available web site was enough. Enough to get their company noticed. Enough to become a major player in the global market. Enough to capture that magical and vaguely defined commodity called *market share*. Somehow that would be enough to ensure that consumers and investors would pour out bags of money on the steps of company headquarters. In those heady days, budgets for web-related technologies appeared limitless, and the development practices of the time reflected that—it seemed perfectly reasonable to follow the celebration of a site's rollout with initial discussions about what the *next* version of that site would look like and do. (Sometimes, the next redesign was already in the works before the current redesign was even launched.) It did not matter, technically, that a site was largely hardcoded and inflexible, or that the scripts that implemented the dynamic applications were messy and impossible to maintain over time. What mattered was that the project was done quickly. If a few bad choices were made along the way, it was thought, they could always be addressed during the inevitable redesign.

Those days are gone.

The goldrush mentality has receded and companies and other organizations are looking for more from their investment in the Web. Simply having a site out there is not enough (and truly, it never was). The site must do something that measurably adds value to the organization and that value must exceed the cost of developing the site in the first place. In other words, the New Economy had a rather abrupt introduction to the rules of Business As Usual. This industry-wide belt-tightening means that web developers must adjust their approach to production. Companies can no longer afford to write off the time and energy invested in developing a web site simply to replace it with something largely similar. Developers are expected to provide dynamic, malleable solutions that can evolve over time to include new content, dynamic features, and support for new types of client software. In short, today's developers are being asked to do more with less. They need tools that can cope with major changes to a site or an application without altering the foundation that is already there.

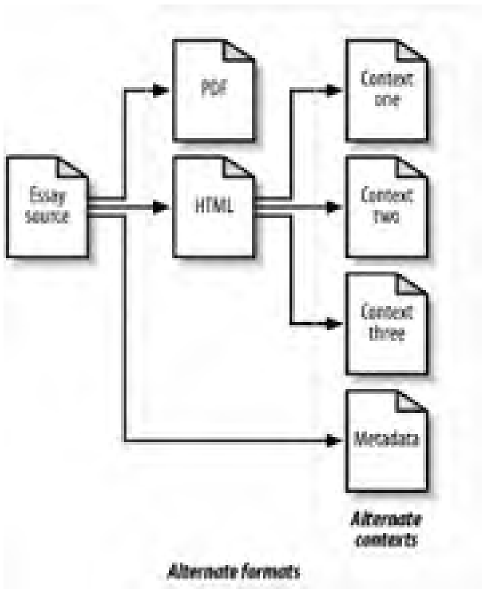
Far from being a story of gloom and doom, the slimming of web budgets has led to a natural and positive reevaluation of the tools and techniques that go into developing and maintaining online media and applications. The need to provide more options with fewer resources is driving the creative development of higher-level application and publishing frameworks that are better able to meet changing requirements over time with a minimum of duplicated effort. Ironically, in many ways, the "dot bomb" was the best thing that could have happened to web software.

One key concept behind today's more adaptive web solutions lies in making sure that the content of the site is *reusable*. By reusable content I mean that the essential information is captured (or available) in a way that lends itself to different uses or views of that data based on the context in which it is being requested. Consider, for example, the task of publishing an informal essay about the life of Jazz great Louis Armstrong. Presuming you will only be publishing this document via the Web, you still have a variety of choices about the form in which the document will be available. You could publish it in HTML for faster downloading or PDF for finer control over the visual layout. If you limit the choice to HTML, you still have many choices to make—what links, ad banners, and other supporting content will you include? Does the data include a generic boilerplate that is the same for every page on the site, or will you attempt to provide a more intimate sense of context by providing links to other related topics? If you want to offer a sense of context, how do you decide what is related? Do you frame the essay in the context of influential Jazz musicians, prominent African Americans, or famous natives of New Orleans? Given that each of these contexts is arguably valid, what if you want to present all three and let the user decide which navigational path suits her interests best? You could also say that the essay's metadata (its title, author's name, abstract summary, etc.) is really just another way of looking at the same document, albeit a highly selective and filtered one. Each of these choices represents nothing more than an alternative *contextual view* of the same content (the Armstrong essay). All that really changes is the way in which that content is presented.

Figure 1-1 shows a simple representation of your essay and some of its possible alternate views. How could you hit all of these targets? Obviously, you could hand-author the document in each of the various formats and contexts, but that would be time-consuming and unrealistic for all but the tiniest of sites. Ideally, what you want is a system that:

- Stores the data in a rich and meaningful way so users can access it easily at various levels of detail
- Provides an easy way to add alternate (expanded or filtered) views of that data without requiring changes to the source document (or, in the case of dynamic content, the code that generates it)

Figure 1-1. Multiple views of a single document



Although many web-development frameworks offer the ability to create sites in a modular fashion through reusable components, most focus largely on automating redundancy through the inclusion of common content blocks and use of code macros. These systems recognize the value of separating content from logic, but they are typically designed to construct documents in only *one* target format. That is, the templates, widgets, and content (or content-generating code) are all focused on constructing a single kind of document (usually HTML). Rendering the same content in multiple formats is cumbersome and often requires so much duplication at the component level that modularity becomes more burden than blessing. One technology, however, is firmly rooted in the ideas of generating context-specific representations of rich content sources through both modular construction and data transformation—that technology is XML.

This is where the subject of this book, AxKit, comes in. As an XML publishing and application server, AxKit begins with XML's high-level notion of reusable content and seeks to simplify the tasks associated with creating dynamic, context-sensitive representations from rich XML sources. That is, the fact that you need to deliver the same content in a variety of ways is a given, and part of what AxKit does is to provide a framework to ensure that the core content is transformed correctly for the given situation.

1.1 Exploding a Few Myths About XML Publishing

XML and its associated technologies have generated enormous interest. XML pundits describe in florid terms how moving to XML is the first step toward a Utopian new Web, while well-funded marketing departments churn out page after page of ambiguous doublespeak about how using XML is the cure for everything from low visitor traffic to male-pattern baldness. While you may admire visionary zeal on the one hand and understand the simple desire to generate new business on the other, the unfortunate result is that many web developers are confused about what XML is and what it is good for. Here, I clear up a few of the more common fallacies about XML and its use as a web-publishing technology.

Using XML means having to memorize a pile of complex specifications.

There is certainly no shortage of specifications, recommendations, or white papers that describe or relate to XML technologies. Developing even a cursory familiarity with them all would be a full-time job. The fact is, though, that many of these specifications only describe a single *application* of XML. Unless that tool solves a specific existing need, there's no reason for a developer to try to use it, especially if you come to XML from an HTML background. A general introduction to XML's basic rules, and perhaps a quick tutorial or two that covers XSLT or another transformative tool, are all you need to be productive with XML and a tool such as AxKit. Be sane. Take a pragmatic approach: learn only what you need to deliver on the requirements at hand.

Moving to XML means throwing away all the tools and techniques that I have learned thus far.

XML is simply a way to capture data, nothing more. No tool is appropriate for all cases, and knowing how to use XML effectively simply adds another tool to your bag of tricks. Additionally (as you will see in Chapter 9), many tools you may be using today can be integrated seamlessly into AxKit's framework. You can keep doing what worked well in the past while taking advantage of what AxKit may offer in the way of additional features.

XML is totally revolutionary and will solve all of my publishing problems.

This is the opposite of the previous myth but just as common. Despite considerable propaganda to the contrary, XML offers nothing more than a way to represent information. In itself, XML does not address the issues of archiving, information retrieval, indexing, administration, or any other tasks associated with publishing documents. It may make finding or building tools to perform these tasks simpler, faster, more straightforward, or less ad hoc, but no magic is involved.

XML is useful only for transferring data structures among web services.

Two popular exchange protocols, SOAP and XML-RPC, use XML to capture data, but suggesting that this is the *only* legitimate use for XML is simply wrong. In fact, XML was originally intended primarily as a publishing technology. Tools such as SOAP only emerged later when it was discovered that XML was quite handy for capturing complex data in a way that common programming languages could share. To say that XML is only useful for transferring data between applications is a bit like saying that the ASCII text format is only useful for composing email messages—popular, yes; exclusive, no.

My project only requires documents to be available to web browsers as HTML; using XML would add complexity and overhead without adding value.

It is true—needing to deliver the same content to different target clients is a compelling reason to consider XML publishing, but it is certainly not the only one. Separating the content from its presentation also provides the ability to fundamentally alter the look and feel of an entire site without worrying about the information being communicated getting clobbered in the bargain. Similarly, new site design prototypes can be created using the actual content that will be delivered in production rather than the boilerplate filler that so often only favors the designers' sense of aesthetics.

As for performance, true XML publishing frameworks such as AxKit offer the ability to cache transformed content—even several views of the same document—and will only reprocess when either the source XML or the stylesheets being applied are modified (or when explicitly configured, reprocess for each request). The latest data available shows that AxKit can deliver cached, transformed content at roughly 90% of the speed (requests per second) offered by serving the same content as static HTML.

1.2 XML Basics

Markup technology has a long and rich history. In the 1960s, while developing an integrated document storage, editing, and publishing system at IBM, Charles Goldfarb, Edward Mosher, and Raymond Lorie devised a text-based markup format. It extended the concepts of generic coding (block-level tagging that was both machine-parsable and meaningful to human authors) to include formal, nested elements that defined the type and structure of the document being processed. This format was called the Generalized Markup Language (GML). GML was a success, and as it was more widely deployed, the American National Standards Institute (ANSI) invited Goldfarb to join its Computer Languages for Text Processing committee to help develop a text description standard-based GML. The result was the Standard Generalized Markup Language (SGML). In addition to the flexibility and semantic richness offered by GML, SGML incorporated concepts from other areas of information theory; perhaps most notably, inter-document link processing and a practical means to *programmatically validate* markup documents by ensuring that the content conformed to a specific grammar. These features (and many more) made SGML a natural and capable fit for larger organizations that needed to ensure consistency across vast repositories of documents. By the time the final ISO SGML standard was published in 1986, it was in heavy use by bodies as diverse as the Association of American Publishers, the U.S. Department of Defense, and the European Laboratory for Particle Physics (CERN).

In 1990, while developing a linked information system for CERN, Tim Berners-Lee hit on the notion of creating a small, easy-to-learn subset of SGML. It would allow people who were not markup experts to easily publish interconnected research documents over a network—specifically, the Internet. The Hypertext Markup Language (HTML) and its sibling network technology, the Hypertext Transfer Protocol (HTTP) were born. Four years later, after widespread and enthusiastic adoption of HTML by academic research circles throughout the globe, Berners-Lee and others formed the World Wide Web Consortium (W3C) in an effort to create an open but centralized organization to lead the development of the Web.

Without a doubt, HTML brought markup technology into the mainstream. Its simple grammar, combined with a proliferation of HTML-specific markup presentation applications (web browsers) and public commercial access to the Internet sparked what can only be called a popular electronic markup publishing explosion. No longer was markup solely the domain of information technology specialists working with complex, mainframe-based publishing tools inside the walls of huge organizations. Anyone with a home PC, a dial-up Internet account, and patience to learn HTML's intentionally forgiving syntax and grammar could publish his own rich hypertext documents for the rest of the wired world to see and enjoy.

HTML made markup popular, but it was a single, predefined grammar that only indicated how a document was to be presented visually in a web browser. That meant much of the flexibility offered by markup technology, in general, was simply lost. All the markup reliably communicated was how the document was supposed to *look*, not what it was supposed to *mean*. In the mid-1990s, work began at the W3C to create a new subset of SGML for use on the Web—one that provided the flexibility and best features of its predecessor but could be processed by faster, lighter tools that reflected the needs of the emerging web environment. In 1996, W3C members Tim Bray and C. M. Sperberg-McQueen presented the initial draft for this new "simplified SGML for Web"—the Extensible Markup Language (XML). Two years later in 1998, after much discussion and rigorous review, the W3C published XML 1.0 as an official recommendation.

In the six years since, interest in XML has steadily grown. While not as ubiquitous as some claim, tools to process XML are available for the most popular programming languages, and XML has been used in some fairly novel (though sometimes not always appropriate) ways. Given its generic nature, inherent flexibility, and ways in which it has (or can be) used, XML is hard to pigeonhole. It remains largely an enigma to many developers. At its core, XML is nothing, more or less, than a text-based format for applying structure to documents and other data. Uses for XML are (and will continue to be) many and varied, but looking back at its history helps to provide a reasonable context—a history inextricably bound to automated document publishing.

Many people, especially those coming to XML from a web-development background, seem to expect that it is either intended to replace HTML or that it is somehow HTML: The Next Generation—neither is the case. Although both are markup languages, HTML defines a specific markup grammar (set of elements, allowed structures) intended for consumption by a single type of application: an HTML web browser. XML, on the other hand, does not define a grammar at all. Rather, it is designed to allow developers to use (or create) a grammar that best reflects the structure and meaning of the information being captured. In other words, it gives you a clear way to create the rich, reusable source content crucial to modern adaptive web-publishing systems.

To understand the value of using a more semantically meaningful markup grammar, consider the task of publishing a poetry collection. If you know HTML and want to get the collection onto the Web quickly, you could create a document, such as the one shown in Example 1-1, for each poem.

Example 1-1. poem.html

```
<html>

  <head>

    <title>Post-Geek-chic Folk Poetry Collection</title>

  </head>
  <body>

  <h1>An Ode To Directed Acyclic Graphs</h1>

  <p><i>by: Anonymous</i></p>

  <p>

   I think that I shall never see, <br>

   a document that cannot be represented as a tree.

  </p>

  </body>

</html>
```

If your only goal is to publish your poetic gems on the Web for people to view in a browser, then once you upload the documents to the right location on an appropriate server somewhere, the job is done. What if you want to do more? At the very least, you will probably want an index document containing a list of links to the poems in your collection. If the collection remains small and time is not a consideration, you could create this index by hand. More likely, though, because you are a professional web developer, you would probably create a small script to extract information (title and author) from the poems themselves to create the index document programatically. That's when the weakness in your approach begins to show. Specifically, using HTML to mark up your poetry only gave you a way to present the work visually. In your attempt to extract the title and author's name, you are forced to impose meaning based solely on inference and your knowledge of the conventions used when marking up the poems. You can *infer* that the first h1 element contains the title of the poem, but nothing states this explicitly. You must trust that all poems in the collection will follow the same structure. In the best case, you can only guess and hope that your guess holds up in the long run.

Marking up your poetry collection in XML can help you avoid such ambiguities. It is not the use of XML, *per se*, that helps. Rather, XML gives you a familiar syntax (nested angle-bracketed tags with attributes, such as those in HTML) while offering the flexibility to choose a grammar that more intimately describes the structure and meaning of the content. It would help simplify your indexing script, for example, if something like an author element contained the author's name. You would not have to rely on an unstable heuristic such as "the string that follows the word `by,' optionally contained in an i element, that is in the first p element after the first h1 element in the document" to extract the data. Essentially, you want to use a more exact, domain-specific grammar whose structures and elements convey the meaning of the data. XML provides a means to do that.

Not surprisingly, marking up poetic content is a task that others before you have faced. A quick web search reveals several XML grammars designed for this purpose. A short evaluation of each reveals that the poemsfrag Document Type Definition (DTD) from Project Gutenberg (a volunteer effort led by the HTML Writer's Guild to make the World's great literature available as electronic text) fits your needs nicely. Using the grammar defined by *poemsfrag.dtd*, the sample poem from your collection takes the form shown in Example 1-2.

Example 1-2. poem.xml

```
<?xml version="1.0"?>

<poem>

  <title>An Ode To Directed Acyclic Graphs</title>

  <author>Anonymous</author>

  <verse>

    <line>I think that I shall never see,</line>

    <line>a document that cannot be represented as a tree.</line>

  </verse>

</poem>
```

Using this more specific grammar makes extracting the title and author data for the index document completely unambiguous—you simply grab the contents of the title and author elements, respectively. In addition, you can now easily generate other interesting metadata, such as the number of verses per poem, the average lines per verse, and so on, without dubious guesswork. Moreover, having an explicit, concrete Document Type Definition that describes your chosen grammar provides the chance to programatically validate the structure of each poem you add to the collection. This helps to ensure the integrity of the data from the outset.

Choosing the best grammar (or data model, if you must) for your content is crucial: get it right and the tools to process your documents will grow logically from the structure; get it wrong and you will spend the life of the project working around a weak foundation. Designing useful markup grammars that hold up over time is an art in itself; resist the urge to create your own just because you can. Chances are there is already a grammar available for the class of documents you will mark up. Evaluate what's available. Even if you decide to go your own way, the time spent seeing how others approached the same problem more than pays for itself.

Switching to XML and the poemsfrag grammar arguably adds significant value to your documents—the structure reveals (or imposes) the intended meaning of the content. At the very least, this reduces time wasted on messy guessing both for those marking up the poems and for those writing tools to process those poems. However, you lose something, as well. You can no longer simply upload the documents to a web server and expect browsers to do the right thing when rendering them (as you could when they were marked up as HTML). There is a gap between the grammar that is most useful to us, as authors and tool builders, and the grammar that an HTML web browser expects. Since publishing your poetry online was the goal in the first place, unless you can bridge that gap (and easily too), then really, you take a step backward.

1.3 Publishing XML Content

In the most general sense, delivering XML documents over the Web is much the same as serving any other type of document—a client application makes a request over a network to a server for a given resource, the server then interprets that request (URI, headers, content), returns the appropriate response (headers, content), and closes the connection. However, unlike serving HTML documents or MP3 files, the intended use for an XML document is not apparent from the format (or content type) itself. Further processing is usually required. For example, even though most modern web browsers offer a way to view XML documents, there is no way for the browser to know how to render your custom grammar visually. Simply presenting the literal markup or an expandable tree view of the document's contents usually communicates nothing meaningful to the user. In short, the document must be *transformed* from the markup grammar that best fits your needs into the format that best fits the expectations of the requesting client.

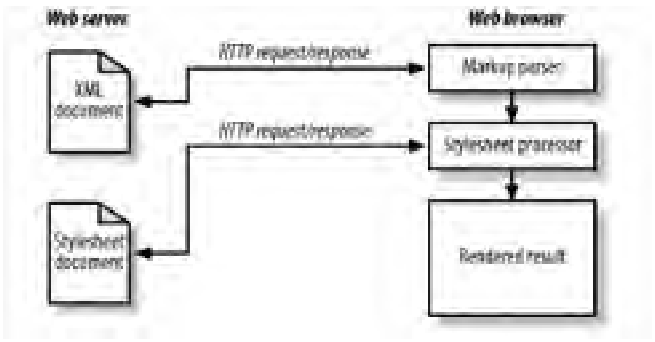
This separation between the source content and the form in which it will be presented (and the need to transform one into the other) is the heart and soul of XML publishing. Not only does making a clear distinction between content and presentation allow you to use the grammar that best captures your content, it provides a clear and logical path toward reusing that content in novel ways without altering the data's source. Suppose you want to publish the poems from the collection mentioned in the previous section as HTML. You simply transform the documents from the poemsfrag grammar into the grammar that an HTML browser expects. Later, if you decide that PDF or PostScript is the best way to deliver the content, you only need to change the way the source is transformed, not the source itself. Similarly, if your XML expresses more record-oriented data—generated from the result of an SQL query, for example—the separation between content and presentation offers a way to provide the data through a variety of interfaces just by changing the way the markup is transformed.

Although there are many ways to transform XML content, the most common is to pass the document—together with a *stylesheet* document—into a specialized processor that transforms or renders the data based on the rules set forth in the stylesheet. Extensible Stylesheet Language Transformations (XSLT) and Cascading Stylesheets (CSS) are two popular variations of this model. Putting aside features offered by various stylesheet-based transformative processors for later chapters, you still need to decide *where* the transformation is to take place.

1.3.1 Client-Side Transformations

In the client-side processing model, the remote application, typically a web browser, is responsible for transforming the requested XML document into the desired format. This is usually achieved by extracting the URL for the appropriate stylesheet from the href attribute of an xml-stylesheet processing instruction or link element contained in the document, followed by a separate request to the remote server to fetch that stylesheet. The stylesheet is then applied to the XML document using the client's internal processor and, assuming no errors occur along the way, the result of the transformation is rendered in the browser. (See Figure 1-1.)

Figure 1-2. The client-side processing model



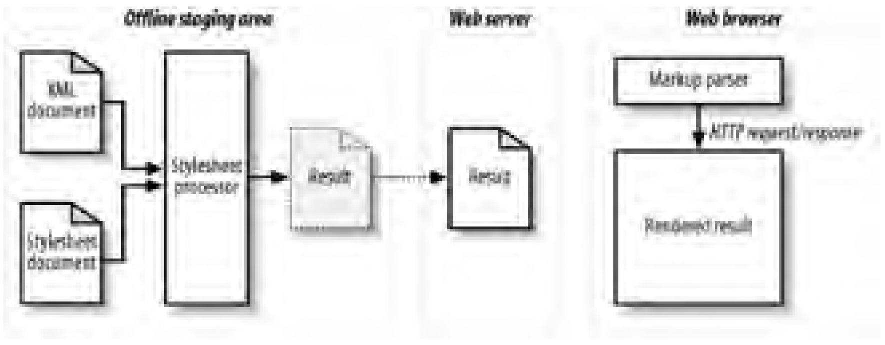
Using the client-side approach has several benefits. First, it is trivial to set up a web server to deliver XML documents in this manner—perhaps adding a few lines to the server's *mime.conf* file to ensure that the proper content type is part of the outgoing response. Also, since the client handles all processing, no additional XML tools need to be installed and configured on the server. There is no additional performance hit over and above serving static HTML pages, since documents are offered up as is, without additional processing by the server.

Client-side processing also has weaknesses. It assumes that the user at the other end of the request has an appropriate browser installed that can process and render the data correctly. Years of working around browser idiosyncrasies have taught web developers not to rely too heavily on client-side processing. The stakes are higher when you expect the browser to be solely responsible for extracting, transforming, and rendering the information for the user. Developers lose one of the important benefits of XML publishing, namely, the ability to repurpose content for different types of client devices such as PDAs, WAP phones, and set-top boxes. Many of these platforms cannot or do not implement the processors required to transform the documents into the proper format.

1.3.2 Preprocessed Transformations

Using preprocessed transformations, the appropriate stylesheets are applied to the source content offline. Only the results of those transformations are published. Typically, a staging area is used, where the source content is transformed into the desired formats. The results are copied from there into the appropriate location on the publicly available server, as shown in Figure 1-2.

Figure 1-3. The preprocessed transformation model



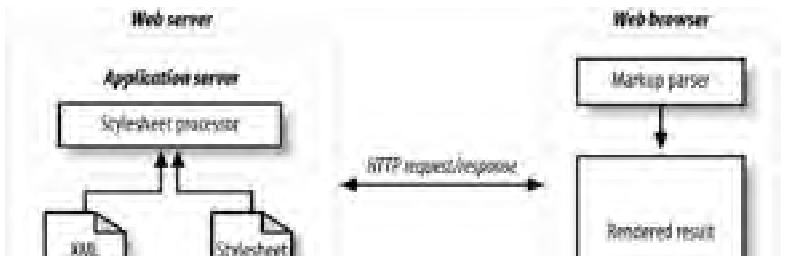
On the plus side, transforming content into the correct format ahead of time solves potential problems that can arise from expecting too much from the requesting client. That is to say, for example, that the browser gets the data that it can cope with best, just as if you authored the content in HTML to begin with, and you did not introduce any additional risk. Also, as with client-side transformations, no additional tools need to be installed on the web-server machine; any vanilla web server can capably deliver the preprocessed documents.

On the down side, offline preprocessing adds at least one additional step to publishing every document. Each time a document changes, it must be retransformed and the new version published. As the site grows or the number of team members increases, the chances of collision and missed or slow updates increase. Also, making the same content available in different formats greatly increases complexity. A simple text change, for example, requires a content transformation for each format, as well as a separate URL for each variation of every document. Scripted automation can help reduce some costs and risks, but someone must write and maintain the code for the automation process. That means more time and money spent. In any case, the static site that results from offline preprocessing lacks the ability to repurpose content on the fly in response to the client's request.

1.3.3 Dynamic Server-Side Transformations

In the server-side runtime processing model, all XML data is parsed and then transformed on the server machine before it is delivered to the client. Typically, when a request is received, the web server calls out via a server extension interface to an external XML parser and stylesheet processor that performs any necessary transformations on the data before handing it back to the web server to deliver to the client. The client application is expected only to be able to render the delivered data, as shown in Figure 1-3.

Figure 1-4. The server-side processing model



Handling all processing dynamically on the server offers several benefits. It is a given that a scripting engine or other application framework will be called on to process the XML data. As a result, the same methods that can be used from within that framework to capture information about a given request (HTTP cookies, URL parameters, POSTed form data, etc.) can be used to determine which transformations occur and on which documents. In the same way, access to the user agent and accept headers gives the developer the opportunity to detect the type of client making the connection and to transform the data into the appropriate format for that device. This ability to transform documents differently, based on context, provides the dynamic server-side processing model a level of flexibility that is simply impossible to achieve when using the client-side or preprocessed approaches.

Server-side XML processing also has its downside. Calling out to a scripting engine, which calls external libraries to process the XML, adds overhead to serving documents. A single transformation from Simplified DocBook to HTML may not require a lot of processing power. However, if that transformation is being performed for each request, then performance may become an issue for high traffic sites. Depending on the XML interface used, the in-memory representation of a given document is 10 times larger than its file size on disk, so parsing large XML documents or using complex stylesheets to transform data can cause a heavy performance hit. In addition, choosing to keep the XML processing on the server may also limit the number of possible hosting options for a given project. Most service providers do not currently offer XML processing facilities as part of their basic hosting packages, so developers must seek a specialty provider or co-locate a server machine if they do not already host their own web servers.

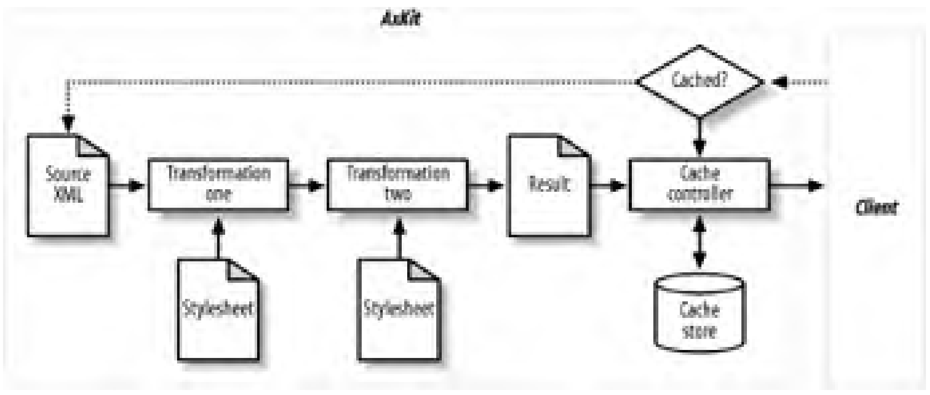
Comparing these three approaches to publishing XML content, you can generally say that dynamic server-side processing offers the greatest flexibility and extensibility for the least risk and effort. The cost of server-side processing lies largely in finding a server that provides the necessary functionality—a far more manageable cost, usually, than that of working around client-side implementations beyond your control or writing custom offline processing tools.

1.4 Introducing AxKit, an XML Application Server for Apache

Originally conceived in 2000 by Matt Sergeant as a Perl-powered alternative to the then Java-centric world of XML application servers, AxKit (short for Apache XML Toolkit) uses the *mod_perl* extension to the Apache HTTP server to turn Apache into an XML publishing and application server. AxKit extends Apache by offering a rich set of server configuration directives designed to simplify and automate common tasks associated with publishing XML content, selecting and applying transformative processes to XML content to deliver the most appropriate result.

Using AxKit's custom directives, content transformations (including *chains* of transformations) can be applied based on a variety of conditions (request URI, aspects of the XML content, and much more) on a resource-by-resource basis. Among other things, this provides the ability to set up multiple, alternate styles for a given resource and then select the most appropriate one at runtime. Also, by default, the result of each processing chain is cached to disk on the first request. Unless the source XML or the stylesheets in the chain change, all subsequent requests are to be served from the cache. Figure 1-4 illustrates the processing flow for a resource with one associated processing chain consisting of two transformations.

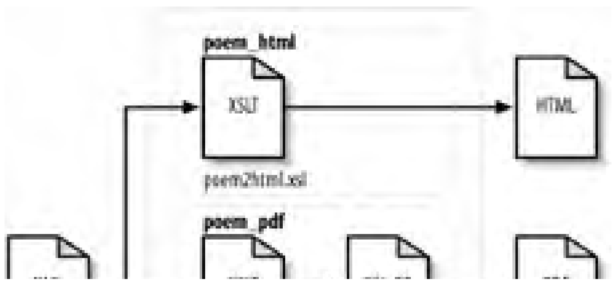
Figure 1-5. Basic two-stage processing chain

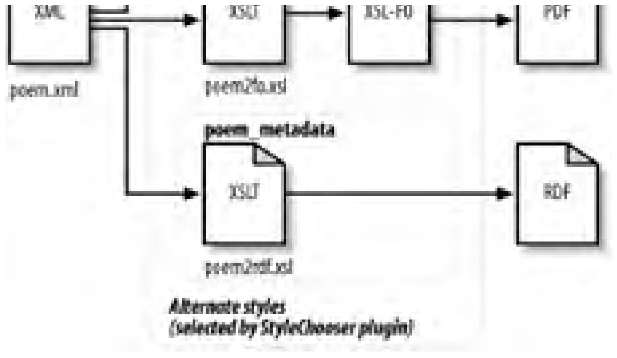


In its design, AxKit implements a modular system that divides the low-level tasks required for serving XML data across a series of swappable component classes. For example, Provider classes are responsible for fetching the sources for the content and stylesheets associated with the current request, while Language modules implement interfaces to the various transformative processors. (You can find details of each type of component class in Chapter 8.) This modular design makes AxKit quite extensible and able to cope with heterogeneous publishing strategies. Suppose that some content you are serving is stored in a relational database. You need only swap in a Provider class that selects the appropriate data for those pages from the database, while still using the default filesystem-based Provider for static documents stored on the disk. Several alternative components of various classes ship with the core AxKit distribution, and many others are available via the Comprehensive Perl Archive Network. Often, little or no custom code needs to be written. You simply drop in the appropriate component and configure its options.

We will look at each AxKit option for creating style processing chains in depth in Chapter 4. But for now, recall the collection of poems that you marked up using the poemsfrag Document Type Definition earlier in this chapter. Also, remember that when you left off, you were a bit stuck: the poems' markup captured the content in a semantically meaningful way, but by abandoning HTML as the source grammar, you lost the ability to just upload the document to a web server and expect that browsers would render it properly. This is precisely the type of task that AxKit was designed to address. Figure 1-5 illustrates a single source document containing a poem and three alternative processing chains implemented as named styles that can be selected at run-time to render that poem in various formats.

Figure 1-6. Alternate style chains





Here is a sample configuration snippet that would implement these styles, making each selectable by adding a style parameter with the appropriate value to the request's query string:

```
<Directory /poems>
    <Files *.xml>
        # choose styles based on the query string
        AxAddPlugin Apache::AxKit::StyleChooser::QueryString

        # renders the poem as HTML
        <AxStyleName poem_html>
            AxAddProcessor text/xsl /styles/poem2html.xsl
        </AxStyleName>

        # generates the poem as PDF

        <AxStyleName poem_pdf>
            AxAddProcessor text/xsl /styles/poem2fo.xsl
            AxAddProcessor application/x-xsl-fo NULL
        </AxStyleName>

        # extracts the metadata from the poem and renders it as RDF
        <AxStyleName poem_rdf>
            AxAddProcessor text/xsl /styles/poem2rdf.xsl
        </AxStyleName>

        # set a default style if none is passed explicitly
        AxStyle poem_html
    </Files>
</Directory>
```

With this in place, you can put your XML documents that use the poemsfrag grammar into the poems directory and render each poem in one of three formats. For example, a request to http://that.host/poems/mypoem.xml?style=poem_pdf returns the selected poem as a PDF document. A request for the same poem with style=poem_rdf in the query string offers the metadata about the selected poem as an RDF document. In each case, the source document does not change. Only the styles *applied* to its contents differ.

Finally, it worth noting here that AxKit is an officially sanctioned Apache Software Foundation (ASF) project. This means that AxKit is not an experimental hobbyware project. Rather it is a battle-tested framework developed and maintained by a community of committed professional developers who need to solve real-world problems. No project of any size is entirely bug-free, but AxKit's role as an ASF-blessed project means, at the very least, that it is held to a high standard of excellence. If something does go wrong, its users can fully expect an active community to be around to address the problem, both now and in the future.

Chapter 2. Installing AxKit

AxKit combines the power of Perl's rich and varied XML processing facilities with the flexibility of the Apache web server. Rather than implementing such an environment in a monolithic package, as some application servers do, it takes a more modular approach. It allows developers to choose the lower-level tools such as XML parsers and XSLT processors for themselves. This neutrality with respect to lower-level tools gives AxKit the ability to adapt and incorporate new, better performing, or more feature-rich tools as quickly as they appear. That flexibility costs, however. You will probably have to install more than just the AxKit distribution to get a working system.

2.1 Installation Requirements

To get AxKit up and running, you will need:

- The Apache HTTP server (Version 1.3.*x*)
- The mod_perl Apache extension module (Version 1.26 or above)
- An XML parser written in Perl or, more commonly, one written in C that offers a Perl interface module
- The core AxKit distribution

2.1.1 Installing Apache and mod_perl

If you are running an open source or open source-friendly operating system such as GNU/Linux or one of the BSD variants (including Mac OS X), chances are good that you already have Apache and mod_perl installed. If this is the case, then you probably will not have to install them by hand. Simply make sure that you are running the most recent version of each, and skip directly to the next section. However, in some cases, using precompiled binaries of Apache and *mod_perl* proved to be problematic for people who want to use AxKit. In most cases, neither the binary in question, nor AxKit, are really broken. The problem lies in the fact that binaries built for public distribution are usually compiled with a set of general build arguments, not always well suited for specialized environments such as AxKit. If you find that all AxKit's dependencies install cleanly, but AxKit's test suite still fails, you may consider removing the binary versions and installing Apache and *mod_perl* by hand. At the time of this writing, AxKit runs only under Apache versions in the 1.3.*x* branch. Support for Apache 2.*x* is currently in development. Given that Apache 2 is quite different from previous versions, both in style and substance, the AxKit development team decided to take things slowly to ensure that AxKit for Apache 2.*x* offers the best that the new environment has to offer.

To install Apache and mod_perl from the source, you need to download the source distributions for each from http://httpd.apache.org/ and http://perl.apache.org/, respectively. After downloading, unpack both distributions into a temporary directory and *cd* into the new *mod_perl* directory. A complete reference for all options available for building the Apache server and mod_perl is far beyond the scope of this book. The following will get you up and running with a useful set of features:

```
$ perl Makefile.PL \
> EVERYTHING=1 \
> USE_APACI=1 \
> DYNAMIC=1 \
> APACHE_SRC=../apache_1.3.xxx/src \
> DO_HTTPD=1 \
> APACI_ARGS="--enable-module=so --enable-shared=info \
> --enable-shared=proxy --enable-shared=rewrite \
> --enable-shared=log_agent"
$ make
$ make install
```

All lines before the make command are build flags that are being passed to perl Makefile.PL. The \ characters are simply part of the shell syntax that allows you to divide the arguments across multiple lines. The > characters represent the shell's output, and you should not include them. Also, be sure to replace the value of the APACHE_SRC option with the actual name of the directory into which you just unpacked the Apache source.

2.1.2 XML Processing Options

As I mentioned in the introduction to this chapter, AxKit is a publishing and application framework. It is not an XML parser or XSLT processor, but it allows you to choose among these lower-level tools while ensuring that they work together in a predictable way. If you do not already have the appropriate XML processing tools installed on your server, AxKit attempts to install the minimum needed to serve transformed XML content. However, more cautious minds may prefer to install the necessary XML parser and any optional XSLT libraries to make sure they work before installing the AxKit core. Deciding which XML parsers or other libraries to install depends on your application's other XML processing needs, but the following dependency list shows which tools AxKit currently supports and which publishing features require which libraries.

Gnome XML parser (libxml2)

> Requires: XML::LibXML
>
> Required by AxKit for: eXtensible Server Pages
>
> Available from: http://xmlsoft.org/

Expat XML parser

> Requires: XML::Parser
>
> Required by AxKit for: XPathScript
>
> Available from: http://sourceforge.net/projects/expat/

Gnome XSLT processor (libxslt)

> Requires: libxml2, XML::LibXSLT
>
> Required by AxKit for: optional XSLT processing
>
> Available from: http://xmlsoft.org/XSLT/

Sablotron XSLT processor

> Requires: Expat, XML::Sablotron
>
> Required by AxKit for: optional XSLT processing
>
> Available from: http://www.gingerall.com/

You do not need to install all these libraries before installing AxKit. For example, if you plan to do XSLT processing, you need to install *either* libxslt or Sablotron, not both. However, I do strongly recommend installing both supported XML parsers: Gnome Project's libxml2 for its speed and modern features, and Expat for its wide use among many popular Perl XML modules. In any case, remember that you must install the associated Perl interface modules for any of the C libraries mentioned above, or AxKit will have no way to access the functionality that they provide.

Again, some operating system distributions include one or more of the libraries mentioned above as part of their basic packages. Be sure to upgrade these libraries before proceeding with the AxKit installation to ensure that you are building against the most recent stable code.

2.2 Installing the AxKit Core

Now that you have an environment for AxKit to work in and have some of the required dependencies installed, you are ready to install AxKit itself. For most platforms this is a fairly painless operation.

2.2.1 Using the CPAN Shell

The quickest way to install AxKit is via Perl's Comprehensive Perl Archive Network (CPAN) and the CPAN shell. Log in as root (or become superuser) and enter the following:

```
$ perl -MCPAN -e shell

> install AxKit
```

This downloads, unpacks, compiles, and installs all modules in the AxKit distribution, as well as any prerequisite Perl modules you may need. If AxKit installs without error, you may safely skip to Section 2.4. If it doesn't, see Section 2.6 for more information.

2.2.2 From the Tarball Distribution

The latest AxKit distribution can always be found on the Apache XML site at http://xml.apache.org/dist/axkit/. Just download the latest tarball, unpack it, and *cd* to the newly created directory. As root, enter the following:

```
 $ perl Makefile.PL

 $ make

 $ make test

 $ make install
```

This compiles and installs all modules in the AxKit distribution. Just like the CPAN shell method detailed above, AxKit's installer script automatically attempts to install any module prerequisites it encounters. If *make* stops this process with an error, skip on to Section 2.6 for help. Otherwise, if everything goes smoothly, you can skip ahead to Section 2.4.

In addition to the stable releases available from CPAN and axkit.org, the latest development version is available from the AxKit project's anonymous CVS archive:

```
cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic login
```

Brave souls who like to live on the edge or who may be interested in helping with AxKit development can check it out. When prompted for a password, enter: anoncvs. You may now check out a piping hot version of AxKit:

```
<![CDATA[cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic co xml-axkit]]>
```

Installing the CVS version of AxKit is otherwise identical to installing from the tarball.

2.3 Installing AxKit on Win 32 Systems

As of this writing, AxKit's support for the Microsoft Windows environment should be considered experimental. Anyone who decides to put such a server into production does so at her own risk. AxKit *will* run in most cases. (Win9x users are out of luck.) If you are looking for an environment in which to learn XML web-publishing techniques, then AxKit on Win32 is certainly a viable choice.

If you do not already have ActiveState's Windows-friendly version of Perl installed, you must first download and install that before proceeding. It is available from http://www.activestate.com/. I suggest you get the latest version from the 5.8.*x* branch. In addition, you need the Windows port of the Apache web server. You can obtain links to the Windows installer from http://httpd.apache.org/. Be sure to grab the latest in the 1.3.*x* branch. Next, grab the official Win32 binaries for libxml2 and libxslt from http://www.zlatkovic.com/libxml.en.html and follow the installation instructions there.

After you install Apache, Perl libxml2, and libxslt, you can install AxKit using ActiveState's *ppm* utility (which was installed when you installed ActivePerl). Simply open a command prompt, and type the following:

```
C:\> ppm

ppm> repository add theoryx http://theoryx5.uwinnipeg.ca/ppms

ppm> install mod_perl-1

ppm> install libapreq-1

ppm> install XML-LibXML

ppm> install XML-LibXSLT

ppm> install AxKit-1
```

Finally, add the following line to your *httpd.conf* and start Apache:

```
LoadModule perl_module modules/mod_perl.so
```

This combination of commands and packages should give you a workable (albeit experimental) AxKit on your Windows system. If things go wrong, be sure to join the AxKit user's mailing list and provide details about the versions of various packages you tried, your Windows version, and relevant output from your error logs.

2.4 Basic Server Configuration

As you will learn in later chapters, AxKit offers quite a number of runtime configuration options that allow fine-grained control over every phase of the XML processing and delivery cycle. Getting a basic working configuration requires very little effort, however. In fact, AxKit ships with a sample configuration file that can be included into Apache's main server configuration (or used as a road map for adding the configuration directives manually, if you decide to go that way instead).

Copy the *example.conf* file in the AxKit distribution's *examples* directory into Apache's *conf* directory, renaming it *axkit.conf*. Then, add the following to the bottom of your *httpd.conf* file:

```
# AxKit Setup

Include conf/axkit.conf
```

You now need to edit the new *axkit.conf* file to match the XML processing libraries that you installed earlier by uncommenting the AxAddStyleMap directives that correspond to tools you chose. For example, if you installed libxslt and XML::LibXSLT, you would uncomment the AxAddStyleMap directive that loads AxKit's interface to LibXSLT. Example 2-1 helps to clarify this.

Example 2-1. Sample axkit.conf fragment

```
# Load the AxKit core.

PerlModule AxKit



# Associates Axkit with a few common XML file extensions

AddHandler axkit .xml .xsp .dkb .rdf



# Uncomment to add XSLT support via XML::LibXSLT

# AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT



# Uncomment to add XSLT support via Sablotron

# AxAddStyleMap text/xsl Apache::AxKit::Language::Sablot



# Uncomment to add XPathScript Support

# AxAddStyleMap application/x-xpathscript Apache::AxKit::Language::XPathScript



# Uncomment to add XSP (eXtensible Sever Pages) support

# AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP
```

The one hard-and-fast rule about configuring AxKit is that the PerlModule directive that loads the AxKit core into Apache via *mod_perl must* appear at the top lexical level of your *httpd.conf* file, or one of the files that it includes. All other AxKit configuration directives may appear as children of other configuration directive blocks in whatever way best suits your server policy and application needs, but the PerlModule AxKit line *must* appear only at the top level.

2.5 Testing the Installation

AxKit's distribution comes with a fairly complete test suite that typically runs as part of the installation process. Running the *make test* command in the root of the AxKit source directory fires up a new instance of the Apache server on an alternate port with AxKit enabled. It then examines the output of a series of test requests made to that instance that exercise various aspects of AxKit's functionality. *make test* runs automatically by default if you are installing AxKit via the CPAN shell. If all test scripts pass during the *make test* process, you can be sure that you have a working AxKit installation and are ready to proceed.

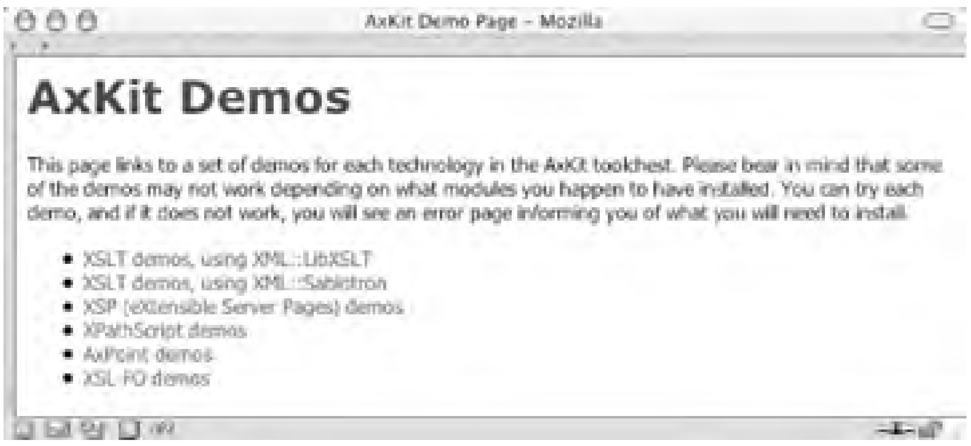
In addition to the automated test suite, AxKit comes with a set of demonstration files that you can also use to test your new installation. To install the demo, copy the *demo* directory and its contents from the root of the AxKit distribution into an appropriate directory to which you have write access. The configuration file in the *demo* directory presumes that you will copy the *demo* directory into */opt/axkit*. So if you choose another location, be sure to edit *all* paths in the demo's *axkit.conf* file to reflect your choice.

Before the demo will work, you need to include the *axkit.conf* contained in the new demo directory into your server's *httpd.conf* file. For example, if you installed the demo in */opt/axkit* (again, the default), you would add the following:

```
# AxKit Demo

Include /opt/axkit/demo/axkit.conf
```

Start (or stop and restart) the Apache server and point a browser to *http://localhost/axkit/*. You should see a page congratulating you on your new AxKit installation. This page also presents a number of links that allow you to test AxKit's various moving parts. For example, if you chose to install libxslt and its Perl interface XML::LibXSLT to use as an XSLT processor, you would click on the XSLT demos, using the XML::LibXSLT link to verify that AxKit works and is configured properly to those libraries to transform XML documents, as shown in Figure 2-1.

Figure 2-1. Proof of a successful demo AxKit installation



AxKit Demos

This page links to a set of demos for each technology in the AxKit toolchest. Please bear in mind that some of the demos may not work depending on what modules you happen to have installed. You can try each demo, and if it does not work, you will see an error page informing you of what you will need to install.

- XSLT demos, using XML::LibXSLT
- XSLT demos, using XML::Sablotron
- XSP (eXtensible Server Pages) demos
- XPathScript demos
- AxPoint demos
- XSL-FO demos

If you receive an error when you click on one of the demo links, verify that you have the associated libraries for that demo installed. If you go back and install any processor that AxKit supports, there is no need to reinstall the demo. Just reload the demo index and click on the appropriate link to verify that the new libraries work. You must, however, stop and start Apache (not just restart it) for AxKit to pick up the new interfaces.

If all goes as expected, congratulations. You have installed a working version of AxKit and are ready to get down to business.

2.6 Installation Troubleshooting

As I mentioned in this chapter's introduction, AxKit's core consists largely of code that glues other things together. In practice, this means that most errors encountered while installing AxKit are due to external dependencies that are missing, broken, out of date, or invisible to AxKit's Makefile. Including a complete list of various errors that may be encountered among AxKit's many external dependencies is not realistic here. It would likely be outdated before this book is on the shelves. In general, though, you can use a number of compile-time options when building AxKit. They will help you diagnose (and in many cases, fix) the cause of the trouble. AxKit's *Makefile.PL* recognizes the following options:

`DEBUG=1`

This option causes the Makefile to produce copious amounts of information about each step of the build process. Although wading through the sheer amount of data this option produces can be tedious, you can diagnose most installation problems (missing or unseen libraries, etc.) by setting this flag.

`NO_DIRECTIVES=1`

This option turns off AxKit's apache configuration directives, which means you must set these via Apache's `PerlSetVar` directives instead. Use this option only in extreme cases in which AxKit's custom configuration directives conflict with those of another Apache extension module. (These cases are very rare, but they do happen.)

`EXPAT_OPTS=" . . . "`

This option is relevant only if you do not have the Expat XML parser installed and decide to install it when installing AxKit. This argument takes a list of options to be passed to libexpat's *./configure* command. For example, `EXPAT_OPTS="--prefix=/usr"` installs libexpat in */usr/lib*, rather than the default location.

`LIBS="-L/path/to/somelib -lsomelib"`

This option allows you to set your library search path. It is primarily useful for pointing the Makefile to external libraries that you are sure are installed but, for some reason, are being missed during the build process.

`INC="-I/path/to/somelib/include"`

This option is like `LIBS`, but it sets the include search path.

2.6.1 Where to Go for Help

If you get stuck at any point during the installation process, do not despair. There are still other resources available to help you get up and running. In addition to this book, there are other sources of AxKit documentation, as well as a strong AxKit user community that is willing and able to help.

2.6.1.1 Installed AxKit documentation

Most Perl modules that comprise the AxKit distribution include a level of documentation. In many cases, these documents are quite detailed. You can access this information using the standard *perldoc* utility typically installed with Perl itself. Just type *perldoc <modulename>*, in which `<modulename>` is the package name of the module that you want to read the docs from. The following list provides a general overview of the information you can find in the various modules.

AxKit

The documentation in *AxKit.pm* provides a brief overview of each AxKit configuration directive, including simple examples.

Example: *perldoc AxKit*

*Apache::AxKit::Language::**

The modules in this package namespace provide support for the various XML processing and transformation languages such as XSLT, XSP, and XPathScript.

Example: *perldoc Apache::AxKit::Language::XSP* provides an XSP language reference.

*Apache::AxKit::Provider::**

The modules in this namespace provide AxKit with the ability to fetch and read the sources for the XML content and stylesheets that it will use when serving the current request.

Example: *perldoc Apache::AxKit::Provider::Filter* shows the documentation for a module that allows an upstream PerlHandler (such as Apache::ASP or Mason) to generate content.

*Apache::AxKit::Plugin::**

Modules in this namespace provide extensions to the basic AxKit functionality.

Example: *perldoc Apache::AxKit::Plugin::Passthru* offers documentation for the Passthru plug-in, which allows a "source view" of the XML document being processed based on the presence or absence of a specific query string parameter.

*Apache::AxKit::StyleChooser::**

The modules in this namespace offer the ability to set the name of a preferred transformation style in environments that provide more than one way to transform documents for a given media type.

Example: *perldoc Apache::AxKit::StyleChooser::Cookie* shows the documentation for a module that allows stylesheet transformation chains to be selected based on the value of an HTTP cookie sent from the requesting client.

Additional user-contributed documentation is also available from the AxKit project's web site at http://axkit.org/. Not only does the project site offer several useful tutorials, it also provides a user-editable Wiki that often contains the latest platform-specific installation instructions, as well as many other AxKit tips, tricks, and ideas.

2.6.1.2 Mailing lists

The AxKit project sports a lively and committed user base with lots of friendly folks who are willing to help. Even if you are not having trouble, I highly recommend joining the axkit-users mailing list. The amount of traffic is modest, the signal-to-noise ratio is high, and topics range from specific AxKit installation questions to general discussions of XML publishing best practices. You can subscribe online by visiting http://axkit.org/mailinglist.xml or by sending an empty email message to mailto:axkit-users-subscribe@axkit.org.

You can find browsable archives of axkit-users at:

- http://axkit.org/cgi-bin/ezmlm-cgi/3
- http://www.mail-archive.com/axkit-users@axkit.org/index.html

Topics relating specifically to AxKit development are discussed on the axkit-devel list. Generally, you should post most questions, bug reports, patches, etc., to axkit-users. If you want to contribute to the AxKit codebase, then axkit-devel is the place for you. You can subscribe to the development list by sending an empty message to mailto: axkit-dev-subscribe@xml.apache.org.

In addition to the mailing lists, the AxKit community also maintains an #axkit IRC channel for discussing general AxKit topics. The IRC server hosting the channel changes periodically, so check the AxKit web site for details.

Chapter 3. Your First XML Web Site

With AxKit installed, you can begin putting it though its paces. In this chapter, we create a simple XML-based web site. Along the way, I will introduce AxKit's facilities for how to apply stylesheets to transform data marked up in XML into a commonly used delivery format, how to combine XML from different sources, and how to configure an alternate style transformation to deliver the same XML content in a different format in response to data received from the requesting client.

3.1 Preparation

By design, XML processing tools are less forgiving about what they accept than the HTML browsers that you may be used to working with. Omitting a closing tag when creating an element in an HTML page, for example, may cause an undesirable result when the page is rendered, but the browser usually tries to recover gracefully and render *something* for you to see. In contrast, omitting an end tag when creating an element in a document that an XML parser will consume results in a fatal well-formedness error, and no such recovery is possible. In the context of AxKit (in which all XML processing happens on the server), this means that if you pass in a bad document, AxKit sends no content to the client. At best, you see an error message that indicates where things went wrong. To avoid frustration, take a little time to familiarize yourself with the XML processing tools available to you. At the very least, investigate how the XML parser you installed can be used from the command line to verify a document's well-formedness and validity. Being able to catch bad documents going in reduces the overall number of potentially user-visible errors. The ability to verify that your content and stylesheets are at least syntactically correct can make finding the cause of an error easier.

Even more than with a static HTML-based site, starting with a good directory structure is key to creating an easy-to-maintain XML-based site. The time and labor-saving benefits of having predictable paths for images, CSS stylesheets, etc. also apply to the files associated with XML processing. It's a good idea to get in the habit of creating a *stylesheets* (or similarly named) directory at the base of the host's DocumentRoot when you start a new project.

If you installed the AxKit demonstration site or included the sample *axkit.conf* in your main Apache configuration (covered in Section 2.4), you do not need to alter the web server's configuration at all. If not, follow the directions there, or add the following lines to the web server's *httpd.conf,* and stop and restart the server before proceeding:

```
PerlModule AxKit
AddHandler axkit .xml .xsp .dkb .rdf


AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT
AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP
AxAddStyleMap application/x-xpathscript Apache::AxKit::Language::XPathScript
```

These directives load the AxKit core into Apache, set up the required Language transformation processors, and configure Apache to process all files that end in *.xml*, *.xsp*, *.dkb*, and *.rdf* with AxKit. With an appropriate directory structure and configuration in place, you can move on to creating the XML documents that you want to publish.

3.2 Creating the Source XML Documents

Often, many benefits of using an XML publishing framework such as AxKit become obvious only later in a project's life (e.g., the ability to easily add new heavy-duty features to an existing site, or the power to completely change the look and feel of an entire site without touching its content). Given this, any examples you may choose for this introduction will surely fall short of illustrating AxKit's real power. Accepting the notion that the task at hand is a bit absurd frees you to have a little fun with it while still learning the basics. Let's run with the absurdity, and imagine that you are charged with the task of publishing a small site on the very silly subject of cryptozoology.

Cryptozoology (literally, *the study of hidden animals*) is concerned with the gathering and analysis of data related to animals that are frequently reported by local residents or found in popular folklore, but whose existence the scientific community has not formally recognized. Familiar examples include the Yeti, Loch Ness Monster, and Mokele-Mbembe.

The first document for your site, *cryptozoo.xml*, contains a list of cryptozoological species (called *cryptids* by insiders). (See Example 3-1.)

Example 3-1. cryptozoo.xml

```
<?xml version="1.0"?>
<cryptids>
  <species>
    <name>Jackalope</name>
    <habitat>Western North America</habitat>
    <description>
      <para>
        Similar to the Bavarian rauarckl (stag-hare), the
        North American Jackalope resembles a large jackrabbit
        with small, deer-like antlers. This vicious
        carnivore is frequently mistaken for common rabbits or hares
        suffering from <italic>papillomatosis</italic> (a condition
        that produces horn-like growths on the head in those species).
      </para>
    </description>
  </species>
  <species>
    <name>Dahut</name>
    <habitat>French Alps</habitat>
    <description>
      <para>
        A shy relative of the Alpine deer, the Dahut has
        adapted to the challenges of its mountainous habitat by
        growing legs that are considerably longer on one side
        of its body. While this asymmetrical limb configuration allows
        for level grazing on steep grades, it leaves the unfortunate
        creature unable to reverse its course. Local hunters exploit
        this weakness by sneaking up behind the Dahut and either

        whistling softly or crying "Dahut!"; when the startled

        creature turns to face its assailant, it finds its

        longer legs on the wrong side and it tumbles to it doom.

      </para>

    </description>

  </species>

  <!--  . . . more species here -->

</cryptids>
```

No cryptozoology site worth its salt is complete without a list of cryptid sightings. Your second XML document, creatively named *cryptid_sightings.xml*, contains just that. (See Example 3-2.)

Example 3-2. cryptid_sightings.xml

```
<?xml version="1.0"?>

<sightings>

  <sighting>

    <species>Jersey Devil</species>

    <location>Bordentown, NJ</location>

    <date>Autumn, 1816</date>

    <description>

      <p>

        A Jersey Devil was reportedly seen

        by Joseph Bonaparte, former King of Spain

        and brother of Napoleon, while hunting in

        the woods near Bordentown, New Jersey.

      </p>

    </description>

    <witnesses>

      <name>Joseph Bonaparte</name>

    </witnesses>

  </sighting>

  <sighting>

    <species>Snipe</species>

    <location>Phelan, CA</location>

    <date>June 2002</date>

    <description>

      <p>

        The Phelan Phine Snipe Hunters Association
        celebrated the opening of this year's Snipe
        season. Unfortunately, the entire
        photographic record of the event was ruined
        during a nasty "keg stand" incident in
        the beer tent after the hunt.
      </p>
    </description>
    <witnesses>
      <name>Jason Nugall</name>
      <name>William Q. Rozborne</name>
    </witnesses>
  </sighting>
</sightings>
```

You need one last XML document: a small file that captures the filenames of the two other documents in the site. You will use this bit of metadata to create the navigation in the final result delivered to the requesting HTML browser, so the name *nav.xml* seems appropriate. (See Example 3-3.)

Example 3-3. nav.xml

```
<?xml version="1.0"?>
<links>
  <a href="cryptozoo.xml">Species</a>
  <a href="cryptid_sightings.xml">Sightings</a>
</links>
```

3.3 Writing the Stylesheet

So far, you have three XML documents that contain three very different, but randomly overlapping, grammars. (The species and name elements appear in different roles in the two main content documents.) Your goal is to make this information available on the Web to HTML browsers. You want to reach the widest possible audience, and that means maintaining the lowest possible expectations of the requesting client's capabilities. That is, you cannot rely on everyone who wants to read your pages having a thoroughly modern browser capable of doing appropriate client-side transformations to your XML documents via CSS or XSLT. You must deliver basic HTML if you expect your data to be widely accessible.

With this in mind, you need a way to transform the disparate data structures contained in each of your XML documents into the unified grammar of simple HTML.That's where AxKit's transformational languages and stylesheets enter the picture. AxKit offers many ways to transform XML data. (We will examine the merits of many of these in later chapters.) In this example, we examine how you can transform your cryptozoology documents into HTML using two of the more popular transformation languages: XSLT and XPathScript.

I will save the examination of the lower-level details of these languages for later. At this point, it suffices to understand that both XSLT and XPathScript offer a declarative syntax that provides a way to create new documents by applying transformations to all or some of the elements, attributes, and other content that an existing XML document contains.

3.3.1 Using XSLT

Rather than taking small steps through the XSLT stylesheet, I present it here in one block to give you an idea of what a full, working stylesheet looks like. (See Example 3-4.) Do not worry if much of it seems foreign; we will look at the syntactic elements of XSLT in more detail in Chapter 5.

As you read through the stylesheet, keep in mind that:

- An XSLT stylesheet itself is an XML document.
- Transformation rules are applied based on the properties of the various parts of the source XML document (element and names, relationships between elements, etc.).
- Template rules are created to match all elements of the grammars found in your XML documents, so the same stylesheet can be used to transform both the list of species and the list of sightings.

Example 3-4. cryptozoo.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet

  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

  version="1.0"

  >

<xsl:output

     method="html" encoding="ISO-8859-1"

/>


<xsl:template match="/">

<html>

  <head><title>Cryptozoology Pages</title>
  <link rel="stylesheet" type="text/css" href="crypto.css"/>
  </head>
  <body>
  <div class="header">
  <h1>My Cryptozoology Pages</h1>
  </div>
  <div class="nav">
    <xsl:apply-templates select="document('nav.xml', /)/*"/>
  </div>
  <div class="content">
    <xsl:apply-templates/>
  </div>
  </body>
</html>
</xsl:template>

<!-- top-level templates -->
<xsl:template match="animals">
  <p>
    Today's rural legend is tomorrow's newly
    discovered species.
  </p>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="sightings">
  <p>
    Here is a list of sightings.
  </p>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="animals/species">
<div class="species">
  <xsl:apply-templates/>
</div>
</xsl:template>

<xsl:template match="sighting">
```

```
<div class="sighting">
  <xsl:apply-templates/>
</div>
</xsl:template>

<xsl:template match="species/name">
<h2>
  <xsl:apply-templates/>
</h2>
</xsl:template>

<xsl:template match="species/habitat">
  <p>
    <i>Habitat:</i>
    <xsl:text> </xsl:text>
    <xsl:value-of select="."/>
  </p>
  <xsl:apply-templates select="*"/>
</xsl:template>

<xsl:template match="sighting/location|sighting/species|sighting/date">
  <div class="subheading">
    <i><xsl:value-of select="name( )"/>:</i>
    <xsl:text> </xsl:text>
    <xsl:value-of select="."/>
  </div>
  <xsl:apply-templates select="*"/>
</xsl:template>

 <xsl:template match="witnesses">
   <div>
     <i>witnesses:</i>
     <xsl:text> </xsl:text>
     <xsl:for-each select="name">
       <xsl;value-of select="." />
       <xsl:if test="position( ) != last( )">, </xsl:if>
     </xsl:for-each>
   </div>
```

```
</xsl:template>


<xsl:template match="witnesses/name">
  <xsl:text> </xsl:text>
  <xsl:value-of select="."/>
</xsl:template>


<xsl:template match="para|p">
<p>
  <xsl:apply-templates/>
</p>
</xsl:template>


<xsl:template match="italic">
<i>
  <xsl:apply-templates/>
</i>
</xsl:template>


<xsl:template match="a">
  <xsl:copy-of select="."/><br/>
</xsl:template>


</xsl:stylesheet>
```

3.3.2 Using XPathScript

Compare the previous XSLT stylesheet, which transforms your cryptozoology documents into HTML, with the one below, written in XPathScript. I will leave the syntactic details for Chapter 6; however, as you read through the stylesheet, note that most moving parts are written in Perl and separated from the larger template document using `<%` and `%>` as delimiters.

```
<%
# declarative templates
$t->{'p'}{pre} = '<p>';
$t->{'p'}{post} = '</p>';


$t->{'para'}{pre} = '<p>';
$t->{'para'}{post} = '</p>';


$t->{'animals'}{pre} = qq|
  <p>
    Today's rural legend is tomorrow's newly
    discovered species.
  </p>
|;

$t->{'sightings'}{pre} = qq|
  <p>
    Here is a list of sightings.
  </p>
|;

$t->{'sighting'}{pre} = '<div class="sighting">';
$t->{'sighting'}{post} = '</div>';


$t->{'habitat'}{pre} =  '<p><i>Habitat:</i> ';
$t->{'habitat'}{post} =  '</p>';



$t->{'location'}{pre} = '<div class="subheading"><i>location: </i>';

$t->{'location'}{post} = '</div>';

$t->{'date'}{pre} = '<div class="subheading"><i>date: </i>';
$t->{'date'}{post} = '</div>';

$t->{'witnesses'}{pre} = '<div><i>witnesses: </i>';
$t->{'witnesses'}{post} = '</div>';

$t->{'a'}{showtag} = 1;

# testcode templates for like-named elements
$t->{'name'}{testcode} = sub {
  my ($node, $t) = @_;
  if (findnodes('parent::species', $node)) {
      $t->{pre} = '<h2>';
      $t->{post} = '</h2>';
  }
  return 1;
};

$t->{'species'}{testcode} = sub {
  my ($node, $t) = @_;
  if (findnodes('parent::animals', $node)) {
     $t->{pre} = '<div class="species">';
     $t->{post} = '</div>';
  }
  else {
    $t->{pre} = '<div class="subheading"><i>species: </i>';
    $t->{post} = '</div>';
  }
  return 1;
};
%>

<html>
  <head><title>Cryptozoology Pages</title></head>
  <link rel="stylesheet" type="text/css" media="screen" href="crypto.css"/>
  <body>
  <div class="header">
  <h1>My Cryptozoology Pages</h1>
  </div>
  <div class="nav">
    <%= apply_templates( "document('nav.xml')" ) %>
  </div>
  <div class="content">
    <%= apply_templates( ) %>
  </div>
  </body>
</html>
```

Do not be overwhelmed. Remember that most sites typically use *either* XSLT or XPathScript and only rarely both, so you need not try to take in both at once. These duplicated examples only intend to show that with AxKit, as with Perl, there is always more than one way to do it. You are free to choose the tools and techniques that suit you best.

3.4 Associating the Documents with the Stylesheet

AxKit offers a variety of configuration options for associating documents with its various language processors. Chapter 4 covers each in detail. In Example 3-5, you create an *.htaccess* file in the same directory as your XML documents. It defines a default style for AxKit to use when processing documents in this directory.

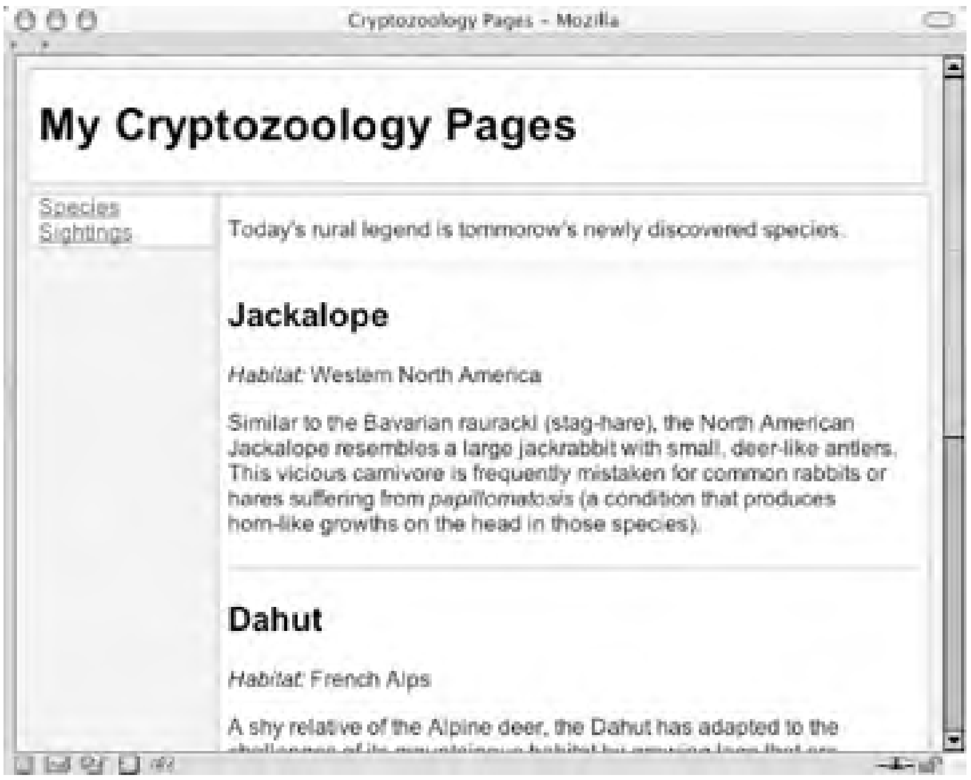
Example 3-5. A simple .htaccess file

```
<AxStyleName "#default">

  AxAddProcessor text/xsl stylesheets/cryptozoo.xsl

</AxStyleName>
```

Pay attention to the arguments passed to the AxAddProcessor directive. The first is the MIME type that AxKit examines to decide which language processor modules to use, and the second is the DocumentRoot-relative path to the stylesheet that will be passed to that language processor to transform your XML documents. If you want to use your XPathScript stylesheet rather than the XSLT, you would use AxAddProcessor application/x-xpathscript stylesheets/cryptozoo.xps instead. This processor definition is wrapped in an AxStyleName block. This directive block, in turn, combines the processor definitions it contains into a single "named style" that a StyleChooser or other plug-in can select at runtime. By giving this style the special name #default, you are configuring AxKit to use this style as a fallback if no other style is explicitly selected.

It's time to fire up a web browser and check the results of your work. A request to http://myhost.tld/cryptozoo.xml yields what is shown in Figure 3-1.

Figure 3-1. cryptozoo.xml rendered as HTML



My Cryptozoology Pages

Today's rural legend is tommorow's newly discovered species.

Jackalope

Habitat: Western North America

Similar to the Bavarian rauracki (stag-hare), the North American Jackalope resembles a large jackrabbit with small, deer-like antlers. This vicious carnivore is frequently mistaken for common rabbits or hares suffering from *papillomatosis* (a condition that produces horn-like growths on the head in those species).

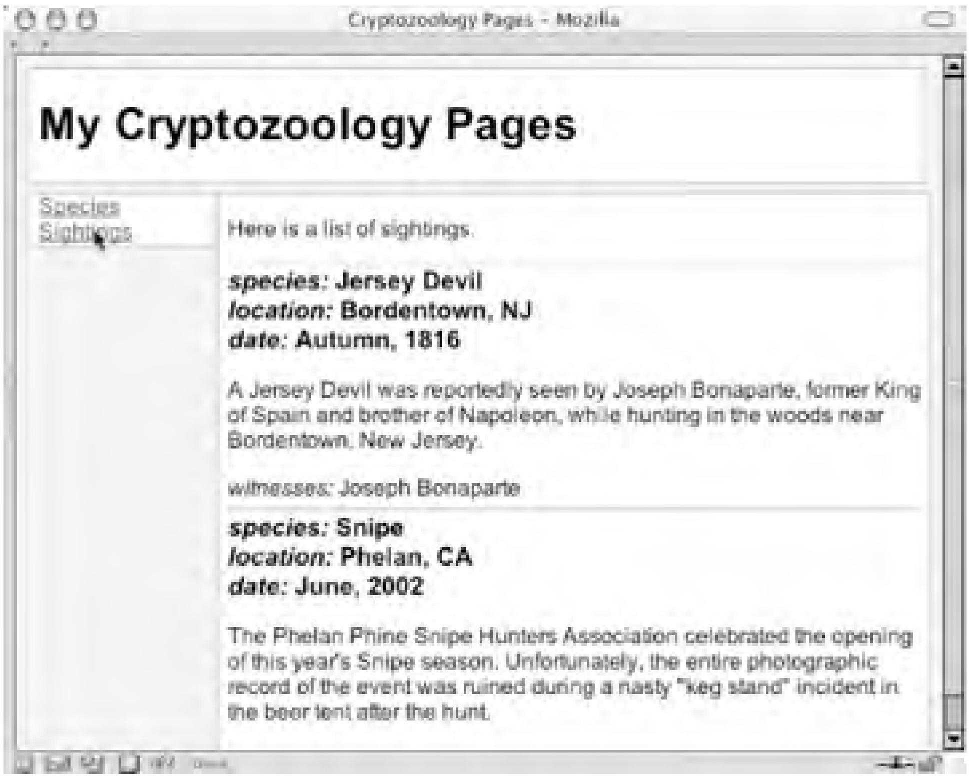
Dahut

Habitat: French Alps

A shy relative of the Alpine deer, the Dahut has adapted to the

Clicking on the Sightings link reveals what is shown in Figure 3-2.

Figure 3-2. cryptid_sightings.xml rendered as HTML



My Cryptozoology Pages

Species
Sightings

Here is a list of sightings.

species: **Jersey Devil**
location: **Bordentown, NJ**
date: **Autumn, 1816**

A Jersey Devil was reportedly seen by Joseph Bonaparte, former King of Spain and brother of Napoleon, while hunting in the woods near Bordentown, New Jersey.

witnesses: Joseph Bonaparte

species: **Snipe**
location: **Phelan, CA**
date: **June, 2002**

The Phelan Phine Snipe Hunters Association celebrated the opening of this year's Snipe season. Unfortunately, the entire photographic record of the event was ruined during a nasty "keg stand" incident in the beer tent after the hunt.

3.5 A Step Further: Syndicating Content

You have reached your initial goal of publishing your XML documents for consumption by HTML browsers on the Web using AxKit. Even if that were all you ever wanted to do, you still made a clear division between the content you will maintain and the way in which it is presented. Among other benefits, you can now redesign the look and feel of pages sent to the client without touching content documents. Don't worry about clobbering or obscuring essential information just to change the way it renders in a browser. Similarly, using a custom XML grammar for your content means that the documents themselves can unambiguously define the intended roles of the data they contain, rather than the way that data may be represented on the visual medium of an HTML browser. This makes reusing the data for other purposes a lot easier.

To understand the practical benefits of separating content from presentation, suppose that your list of cryptid sightings becomes wildly popular on the Web. People start asking for a way to put links to the newly reported sighting on their own cryptozoology sites. You could tell them to screen-scrape the HTML list. Instead, you decide to be a good information-sharing citizen and make the list available as an RSS syndication feed. To achieve this, the first thing you need is a stylesheet that transforms the list of cryptid sightings to RSS, in addition to the one you already have that transforms the data into HTML.

For those who may not be familiar with it, RSS (RDF Site Summary, Rich Site Summary, or Really Simple Syndication, depending on whom you ask) is a popular XML grammar used for syndicating online content, especially news headlines. Most weblogs use RSS as the means to both publish content and share links with other bloggers, and many weblog tools store their data natively as RSS. (See Example 3-6.) For more information about RSS and some of its more creative uses, see Ben Hammersley's *Content Syndication with RSS* (O'Reilly).

Example 3-6. cryptidsightings_rss.xsl

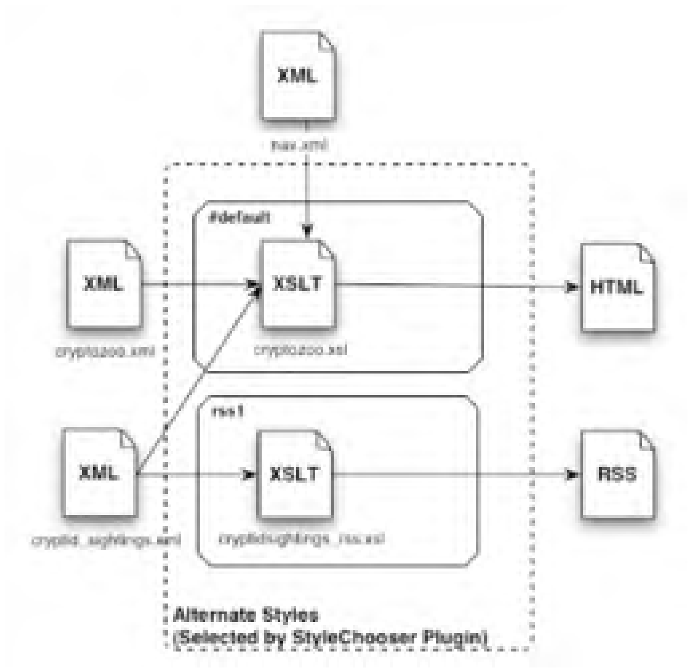
```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://purl.org/rss/1.0/"
  version="1.0"
>

<xsl:variable
     name="this_url"
     select="'http://myhost.tld/cryptosightings.xml'"
/>

<xsl:template match="/">
<rdf:RDF>
  <channel rdf:about="{$this_url}">
    <title>My Cryptozoology Pages- Sightings</title>
    <link>
      <xsl:value-of select="$this_url"/>
    </link>
    <description>
      The latest sightings of animals that do not officially exist.
    </description>
    <items>
      <rdf:Seq>
        <xsl:for-each select="/sightings/sighting">
          <rdf:li rdf:resource="{$this_url}#{position( )}"/>
        </xsl:for-each>
      </rdf:Seq>

    </items>
  </channel>
  <xsl:apply-templates select="/sightings/sighting"/>
</rdf:RDF>
</xsl:template>

<xsl:template match="sighting">
<item rdf:about="{$this_url}#{position( )}">
  <link>
    <xsl:value-of select="concat($this_url,'#', position( ))"/>
  </link>

  <title>
    <xsl:value-of select="species"/> Sighting - <xsl:value-of select="date"/>
  </title>
  <description>
    <xsl:apply-templates select="description"/>
  </description>
</item>
</xsl:template>

<xsl:template match="description|p">
  <xsl:value-of select="normalize-space(.)" />
</xsl:template>

</xsl:stylesheet>
```

Before you can see this stylesheet in action, you need to add a couple of configuration directives to the *.htaccess* file you created earlier:

```
<Files cryptid_sightings.xml>

  <AxStyleName rss1>

    AxAddProcessor text/xsl stylesheets/cryptidsightings_rss.xsl

  </AxStyleName>

  AxAddPlugin Apache::AxKit::StyleChooser::QueryString

</Files>
```

The AxStyleName block creates a named style called rss1. The AxAddProcessor it contains associates that named style with the RSS stylesheet you just created. The AxAddPlugin directive, in this case, tells AxKit to use additional logic to examine the query string sent from the requesting client for a parameter named style. If it finds one, and the value of that parameter matches one of the named styles configured for that URI, it uses the stylesheets contained in that named style to transform the XML source document. Here, this means that a request to http://myhost.tld/cryptid_sightings.xml?style=rss1 returns the list of species sightings processed by your RSS output stylesheet, not the default style you created earlier. (See Figure 3-3.)

Figure 3-3. Cryptozoology site-processing diagram



Example 3-7 shows the result for a request for the *cryptid_sightings.xml* file as an RSS document.

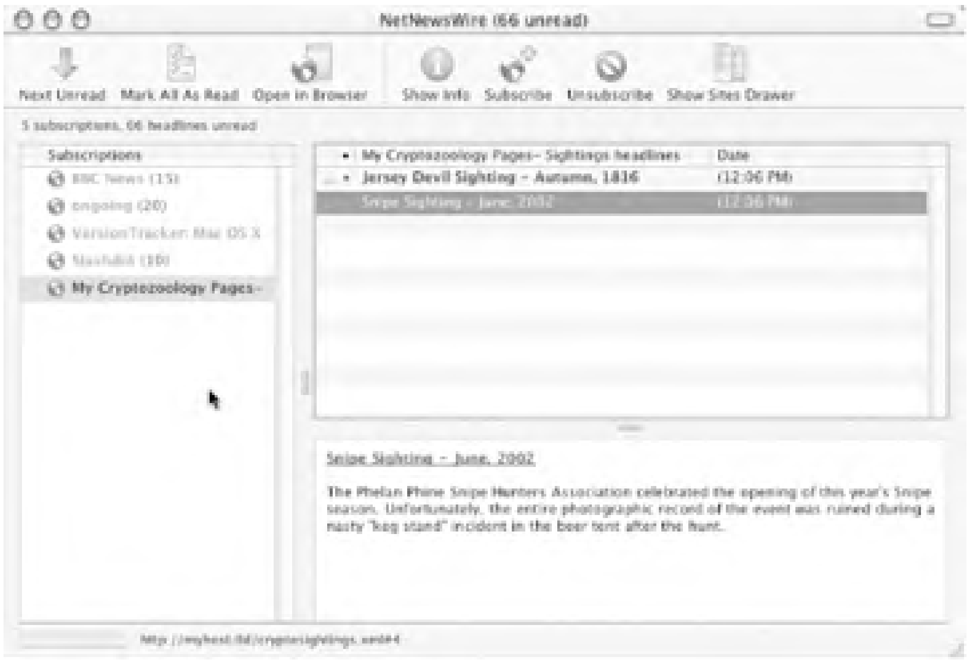
Example 3-7. Cryptid sightings delivered as an RSS feed

```
<?xml version="1.0"?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

xmlns="http://purl.org/rss/1.0/">

  <channel rdf:about="http://myhost.tld/cryptosightings.xml">

    <title>My Cryptozoology Pages- Sightings</title>

    <link>http://myhost.tld/cryptosightings.xml</link>

    <description>
```

```
      The latest sightings of animals that do not officially exist.
    </description>
    <items>
      <rdf:Seq>
        <rdf:li rdf:resource="http://myhost.tld/cryptosightings.xml#1"/>
        <rdf:li rdf:resource="http://myhost.tld/cryptosightings.xml#2"/>
      </rdf:Seq>
    </items>
  </channel>
  <item rdf:about="http://myhost.tld/cryptosightings.xml#1">
    <link>http://myhost.tld/cryptosightings.xml#1</link>
    <title>Jersey Devil Sighting - Autumn, 1816</title>
    <description>
        A Jersey Devil was reportedly seen
        by Joseph Bonaparte, former King of Spain
        and brother of Napoleon, while hunting in
        the woods near Bordentown, New Jersey.
      </description>
  </item>
  <item rdf:about="http://myhost.tld/cryptosightings.xml#2">
    <link>http://myhost.tld/cryptosightings.xml#2</link>
    <title>Snipe Sighting - June, 2002</title>
    <description>
        The Phelan Phine Snipe Hunters Association
        celebrated the opening of this year's Snipe
        season. Unfortunately, the entire
        photographic record of the event was ruined
        during a nasty "keg stand" incident in
        the beer tent after the hunt.
      </description>
  </item>
</rdf:RDF>
```

Figure 3-4 shows a request to that same URL rendered in the NetNewsWire RSS client.

Figure 3-4. Information rendered as RSS in NetNewsWire



5 subscriptions, 66 headlines unread

My Cryptozoology Pages- Sightings headlines	Date
Jersey Devil Sighting - Autumn, 1816	(12:06 PM)
Snipe Sighting - June, 2002	(12:06 PM)

Snipe Sighting - June, 2002

The Phelan Phine Snipe Hunters Association celebrated the opening of this year's Snipe season. Unfortunately, the entire photographic record of the event was ruined during a nasty "keg stand" incident in the beer tent after the hunt.

http://myhost.tld/cryptosightings.xml#4

Now your friends in the cryptozoology community can add your list of sightings to their list of daily news feeds by pointing their RSS viewer or other client to http://myhost.tld/cryptid_sightings.xml?style=rss1, while casual web surfers get the default HTML version. Keep in mind, too, that the query string StyleChooser is only one way to dynamically select the preferred style. Using other plug-ins, you could just as easily examine the connecting client's User-Agent header, or another aspect of the request, to get the same effect. Remember, too, that XSLT and XPathScript are only two transformative Language modules that AxKit supports; there are several others, each with its own unique strengths and weaknesses.

As promised, your first example site is a bit silly (Dahuts, indeed), but let's examine exactly what you have done:

- You used stylesheets to transform data marked up in custom XML grammars into a commonly used delivery format (HTML).
- You combined XML from different resources (the content documents and the navigation metadata) to meet your application's needs.
- You mixed standard Apache configuration blocks with AxKit's custom directives to select an alternative style transformation that delivered the same XML content in a different format in response to data received from the requesting client.

Taken together, these points represent the basic foundation of publishing XML documents with AxKit. You can apply the patterns you learned here to most, if not all, of your future XML publishing projects. However, your little cryptids' site offers only a glimpse of the flexibility and power that AxKit offers. The following chapters build on these basic concepts to show how you can use AxKit to create sophisticated, dynamic, and feature-rich sites.

Chapter 4. Points of Style

Style is an important concept in the AxKit world. Much of the value that AxKit adds as an XML publishing and application server lies in the flexibility and ease with which documents can be associated with one or more sets of transformations that can be selected dynamically in response to a condition. From more common web-publishing tasks, such as vendor co-branding and user customization, to more advanced dynamic data-oriented applications, the key to developing clean, manageable sites with AxKit lies in learning how to apply the appropriate styles to the content to meet the specific need. In this chapter, I introduce AxKit's basic styling configuration options and show how these options can be combined to create sophisticated, responsive sites.

4.1 Adding Transformation Language Modules

Before you can begin associating documents with the stylesheets that will be used to transform them, you must tell AxKit which lower-level processors to use to perform those transformations. In AxKit, access to various transformative processors is provided by its Language modules. Many of these modules create a bridge between AxKit and an existing XML processing tool. For example, the Apache::AxKit::Language::LibXSLT module allows AxKit access to Perl's XML::LibXSLT interface and, hence, to the Gnome project's XSLT processing library, `libxslt`. Other Language modules, such as AxKit's implementation of eXtensible Server Pages, ApacheAxKit::Language::XSP, are unique to AxKit and implement both the interface that allows it to be added to the AxKit processing chain and the code that actually processes the XML content. The core AxKit distribution contains several such Language modules:

Apache::AxKit::Language::LibXSLT

Adds support for the Gnome Project's `libxslt` processor; used to transform documents with XSLT

Apache::AxKit::Language::Sablot

An alternate XSLT transformer using the Sablotron XSLT processor from The Ginger Alliance

Apache::AxKit::Language::XPathScript

Adds support for a more Perlish alternative to XSLT, XPathScript

Apache::AxKit::Language::XSP

Provides an interface to AxKit's implementation of the eXtensible Server Pages (XSP)

Apache::AxKit::Language::SAXMachines

Provides an AxKit interface to Barrie Slaymaker's popular XML::SAX::Machines Perl module, which offers an easy way to set up chains of SAX Filters to transform XML content

Apache::AxKit::Language::PassiveTeX

Offers XSL-FO (XSL Formatting Objects) support for generating PDF documents from FO sources via PassiveTeX suite's `pdfxmltex` utility

Apache::AxKitLanguage::HtmlDoc

Converts XHTML documents to PDF, also using PassiveTeX

Several other Language modules come with AxKit, and still others are available via the Comprehensive Perl Archive Network (CPAN). For example, many of Perl's more popular templating packages (the Template Toolkit, Petal, etc.) are also available as AxKit Language processors. However, while AxKit or an individual CPAN distribution may provide the Language modules needed to allow AxKit to access a given type of processor, they do not usually install any lower-level tools with which the modules may interface. For example, if you intend to use the Apache::AxKit::LibXSLT module for XSLT processing, you must be sure to first install the XML::LibXML and XML::LibXSLT Perl modules as well as the `libxml2` and `libxslt` C libraries that it requires.

Once you decide which kinds of transformations to use for your site, the Language module is added to AxKit via the `AxAddStyleMap` directive. This directive expects two arguments: a MIME type that uniquely identifies the language, and the Perl package name of the module that implements that language's processor:

```
<AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT
```

This associates the MIME type `text/xsl` with the Perl package name of the LibXSLT Language module, Apache::AxKit::Language::LibXSLT. This mapping is important since the MIME type assigned here is used as the identifier to associate the site's stylesheets with the appropriate language processor that will be used to apply those stylesheets to the source content. So, given the above example, all style definitions that declare the type `text/xsl` will be applied to the source XML using the Apache::AxKit::Language::LibXSLT language module.

Although AxAddStyleMap directives may appear anywhere in a site's configuration files, these type-to-module mappings are typically meant to apply to a whole site and, therefore, usually appear together at the top level of the host's AxKit configuration:

```
<Directory "/www/sites/myaxkitsite">

  AddHandler axkit .xml .xsp. .xsp .dbk]]>

  <emphasis role="bold">AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT

  AxAddStyleMap application/x-xpathscript Apache::AxKit::Language::XPathScript

  AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP</emphasis>

</Directory>
```

Once the language processors are set up, the style definitions that will control the transformations that AxKit will apply to the content can be added.

4.2 Defining Style Processors

The most basic component used to define which transformations will be applied to a given resource within AxKit is perhaps best termed a *style processor definition*. These definitions indicate a single transformational step (applying an XSLT stylesheet or passing the content through a SAX Filter, for example) in what may be a chain of transformations. In their most basic and typical form, these style definitions declare two bits of crucial information: a MIME type that will be used by AxKit to determine which Language module will be used to transform the content, and a file path to a stylesheet (or other Language-specific argument) that will be used by that Language module to determine how to transform the content. Individual processor definitions may optionally be combined into named style and media groups that can then be selected conditionally, based on a number of factors.

By default, style processors are defined within AxKit in one of two ways: by using AxKit's processor configuration directives or by special stylesheet processing instructions contained in the source documents themselves. (In Chapter 8, you'll learn to create your own way to configure AxKit's styling rules by rolling a custom ConfigReader module, but that's another story.) The following illustrates how to create a simple named style containing a single style definition using AxKit's server configuration directives. The lone processor definition contains the required MIME type and path to the stylesheet that will be applied.

```
<AxStyleName "#default">

   AxAddProcessor text/xsl /path/to/style1.xsl

</AxStyleName>
```

The MIME type used in the processor definition corresponds to the one you associated earlier with the Language::LibXSLT module. The effect of this is that, starting with the source, XML will be transformed by the LibXSLT processor by applying the stylesheet at the location defined by the second argument.

4.2.1 AxKit's Runtime Styling Directives

AxKit offers a small host of runtime configuration directives that can be used to define the styles for a given site. These directives are actually extensions of Apache's configuration syntax and, as such, are added to the usual server configuration files, such as *httpd.conf* or *.htaccess* files. They can be mixed and matched with other Apache directives. A basic familiarity with Apache's runtime configuration syntax is presumed in this section.

Using AxKit's configuration directives, style definitions can be added and applied in all cases or applied conditionally, based on an aspect of the XML document being served (the document's root element name, URI on the server, etc.). A style definition's first argument is the MIME type associated with the language module that will apply the transformation, and the second is the path to the stylesheet that will be applied. Conditional processor definitions add a third, directive-specific argument that defines the rules that govern when and if that processor will be adding to the current processing chain.

Relative stylesheet paths are resolved in the context of the directory that contains the resource being requested, while qualified paths are resolved relative to the host's `DocumentRoot`. However, not all language modules employ external stylesheets. For example, the SAXMachines module expects a quoted, whitespace-separated list of Perl package names that implement SAX Filters instead of a stylesheet path. Others, such as XSP, are self-contained (the source document itself defines the transformations), and the literal string `NULL` is used in place of the stylesheet path.

We will now take a look at each of AxKit's processor definition configuration directives. Keep in mind as you read the details of each directive that these are just the lowest-level building blocks from which you can create sophisticated application- and media-specific processing chains.

4.2.1.1 AxAddProcessor

`AxAddProcessor`, the most basic of the styling configuration directives, unconditionally adds that processor definition to AxKit's processing chain. The processor definition requires two arguments: the MIME type associated with the language processor that will be used to do the transformation, and the path to the stylesheet that will be used by that processor to transform the XML source.

```
AxAddProcessor text/xsl /styles/global.xsl
```

This unconditionally adds the stylesheet */styles/global.xsl* to the processing chain and tells AxKit to use the language module associated with the MIME type `text/xsl` to perform the transformation.

The `AxAddProcessor` directive is commonly used in conjunction with Apache's standard `<Files>`, `<Directory>`, `<Location>`, and similar directives. For example, adding the following to a *.htaccess* at the root level of a site would configure AxKit to apply the stylesheet */styles/docbook_simple.xsl* to all documents in that site that have the file extension *.dkb,* while transforming all documents with *.xml* extension with the global stylesheet from the above example:

```
<Files *.dkb>

  AxAddProcessor text/xsl /styles/docbook_simple.xsl

</Files>



<Files *.xml>

  AxAddProcessor text/xsl /styles/global.xsl

</Files>
```

Transformation chains can be created by adding more than one style processor definition to a given context. The following would configure AxKit to apply the stylesheet *pre_process.xsl* to all documents in the /docs directory and then to apply the stylesheet *main.xsl* to the result of the first transformation:

```
<Directory /docs>

  AxAddProcessor text/xsl /styles/pre_process.xsl

  AxAddProcessor text/xsl /styles/main.xsl

</Directory>
```

AxKit processing definitions, like many Apache directives, are inherited recursively down directory branches. So, a style definition configured for the root of a site will be applied to all documents in that site, unless explicitly overridden.

4.2.1.2 AxAddRootProcessor

The AxAddRootProcessor directive sets up document-to-stylesheet mappings based on the name of the root element in the source XML document. It requires three arguments: the MIME type associated with the language processor that will perform the transformation, the path to the stylesheet that will be applied, and the name of the root element to match against.

The following configures AxKit to apply the stylesheet */styles/docbook_simple.xsl* to all documents in the current scope that have a root element named article:

```
AxAddRootProcessor text/xsl /styles/docbook_simple.xsl article
```

Matching top-level element names in documents employing XML namespaces is supported using the Clarkian Notation, in which the element's namespace URI is wrapped in curly braces { } and prepended to the element's local name.

```
AxAddRootProcessor text/xsl /styles/mydoc.xsl {http://localhost/NS/mydoc}document
```

This would apply the stylesheet */styles/mydoc.xsl* to both of the following documents:

```
<?xml version="1.0"?>

<document xmlns="http://localhost/NS/mydoc">

  <element/>

</document>



<?xml version="1.0"?>

<doc:document xmlns:doc="http://localhost/NS/mydoc">

  <doc:element/>
```

```
</doc:document>
```

The element's namespace prefix is not considered, since it is really only a syntactic shortcut used to bind the given element to the namespace URI to which the prefix is bound. Hence, the following would match *neither* of the above documents:

```
AxAddRootProcessor text/xsl /styles/mydoc.xsl {http://localhost/NS/mydoc}

doc:document
```

4.2.1.3 AxAddDocTypeProcessor and AxAddDTDProcessor

These directives allow for stylesheet selection based on aspects of the source content's Document Type Definition (DTD).

The AxAddDocTypeProcessor directive conditionally adds processors based on the value assigned to the PUBLIC identifier contained in the source document's DTD. So, to add a style processor to match the following Simplified DocBook document, do the following:

```
<?xml version="1.0"?>

<!DOCTYPE article

        PUBLIC "-//OASIS//DTD Simplified DocBook XML V4.1.2.5//EN"

        "http://www.oasis-open.org/docbook/xml/simple/4.1.2.5/sdocbook.dtd" [  ]>

<article>

  . . .

</article>
```

You can use the following AxAddDocTypeProcessor directive:

```
AxAddDocTypeProcessor text/xsl /styles/sdocbook.xsl "-//OASIS//DTD Simplified

DocBook XML V4.1.2.5//EN"
```

Similarly, the AxAddDTDProcessor directive conditionally adds processors based on the path contained in the SYSTEM identifier in the source document's DTD:

```
<?xml version="1.0"?>

<!DOCTYPE mydoc

        SYSTEM "/path/to/my.dtd" [  ]>

<article>

  . . .

</article>
```

and the matching AxAddDTDProcessor directive:

```
AxAddDTDProcessor text/xsl /styles/sdocbook.xsl /path/to/my.dtd
```

It is important to remember that *only* the original source document is examined for matches against the conditional AxAddRootProcessor, AxAddDTDProcessor, and AxAddDocTypeProcessor directives. So, for example, if you alter the document's root element name during a transformation by one stylesheet, that new root element will *not* be evaluated against any AxAddRootProcessor directives that may exist in the current context.

4.2.1.4 AxAddURIProcessor

The AxAddURIProcessor provides a way to apply styles by matching a Perl regular expression against the current request URI. Mostly, this directive is a useful way to emulate LocationMatch blocks in contexts in which those blocks are not allowed.

```
# In httpd.conf, fine here

<LocationMatch "/my/virtual/uri">

  AxAddProcessor text/xsl /styles/application.xsl

</LocationMatch>



# But I like .htaccess files and can't use LocationMatch!!

AxAddURIProcessor text/xsl /styles/application.xsl "/my/virtual/uri"
```

4.2.1.5 AxAddDynamicProcessor

AxAddDynamicProcessor, the syntactic oddball among the processor directives, accepts the name of a Perl package as its sole argument. When this directive is found in a given context, AxKit calls the *handler( )* subroutine in the package specified. That function is expected to return a list of processor definitions that will be added to the current processing chain.

The *handler( )* subroutine is passed, in order: an instance of the current ContentProvider class, the preferred media name, the preferred style name, the source content's DTD PUBLIC identifier, its SYSTEM identifier, and root element name. If the source content does not contain a DTD, or a preferred media and style that have not been set by an upstream plug-in for the current request, those corresponding arguments will not be defined. The styles returned take the form of a list of HASH references, each containing two required key/value pairs: href, whose value should contain the path to the stylesheet being applied; and type, whose value should contain the MIME associated with the Language module that will do the transformation.

```
package My::Processors;

sub handler {

    my ($provider, $preferred_media_name, $preferred_style_name, $doctype,

$dtd, $root) = @_;



    # normally, @styles will be generated dynamically rather than hardcoded, as

    # it is here.



    my @styles = (

        { type => 'application/x-xsp', href => 'NULL' },

        { type => 'text/xsl', href => '/styles/mystyle.xsl' },

    );



    return @styles;

}
```

4.2.1.6 AxResetProcessors

The AxResetProcessors directive clears the list of processor mapping within the scope of its surrounding block. This is especially useful for handling special cases in otherwise homogeneous configurations.

```
# Set the default style for the entire site

<Directory "/www/sites/mysite">

  AxAddProcessor text/xsl /styles/global.xsl

</Directory>



# But you need to transform the content in the 'products'

# directory with a different style and you don't want to

# inherit the global style.

<Directory "/www/sites/mysite/products">

  AxResetProcessors

  AxAddProcessor text/xsl /styles/products.xsl

</Directory>
```

4.2.1.7 Style definition directive inheritance

Like most Apache configuration directives, AxKit style processor definitions are inherited recursively down directories. A style defined at the root level of the site, for example, will be applied to all documents in or below that directory in the hierarchy, and, hence, the entire site. Styles added to locations deeper in the hierarchy do not override the styles defined by their parents but rather are *added* to the processing chain. So, for instance, if you have a global style defined for the root level of the site, and one defined for a child directory named contact, all documents in the contact directory will have both styles applied during processing. What surprises many new AxKit users, however, is the order in which the styles are applied in these cases—styles defined in child directories are *prepended* to the list of processors defined by their parents, not appended, as you may initially expect. Consider the following style configuration:

```
<Directory "/www/sites/mysite">

  AxAddProcessor text/xsl /styles/global.xsl

</Directory>



<Directory "/www/sites/mysite/contact">

  AxAddProcessor application/x-xsp NULL

</Directory>
```

The *mysite* directory has a style definition that applies the *global.xsl* XSLT stylesheet to all of its contents, and that the *.contact* directory (a child of the mysite directory) adds a style definition that configures AxKit to process contents of that directory with the XSP processor. With this configuration, a document requested from the *contact* directory will have both styles applied, but the local XSP process will be applied *first* and the result of that will be processed using the *global.xsl* stylesheet. While this behavior seems a bit strange at first glance, in practice it helps simplify setting up the Style processing for many sites where a common look and feel is applied to all content and various source-specific transformations are configured for lower levels in the document hierarchy. (The results are often copied through *verbatim* by the higher level transformations.)

Taken together, these configuration directives represent a rich set of options that, when combined with Apache's standard configuration blocks, can meet the styling requirements of a great many sites. However, in Section 4.3, you will see how these components can be combined with AxKit's StyleChooser and media chooser modules to make your sites even more dynamic and responsive.

4.2.2 Stylesheet Processing Instructions

In addition to defining style processor mappings through the configuration directives, stylesheets can also be applied based on one or more xml-stylesheet processing instructions in the source document itself. Stylesheet processing instructions must appear in the prolog of the document—that is, between the XML declaration and the top-level document element:

```
<?xml version="1.0"?>]]><emphasis role="bold"><![CDATA[

<?xml-stylesheet type="text/xsl" href="/styles/docbook_simple.xsl" ?>]]></emphasis><![CDATA[

<article>

  . . .

</article>
```

Similar in behavior to the AddAddProcessor configuration directive, the xml-stylesheet processing instruction's type attribute should contain the MIME type associated with the language module that will perform the transformation. The href attribute should contain the path to the stylesheet that will be applied.

Styles added via xml-stylesheet processing instructions come in three flavors: persistent styles that are applied in all cases, preferred styles that define a default style if none is specifically selected, and alternate styles that are only applied under specific conditions.

Persistent styles are defined as those whose processing instruction has neither a title nor an alternate attribute. The following adds two persistent styles to the document that contains them:

```
<?xml-stylesheet type="text/xsl" href="/styles/nav_includes.xsl"?>

<?xml-stylesheet type="text/xsl" href="/styles/html.xsl"?>
```

All relevant styles in a given document are applied in the order in which their xml-stylesheet processing instructions appear. So, the above would tell AxKit to apply the */styles/nav_includes.xsl* stylesheet to the source XML and then apply the */styles/html.xsl* stylesheet to the result of the first transformation.

A preferred style is one whose processing instruction contains a title but not an alternate attribute. It is used to define the nonpersistent default style among a set of alternate styles:

```
<?xml-stylesheet type="text/xsl" href="/styles/html.xsl"

title="html"?>
```

An alternate style is one whose processing instruction contains *both* a title and an alternate attribute:

```
<![CDATA[

<?xml-stylesheet type="text/xsl" href="/styles/html.xsl" title="html"

alternate="yes"

]]>
```

In AxKit, alternate styles are used together with a StyleChooser module to select the appropriate styles to apply for a given circumstance. (See Section 4.3.1 later in this chapter.)

Finally, styles set via xml-stylesheet processing instruction can associate themselves with a specific media type by adding the media attribute. This allows alternate styles to be selected based on the type of device requesting the resource. (See Section 4.3.1.)

```
<?xml-stylesheet type="text/xsl" href="/styles/wap.xsl"

alternate="yes" media="handheld"?>
```

Stylesheet processing instructions are extracted *only* by the ContentProvider module that fetches the original source XML document. This means that you cannot add stylesheets to the processing chain by including xml-stylesheet instructions in the output of a stylesheet transformation. Once the document is sent to AxKit's language modules to process, the stylesheet PIs (if any) have already been extracted, and the results of the transformations are not reexamined.

4.2.2.1 AxIgnoreStylesheetPI

Any styles defined by xml-stylesheet processing instructions in the source document *override* all those defined via configuration directive. This is often desirable. For example, it provides applications that return dynamic documents through a custom ContentProvider, an easy way to set the styles for that application state while still having defaults set via configuration directive to cover common cases. In many situations, however, this is not the behavior that you want. To disregard all styles defined by xml-stylesheet processing instructions in a particular context, use the AxIgnoreStylesheetPI directive:

```
AxIgnoreStylesheetPI On
```

So, which is better, styles defined by configuration directive or by xml-stylesheet processing instruction? The answer depends largely on the site's requirements, but, in general, sites that define style processors via configuration directive tend to be more flexible and easier to maintain in the long run. For example, altering the stylesheet path for a given style is a one-line change for a site that uses configuration directive to define its styles, while the path would have to be altered in *each document* that references that style in the same site that relies on xml-stylesheet processing instructions.

PREV < Day Day Up > NEXT

4.3 Dynamically Choosing Style Transformations

The options that you've looked at so far for associating documents with transformative processes are quite capable. Often, the use of stylesheet processing instructions, or simple AxKit processor definitions, combined with constraints imposed by Apache built-in <Files>, <Directory>, <Location>, and similar block-level directives, are all you need to meet the needs of many sites. However, AxKit offers even more flexibility by providing additional mechanisms that allow you to combine these low-level style processing options into logical groups that can be selected at runtime. In this section, I introduce the concepts and syntax for creating named styles and media types and explain how these can be used in conjunction with the StyleChooser and MediaChooser modules to apply just the right content transformations under the right circumstances.

The reasons for using named style and media blocks are quite varied. Generally, they are best suited for cases when you need to select a transformation (or a chain of transformations) based on a condition *external* to the properties of the source XML content itself. Some reasons to use named styles and media blocks include:

Vendor branding

Your site offers a service, and each customer wants the content presented in a way that matches his unique look and feel.

User-selected skinning

One size never fits all. You want to offer your visitors the ability to select the style that suits them best.

Automated metadata extraction

You want to extract important metadata, like abstract summaries, author, title, copyright information, etc., by simply applying an alternative set of styles to your documents.

Role-specific data transformations

Your customers, vendors, shipping department, and company president all have different needs when they look at your product list. You want to serve all of them from the same XML data source.

Application state-specific transformations

You want to apply different transformations to your XML data to reflect the current state (or screen) of your online applications.

Any or all of the above

You want to incorporate any or all of the features listed, *plus* the ability to provide each option across a series of different client devices (phone, desktop browser, TV, etc.).

AxKit's named styles and media blocks, in conjunction with StyleChooser and MediaChooser plug-in modules, provide absolute control over when and how your XML content gets transformed. Literally, any condition that can be determined via the Perl programming language can be used to regulate which style processors will be applied to your content. The next two sections examine both named styles and named media, how they can be used together, and how AxKit's plug-ins can be used to select just the right combination.

4.3.1 Named Styles and StyleChoosers

A *named style* consists of one or more style processor definitions grouped together and given a unique name. The name given is used in conjunction with a StyleChooser or other AxKit plug-in to select the processors to apply in response to a given request. These modules set AxKit's internal preferred_style property based on a condition; then, if the name contained in that property matches the name given a named style block within the scope of the current request, all styles associated with that name are applied to the source document.

Named styles are created in one of two ways: by using the <AxStyleName> configuration directive or by adding the optional title and alternate attributes to an xml-stylesheet processing instruction in the source XML document. When using directive-based named styles, the <AxStyleName> configuration block acts as a container of one or more styling directives and uniquely associates that set of processors with the given name.

Suppose that you have an XML document that contains detailed data about the list of products offered by an online shop. You want to present a page typical for this kind of application: a list of all products, and links to more detailed information for each. Now, you could use an offline transformation to carve up the larger document into a set of smaller ones, then associate different style processors based on the files' location on the disk—one for the list view, and one for the detail views. The weakness of that approach, however, is that you must remember to rerun the process every time the product list is updated. AxKit's named styles provide another option: you can create two named styles that can be applied to the same global list of products—one presents the full list with minimal detail, and one reveals additional details for a specific product. The directives used to set up these named alternate views may look like this:

```
<Files products.xml>

  <AxStyleName
               list_view
    AxAddProcessor text/xsl /styles/list.xsl
  </AxStyleName>


  <AxStyleName detail_view
    AxAddProcessor text/xsl /styles/detail.xsl
  </AxStyleName>
</Files>
```

This creates two named styles that AxKit can apply conditionally when serving the *products.xml* document: the list_view style, which transforms the content into the complete list of products, and the detail_view style, which selects a single record from the global list and shows more detailed information about that specific item. The same can be achieved when using xml-stylesheet processing instructions by including title and alternate attributes with the appropriate values:

```
<?xml-stylesheet type="text/xsl" href="/styles/list.xsl"
                 title="list_view"  alternate="yes"?>
<?xml-stylesheet type="text/xsl" href="/styles/detail.xsl"
title="detail_view" alternate="yes"?>
```

You can now use a StyleChooser plug-in to select between the two styles. AxKit ships with a number of default StyleChooser modules that set the preferred style based on commonly used conditions such as the name of the requesting user agent, a specific query string parameter, or the value of an HTTP cookie. These modules are added to AxKit via the AxAddPlugin configuration directive. So, for example, if you want to choose between the list view and detail view of your products document, based on the query string sent by the browser, you add the following to your configuration file:

```
 AxAddPlugin Apache::AxKit::StyleChooser::QueryString
```

With this in place, you can now easily select the different views of your products document by adding a style parameter whose value matches one of your named styles to the query string of a request to *products.xml*:

```
http://myhost.tld/products.xml?style=detail_view
```

When using named styles, it is important to set up a default style to cover cases in which the StyleChoosers may not set a preferred style or in which the style name returned does not match any style configured for the current document. This can be done by adding a style with the name #default or by using the AxStyle directive to set an existing named style as the default style:

```
# Fall back to a default style that does not match any
# other named style
<AxStyleName
                 "#default"
```

```
   AxAddProcessor test/xsl /styles/list.xsl

</AxStyleName>


# Does the same, but uses the existing 'list_view' style

# as the default style.

<AxStyleName

                    list_view

  AxAddProcessor test/xsl /styles/list.xsl

 </AxStyleName>


AxStyle list_view
```

You can achieve the same using xml-stylesheet processing instructions by defining a preferred style (one that has a title attribute but not an alternate attribute with the value of yes):

```
<?xml-stylesheet type="text/xsl" href="/styles/product_list.xsl"

title="product_list"?>

<?xml-stylesheet type="text/xsl" href="/styles/product_detail.xsl"

title="product_detail" alternate="yes"?>
```

Another common use for named styles is user-defined "skinning" of a site's content. Recall your cryptozoology site from Chapter 3. Imagine that this site has become the primary resource on the Web for that topic. Both serious researchers in the area, as well as the hyper-ironic, slumming for a kitschy thrill, frequent it. Both groups share an interest in the site's content, but their expectations and preferences about *how* that content is best presented are likely to be quite different. To meet this need, you can create a new, flashier stylesheet to render the content for your hipster visitors and then add a named style to your configuration file that reflects that preference:

```
# the existing, plain style

<AxStyleName plain>

  AxAddProcessor text/xsl stylesheets/cryptozoo.xsl

</AxStyleName>


# provides the RSS 1.0 news feed

# view of the sightings

<AxStyleName rss1>

  AxAddProcessor text/xsl stylesheets/cryptidsightings_rss.xsl

</AxStyleName>


# the new, flashy style

<AxStyleName flashy>

  AxAddProcessor text/xsl stylesheets/cryptozoo_flashy.xsl
```

```
</AxStyleName>

# set the plain style as default
AxStyle plain

# lets the ?style=rss1 interface work for the news feed
AxAddPlugin Apache::AxKit::StyleChooser::QueryString
```

Here, the choice between styles is not so much a short-term, functional one (as in the list/detail view for your products document) but rather a persistent preference that should be reflected each time the reader visits the site. In this case, you are better off giving the user an HTTP cookie and using AxKit's Cookie StyleChooser.

But wait, you are already using the query string `StyleChooser` to provide the interface to the RSS feed. You don't want that link to break. No matter. Since AxKit plug-ins are executed in the order that they appear in the configuration, you only need to add the Cookie StyleChooser *before* the one that examines the query string. That way, even if the user has a cookie that sets AxKit's `preferred_style`, its value will be overwritten by the query string `StyleChooser`. Therefore, the functional transformation that provides the RSS interface will continue to work:

```
# lets the ?style=rss1 interface work for the news feed
# and allows skinning based on the user's cookie
AxAddPlugin Apache::AxKit::StyleChooser::Cookie
AxAddPlugin Apache::AxKit::StyleChooser::QueryString
```

What if you decide not to rely solely on AxKit's order of execution to handle the cases in which the styles may overlap? You can choose instead to write a little plug-in module that combines the two StyleChoosers while giving RSS query string interface top priority. (See Example 4-1.)

Example 4-1. CryptidStyleChooser.pm

```
package CryptidStyleChooser;

# Combine the Cookie and QueryString StyleChoosers, giving
# preference to a style named 'rss1' in the query string.

use strict;
use Apache::Constants qw(OK);
use Apache::Cookie;

sub handler {
    my $r = shift;

    my $preferred_style = undef;

    # Borrow the key names for the existing StyleChoosers
    my $query_key  = $r->dir_config('AxStyleChooserQueryStringKey') || 'style';
    my $cookie_key = $r->dir_config('AxStyleChooserCookieKey') ||
                     'axkit_preferred_style';

    my %query_params = $r->args( );

    if ( defined ($query_params{$query_key} ) ) {
        $style = $query_params{$query_key});

        # give preference to the rss1 query param by returning if you find it set.
        if ( $style eq 'rss1' ) {
            $r->notes('preferred_style', $style);
            return OK;
        }
    }

    # let the users' cookie override the query string otherwise
    my $cookies = Apache::Cookie->fetch; . .
    if ( defined $cookies->{$cookie_key} ) {
        $style = $cookies->{$cookie_key}->value);
    }

    # finally, set AxKit's internal 'preferred_style' if you found a style
    if ( defined( $style ) ) {
        $r->notes('preferred_style', $style;
    }
    return OK;
}

1;
```

To use your custom StyleChooser plug-in, you need to install it on your server in a location that Perl knows about and to replace the former AxAddPlugin directives with:

```
# lets the ?style=rss1 interface work for the news feed
# and allows skinning based on the user's cookie
AxAddPlugin CryptidStyleChooser
```

You can now rest assured that the correct stylesheets will be applied to the documents in your cryptid site for each case that arises. In reality, custom StyleChooser plug-ins are rarely required. (In fact, you didn't really *need* one here.) By creating a StyleChooser that covers both the presentational and functional transformations for your site, you peeked at just how easy it can be to use named styles to get AxKit to apply exactly the styles that you want, no matter how tricky the requirements first appear.

4.3.2 AxMediaType and MediaChoosers

In addition to named styles, AxKit offers a way to further define the styles that are conditionally applied to a site's resources by associating sets of styling directives with the media type the requesting client expects. This is achieved using the <AxMediaType> container directive.

```
<AxMediaType screen>
  AxAddProcessor text/xsl /styles/screen.xsl
</AxMediaType>
```

Similar to the <AxStyleName> block, the <AxMediaType> tells AxKit to examine an internal property, preferred_media, set by a MediaChooser, or other plug-in. If the value of that property matches the name given to an <AxMediaType> block, the directives within that block are used to process the current request:

```
# Set a global style for most client devices
<AxMediaType screen>
  AxAddProcessor text/xsl /styles/screen.xsl
</AxMediaType>

# But give smaller devices a different look
<AxMediaType handheld>
  AxAddProcessor text/xsl /styles/handheld.xsl
</AxMediaType>
```

Unlike named styles, AxKit sets a default value of screen for the preferred media type, if no other is found. This behavior can be altered by setting the AxMedia directive to the name of the <AxMediaType> block that you want to use as the default. For example, the following configures AxKit to use the handheld media type as the default within the current context:

```
AxMedia handheld
```

Pretend that you want to further enhance your silly cryptozoology site. You want to support visitors using web phones. In this case, you simply create another set of stylesheets that renders the content appropriately for that platform and alter your configuration to reflect the change:

```
# the previous, screen-only config, now wrapped in an <AxMediaType> block for clarity
<AxMediaType screen>
  <AxStyleName plain>
    AxAddProcessor text/xsl stylesheets/cryptozoo.xsl
  </AxStyleName>

  <AxStyleName rss1>
    AxAddProcessor text/xsl stylesheets/cryptidsightings_rss.xsl
  </AxStyleName>

  <AxStyleName flashy>
```

```
    AxAddProcessor text/xsl stylesheets/cryptozoo_flashy.xsl

  </AxStyleName>

</AxMediaType>


# the new <AxMediaType> for handheld support

<AxMediaType handheld>

  <AxStyleName plain>

    AxAddProcessor text/xsl stylesheets/cryptozoo_handheld.xsl

  </AxStyleName>


  <AxStyleName rss1>

    AxAddProcessor text/xsl stylesheets/cryptidsightings_rss.xsl

  </AxStyleName>

</AxMediaType>


AxStyle plain
```

You can give the same name to two different-named style blocks in the same configuration context without collision, so long as they appear as the children of different media type blocks. Here, you give the default style contained by the new media type the same name as the default screen style. This allows the AxStyle directive to work as expected for both media types:

```
# lets the ?style=rss1 interface work for the news feed

AxAddPlugin CryptidStyleChooser

AxAddPlugin Apache::AxKit::MediaChooser::WAPCheck
```

With this addition, your absurd little mystery animals' site becomes quite respectable, technologically speaking. You offer users a choice of styles through which to view the content, an RSS 1.0 news feed of recent cryptid sightings, and full support for surfers using web phones—all this from two XML documents, a few stylesheets, and, most importantly, knowing how to configure AxKit to apply the right style at the right time.

PREV < Day Day Up > NEXT

4.4 Style Processor Configuration Cheatsheet

So far, we have looked at the various individual elements that can be used to control how AxKit applies style transformations. So many, in fact, that seeing how these parts all fit together may be a bit tough. For example, a named style block (selected by a StyleChooser based on an environmental condition) may contain one or more AxAddDTDProcessor or similar conditional processing directives that are only applied if an additional condition is met. True AxKit mastery comes from knowing how to combine all its various configuration options to create elegant styling rules that meet the need of your specific application.

To help examine the processing order for various configuration combinations, we will create a series of very simple XSLT stylesheets whose sole purpose is to show the order in which AxKit applies a given style. The stylesheet in Example 4-2, *alpha.xsl*, simply appends the string . . . Alpha processed to the text of the top-level root element of the document being processed.

Example 4-2. alpha.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

  version="1.0">



<xsl:template match="/">

  <root><xsl:value-of select="/*"/> . . . Alpha processed</root>

</xsl:template>



</xsl:stylesheet>
```

The tiny sample XML document used for the transformations is shown in Example 4-3.

Example 4-3. minimal.xml

```
<?xml version="1.0"?>

<root>Base content</root>
```

Add more stylesheets to these—*beta.xsl*, *gamma.xsl*, and so on—that do more or less the same thing—that is, adding . . . Beta processed, etc., to the text of the root element. Wherever a simple description does not suffice, use these stylesheets to examine the precise processing order based on the returned result.

4.4.1 Rule 1: Style Processors at the Same Lexical Level Are Evaluated in Configuration-File Order

Wherever more than one style processing directive exists within a given context, each will be added (or, in the case of conditional processors, evaluated) in the order in which they appear in the configuration files. In cases in which directives are added to the same context by both the *httpd.conf* file and an *.htaccess* file, those from the *httpd.conf* are added or evaluated first:

```
<Directory /www/mysite.com/>

   AxAddProcessor text/xsl /styles/alpha.xsl

   AxAddProcessor text/xsl /styles/beta.xsl

</Directory>
```

As you may expect with this configuration, a document requested from root directory *mysite.com* will have the *alpha.xsl* stylesheet applied to the source content, then the *beta.xsl* stylesheet applied to the result of the first transformation:

```
<?xml version="1.0"?>

<root>Base content . . . Alpha processed . . . Beta processed</root>
```

Similarly, the rules governing conditional processing directives are tested in the order in which they appear in the configuration files:

```
<Directory /www/mysite.com/>

  AxAddProcessor text/xsl /styles/alpha.xsl

  AxAddRootProcessor text/xsl /styles/beta.xsl root

  AxAddProcessor text/xsl /styles/gamma.xsl

</Directory>
```

Here, the *alpha.xsl* stylesheet is added unconditionally. Then, if the source document's top-level element is named <root>, the *beta.xsl* stylesheet is added. Finally, the *gamma.xsl* stylesheet is unconditionally added to the processing chain. The result looks like this:

```
<?xml version="1.0"?>

<root>Base content . . . Alpha processed . . . Beta processed . . . Gamma processed</root>
```

If the root element of the source document were something other than <root>, or if that element were bound to a namespace that did not match the one used in the AxAddRootProcessor directive, only the *alpha.xsl* and *gamma.xsl* processors would be added to the processing chain. Compare this configuration snippet to the previous one:

```
<Directory /www/mysite.com/>

  AxAddProcessor text/xsl /styles/alpha.xsl

  AxAddRootProcessor text/xsl /styles/beta.xsl {http://myhost.tld/namspaces/root}root

  AxAddProcessor text/xsl /styles/gamma.xsl

</Directory>
```

Here, the *beta.xsl* would *not* be added to the processing chain because, although the name of the top-level element matches the one in your AxAddRootProcessor directive, the directive specifies that the <root> element must be bound to the http://myhost.tld/namespaces/root namespace URI. This is not the case in our sample XML document. Therefore, only the *alpha.xsl* and *gamma.xsl* stylesheets would be applied.

Mixing various conditional processing directives at the same level can be quite powerful. Many simple sites really only use a handful of source XML grammars, and if only one result format is expected (i.e., HTML), setting up a block of carefully chosen conditional processors for the host's top-level directory can often cover most, if not all, the processing rules for the entire site. Here is a more practical example:

```
# The root directory for our AxKit-enabled site

<Directory /www/mysite.com/>

  AxAddDocTypeProcessor text/xsl /styles/docbook/html/docbookx.xsl "-//OASIS//DTD DocBook

 XML V4.2//EN"

  AxAddRootProcessor text/xsl  /styles/rss2html.xsl {http://www.w3.org/1999/02/22-rdf-
syntax-ns#}RDF

   AxAddRootProcessor application/x-xsp NULL {http://apache.org/xsp/core/v1}page

   AxAddRootProcessor text/xsl /styles/appgrammar2html.xsl

{http://apache.org/xsp/core/v1} page

</Directory>
```

Here, you have a small group of conditional processing directives—one that will be applied to documents containing a Document Type Definition with the SYSTEM ID indicating the latest version of DocBook, one that will be applied if the content's top-level element is named RDF and that element is bound to the http://www.w3.org/1999/02/22-rdf-syntax-ns# namespace, and two that will match any document that contains a top-level page element bound to the http://apache.org/xsp/core/v1 namespace URI.

Just by setting up this simple block of conditional processing directives at the top level of the site, you can now publish static RSS 1.0 news feeds and DocBook documents (in their myriad forms from books to FAQs), as well as generate dynamic XML content via eXtensible Server Pages from any directory at or below this level in the site. AxKit will simply work as expected. Admittedly, this setup presumes that you will publish documents only as HTML, but that's still a lot of power for minimal effort.

By setting two conditional AxAddRootProcessor directives to match the same root element, you create a two-step processing chain for XSP documents in which AxKit's XSP engine processes the source, and the */styles/appgrammar2html.xsl* XSLT stylesheet processes the resulting markup.

4.4.2 Rule 2: Conditional Processing Directives Are Evaluated Only Against the Original XML Source

The rules for style processors that are applied conditionally based on a feature contained in the XML source (AxAddDTDProcessor, AxAddRootProcessor, etc.) are only evaluated against the original source content, not subsequent transformations in the processing chain. That is, the rules are evaluated *once*, using the source document returned from the ContentProvider. The rules are *not* reevaluated. Therefore, if you have a stylesheet that changes the root element name during transformation, and that new root element coincides with the name passed to an AxAddRootProcessor directive that is in scope for that request, the AxAddRootProcessor rule does *not* match, since the transformed result is not examined. Similarly, only those xml-stylesheet processing instructions contained in the original source document will be considered. You cannot add to or otherwise alter the processing chain by adding a stylesheet PI to the result of a transformation.

4.4.3 Rule 3: Style Processors Are Prepended to the Processing Chain as You Descend into the Directory Hierarchy

Although we touched on this earlier, it bears repeating: style definitions are *prepended* to the processing chain as one descends into the site's directory tree. That is, processing definitions at deeper levels in the site are added to the front of the processing chain, not to the end. This is most easily explained with a simple example using the stylesheets that illustrate AxKit processing order:

```
# the site's DocumentRoot directory

<Directory /www/mysite.com/>

   AxAddProcessor text/xsl /styles/alpha.xsl

</Directory>


# a child directory of the DocumentRoot

<Directory /www/mysite,com/levelone/>

   AxAddProcessor text/xsl  /styles/beta.xsl
```

```
</Directory>


# a child directory of the 'levelone' directory

<Directory /www/mysite.com/levelone/leveltwo/>

  AxAddProcessor text/xsl  /styles/gamma.xsl

</Directory>
```

Given this configuration, a request for http::/myhost.tld/levelone/leveltwo/minimal.xml yields the following result:

```
<?xml version="1.0"?>

<root>Base content . . . Gamma processed . . . Beta processed . . . Alpha processed</root>
```

The styles are added to the processing chain from the bottom up, so the *alpha.xsl* stylesheet is applied last since it is at the top level of the site's hierarchy. Again, the reason is that, in general, most sites deploy more generic styles (those that will be applied to all or most of the documents in the site) at higher levels in the directory tree. This behavior allows that to work as expected, while handling special cases in lower-level directories by prepending special processors to the chain.

To see the utility of this behavior, imagine that you are publishing a site that is divided into several sections. In addition to the content being rendered, these sites usually have a certain amount of markup that appears on every page (a common graphical header, a legal/copyright footer, a common navigation bar, etc.). They also have a certain amount of markup that is common, but unique, to each individual section (section headers, contextual navigation, etc.). By building the processing chain from the bottom up, AxKit allows you to put the site-wide boilerplate markup in a top-level stylesheet, while handling the section-specific markup in each section's unique directory.

```
# the site's DocumentRoot

<Directory /www/mysite.com/>

  # the global stylesheet that contains the site-wide headers/footers, etc.

  AxAddProcessor text/xsl  /styles/global.xsl

</Directory>


<Directory /www/mysite.com/products/>

  # adds the markup common to all pages in the 'products' section (contextual navigation,

 section headers, etc)..

  AxAddProcessor text/xsl  /styles/products.xsl

</Directory>


<Directory /www/mysite.com/locations/>

  # adds the markup common to all pages in the 'locations' section . . .

  AxAddProcessor text/xsl  /styles/locations.xsl

</Directory>
```

Here, a request for a document from the products directory would be transformed by the */styles/locations.xsl* stylesheet to add the section-specific content, then passed to the */styles/global.xsl* stylesheet to add the site-wide boilerplate. This strategy presumes that higher-level stylesheets add only what is needed at that level and pass the rest of the markup through as-is. For example, with the above configuration, the stylesheet applied to the source content may return a result that has a div root element containing the page's main text; that result may then be wrapped in another div by the */styles/location.xsl* stylesheet to add the section-specific markup (while copying the result of the original transformation through, untouched). Finally, *that* result is wrapped by the */styles/global.xsl* stylesheet to create the completed document. (Think: the component-based approach to constructing pages but built from the bottom up.)

Also remember that Rule 1 still applies: style definitions in the same lexical scope are evaluated in the order found in the configuration files. This means that if you have more than one processing directive that matches at a given level in the directory hierarchy, the processors for that level are still added in the order they appear, and that *group* of processors will be prepended to the list of processors from a higher level. An example may help clarify:

```
 # the site's DocumentRoot directory

<Directory /www/mysite.com/>

  AxAddProcessor text/xsl /styles/alpha.xsl

  AxAddProcessor text/xsl  /styles/beta.xsl

</Directory>


# a child directory of the DocumentRoot

<Directory /www/mysite.com/levelone/>

  AxAddProcessor text/xsl  /styles/gamma.xsl

  AxAddProcessor text/xsl  /styles/delta.xsl

</Directory>


# a child directory of the 'levelone' directory

<Directory /www/mysite.com/levelone/leveltwo/>

  AxAddProcessor text/xsl  /styles/epsilon.xsl

</Directory>
```

Here, using your processing order stylesheets, a request for http::/myhost.tld/levelone/leveltwo/minimal.xml returns:

```
<?xml version="1.0"?>

<root>Base content . . . Epsilon processed . . . Gamma processed . . .

   Delta processed . . . Alpha processed . . . Beta processed</root>
```

At each level, the processors are added in the order found in the configuration file for that scope, but each of those groups of processors is added to the chain from the bottom up. Now, apply this idea to the previous sample. You can see how the page-level content can be generated for the location and products sections of the site:

```
# the site's DocumentRoot

<Directory /www/mysite.com/>

  # the global stylesheet that contains the site-wide headers/footers, etc.

  AxAddProcessor text/xsl  /styles/global.xsl

</Directory>


<Directory /www/mysite,com/products/>

  # generate XML from database select for 'products'

  AxAddProcessor application/x-xsp NULL


  # Transform 'product' application grammar to HTML
  AxAddProcessor text/xsl /styles/product2html.xsl


  # adds the markup common to all pages in the 'products'

  # section (contextual navigation, section headers, etc)..

  AxAddProcessor text/xsl  /styles/products.xsl

</Directory>


<Directory /www/mysite,com/locations/>

  # generate XML from database select for 'locations'

  AxAddProcessor application/x-xsp NULL


  # Transform 'product' application grammar to HTML

  AxAddProcessor text/xsl /styles/locations2html.xsl


  # adds the markup common to all pages in the 'locations' section . . .

  AxAddProcessor text/xsl  /styles/locations.xsl

</Directory>
```

Now, each subdirectory has two processing definitions: one that processes the XSP source content to generate markup for the location and products data from a database query, respectively, and one that transforms each domain-specific grammar into HTML. The results of those transformations are then passed to the *global.xsl* stylesheet, which passes that markup through, adding things like the common header, site-wide navigation, copyright information, etc. Again, in each lexical scope (the *location/* and *products/* subdirectories), the processor definitions are added to the processing chain in the same order they are found in the configuration file. However, since they are at a lower level in the directory structure than the global stylesheet, each ordered group is processed *before* the global one. If you want to keep this same layout but need to allow more than a single type of document in each of the subdirectories, you could use groups of conditional processing directives instead of the unconditional AxAddProcessor directives you have here to match the properties of each supported document type. As long as the result passes up to the global stylesheet in the grammar expected, you can mix and match as needed. Everything still works.

4.4.4 Rule 4: Processors in the Named BlocksAre Always Evaluated Last

Essentially, named style and media blocks create their own lexical scope. When a named style or media is selected by a StyleChooser, MediaChooser, or other plug-in and there are other processing directives that match for the same request, the processors from the named blocks are always added last. Let's break out the processing order stylesheets again and examine the details:

```
# the site's DocumentRoot

<Directory /www/mysite.com/>


 # First, add the QueryString StyleChooser so you can easily select named styles

 AxAddPlugin Apache::AxKit::StyleChooser::QueryString


 # Add a simple named style

 <AxStyleName styleone>

    AxAddProcessor text/xsl /styles/beta.xsl

    AxAddProcessor text/xsl /styles/gamma.xsl
```

```
  </AxStyleName>


  # Add a 'global' style

  AxAddProcessor text/xsl /styles/alpha.xsl

</Directory>
```

A request for the *minimal.xml* file in the directory that does not specify a preferred style yields the expected result—only the *alpha.xsl* stylesheet is applied:

```
<?xml version="1.0"?>

<root>Base content . . . Alpha processed</root>
```

But what happens if you select the named style for a request to the same document? Given Rule 1, you may expect the processors in the named style block to be applied first, since the named block appears before the global style. This is not the case. A request for *minimal.xml* with style=styleone in the query string gives the following:

```
<?xml version="1.0"?>

<root>Base content . . . Alpha processed . . . Beta processed . . . Gamma processed</root>
```

The styles contained in the <AxStyleName> block are added according to Rule 1. (They are at the same lexical level and are, therefore, added in the order they appear in the configuration file.) Because they are contained in that block, they are added last to the processing chain, *after* the unnamed, global style (*alpha.xsl*).

The "named processors are always last" rule also applies in cases in which a named block appears at a lower level in the document hierarchy than any matching unnamed directives. This creates a partial exception to Rule 3.

```
# the site's DocumentRoot

<Directory /www/mysite.com/>


  # Add the QueryString StyleChooser so you can easily select named styles

  AxAddPlugin Apache::AxKit::StyleChooser::QueryString


  # Add a 'global' style

  AxAddProcessor text/xsl /styles/alpha.xsl

</Directory>


# A first-level subdirectory

<Directory /www/mysite.com/levelone/>


  # Add a simple named style

  <AxStyleName styleone>

      AxAddProcessor text/xsl /styles/beta.xsl

      AxAddProcessor text/xsl /styles/gamma.xsl

  </AxStyleName>

</Directory>
```

Here, a request for *http://mysite.com/levelone/minimal.xml?style=styleone* gives the same result as before, despite the fact that the matching named style block is defined at a lower level in the directory tree than the global stylesheet configured for the site's DocumentRoot:

```
<?xml version="1.0"?>

<root>Base content . . . Alpha processed . . . Beta processed . . . Gamma processed</root>
```

One final, admittedly pernicious, example reveals precisely what's going on:

```
# the site's DocumentRoot

<Directory /www/mysite.com/>


  # Add the QueryString StyleChooser so you can easily select named styles

  AxAddPlugin Apache::AxKit::StyleChooser::QueryString


  # Add a 'global' style

  AxAddProcessor text/xsl /styles/alpha.xsl


# Add a simple named style

  <AxStyleName styleone>

      AxAddProcessor text/xsl /styles/beta.xsl

      AxAddProcessor text/xsl /styles/gamma.xsl

  </AxStyleName>

</Directory>


# A first-level subdirectory

<Directory /www/mysite.com/levelone/>


  # another  'global' style

  AxAddProcessor text/xsl /styles/delta.xsl


  # The style name here is the same as the one in the parent directory!

  <AxStyleName styleone>

      AxAddProcessor text/xsl /styles/eplison.xsl

      AxAddProcessor text/xsl /styles/zeta.xsl

  </AxStyleName>

</Directory>
```

Here, a request to *http://mysite.com/levelone/minimal.xml?style=styleone* results in the following:

```
<?xml version="1.0"?>

<root>Base content . . . Delta processed . . . Alpha processed . . . Epsilon processed . . .

    Zeta processed . . . Beta processed . . . Gamma processed</root>
```

Surprised? Let's take things step by step:

1. The global *alpha.xsl* stylesheet is added to the processing chain.
2. The global processor *delta.xsl* is configured for a directory at a lower level and is, therefore, prepended to the list of processors, according to Rule 3.
3. The style named styleone is selected, so the two processors contained in the named block at the root level of the site (*beta.xsl* and *gamma.xsl*) are added in configuration order, according to Rule 1.
4. The two processors (*epsilon.xsl* and *zeta.xsl*) in the named block in the levelone directory are also added in configuration order, according to Rule 1. Since they appear at a lower level in the directory tree, they are added, as a group, to the list of named styles *before* the two in the parent directory, according to Rule 3.
5. *All* processors associated with a named style or media block are added to the processing chain *after* all those that are not, according to Rule 4.

The simplest way to conceptualize the processing order for rare cases such as this is that all processing directives not contained in an <AxStyleName> or <AxMediaType> blocks are combined into one list, according to Rules 1-3; a second list of those contained in a named style or media block is created, also using Rules 1-3; then the two lists are combined, with the "named styles list" always appearing last.

If all this seems a bit much, rest assured. Cases such as this last example are extremely rare in the real world. The example here is particularly nasty only because it illustrates every detail of how AxKit creates its processing chain.

Now that you have had a close look at AxKit's many options for applying different transformative styles to XML documents. It is time to look at *how* those transformations happen by examining two of the more popular languages that it supports for transforming content: XSLT and XPathScript.

Chapter 5. Transforming XML Content with XSLT

XSLT (The eXtensible Stylesheet Language: Transformations) is an XML application language used for transforming XML documents into other documents. It is implemented by an application called an XSLT processor that takes an XML document and an XSLT stylesheet as input and produces a new document by applying the instructions contained in the stylesheet to the original source XML document. The result of an XSLT transformation can be any text-based format, but the output is typically either another XML document, or a document in a widely deployed markup language such as HTML that can be readily consumed by a given client application.

AxKit is not an XSLT processor, nor does it ship with one. If you want to use XSLT to transform your XML content using AxKit, you need to install an XSLT processor and any necessary Perl interface modules separately. For the list of XSLT processors that AxKit currently supports, see XML Processing Options in Chapter 2. For details about how to use the AxAddStyleMap directive to associate XSLT stylesheets with the processor you install and the various directives that govern which stylesheets are applied to your XML documents, see Chapter 4.

Exhaustive coverage of XSLT is well beyond the scope of this chapter. The goal here is to introduce enough of the basic concepts of writing XSLT stylesheets to allow you to start being productive with AxKit as quickly as possible. For a more detailed look at XSLT, see *XSLT*, by Doug Tidwell, or *Learning XSLT*, by Mike Fitzgerald (both from O'Reilly). All of the samples here use only XSLT 1.0. At the time of this writing, XSLT 2.0 is still very new and not widely implemented, and existing implementations are highly experimental. Rest assured though, XSLT 1.0 is still a viable tool. The topics covered here generally apply to both versions, and support for use of Version 2.0 processors from within AxKit will be added just as soon as stable implementations begin to appear.

5.1 XSLT Basics

An XSLT stylesheet is made up of a single top-level xsl:stylesheet element that contains one or more xsl:template elements. These templates can contain literal elements that become part of the generated result, functional elements from the XSLT grammar that control such things as which parts of the source document to process, and often a combination of the two. The contents of the source XML document are accessed and evaluated from within the stylesheet's templates and other function elements using the XPath language. The following shows a few XSLT elements (the associated XPath expression is highlighted):

```
<xsl:value-of select="price"/>

<xsl:apply-templates select=" /article/section"/>

<xsl:copy-of select="order/items"/>
```

5.1.1 The XPath Language

XPath is a language used to select and evaluate various properties of the elements, attributes, and other types of nodes found in an XML document. In the context of XSLT, XPath is used to provide access to the various nodes contained by the source XML document being transformed. The XPath expressions used to access those nodes is based on the relationships between the nodes themselves. Nodes are selected using either the shortcut syntax that is somewhat reminiscent of the file and directory paths used to describe the structure of a Unix filesystem, or one that describes the abstract relationship axes between nodes (parent, child, sibling, ancestor, descendant, etc.). In addition, XPath provides a number of useful built-in functions that allow you to evaluate certain properties of the nodes selected from a given document tree. The most common components of an XPath expression are location paths and relationship axes, function calls, and predicate expressions.

5.1.1.1 Location paths and relationship axes

Borrowing from the Document Object Model, XPath visualizes the contents of an XML document as an abstract tree of nodes. At the top level of that tree is the root node represented by the string /. The root node is *not* the same as the top-level element (often called the document element) in the XML document, but is rather an abstract node above that level, which contains the document element and any special nodes, such as processing instructions:

```
<?xml version="1.0"?>

<?xml-stylesheet href="mystyle.xsl" type="text/xsl"?>

<page>

  <para>I &#2665; the XML Infoset</para>

</page>
```

In this document, the xml-stylesheet processing instruction is a meaningful part of the document as a whole, but it is not contained by the page document element. Were it not for the abstract root node floating above the document element, you would have no way to access the processing instruction from within your stylesheets.

The practical result of having a root node above the top-level document element is that all XPath expressions that attempt to select nodes using an absolute path from root node to a node contained in the document must include both an / for the root node and the name of the document element.

Here are a few examples of absolute location paths:

```
/

/html/body

/book/chapter/sect1/title

/recordset/row/order-quantity
```

Relative location paths are resolved within the context of the current node (element, attribute, etc.) being processed. The following are functionally identical:

```
chapter/sect1

child::chapter/sect1
```

Attribute nodes are accessed by using the attribute:: axis (or the shortcut, @), followed by the name of the attribute:

```
attribute::class

@class

sect1/attribute::id

sect1/@id

/html/body/@bgcolor
```

Relationship axes provide a way to access nodes in the document based on relative relationships to the context node that often cannot be captured by a simple location path. For example, you can look back up the node tree from the current node using the ancestor or parent axis:

```
ancestor::chapter/title

parent::chapter/title
```

or across the tree at the same level using the preceding-sibling or following-sibling axes:

```
preceding-sibling::product/@id

following-sibling::div/@class
```

XPath also provides several useful shortcuts to help make things easier. The . (dot) is an alias for the self axis, and .. is an alias for the parent axis. The // path abbreviation is an alias for the descendant-or-self axis. While using // can be expensive to process, it is hard to fault the simplicity it offers. For example, collecting all the hyperlinks in an XHTML document into a single nodeset, regardless of the current context or where the links may appear in the document, is as simple as:

```
//a
```

Similarly, you can select all the para descendants of the context node (the node currently being processed) using:

```
.//para
```

If all these relationship axes are confusing, recall that XPath visualizes the document as a hierarchical tree of nodes in much the same way that the filesystem is presented in most Unix-like operating systems. (Parents and ancestors are "up" towards the "root," siblings are at the same level on a given branch, and children and descendants are contents of the current node.)

5.1.1.2 Functions

XPath provides a nice list of built-in functions to help with node selection, evaluation, and processing. This list includes functions for accessing the properties of nodesets (e.g., position( ) and count( )), functions for accessing the abstract components of a given node (e.g., namespace-uri( ) and name( )), string processing functions (e.g., substring-before( ), concat( ), and translate( )), number processing (e.g., round( ), sum( ), and ceiling( )) and Boolean functions (e.g., true( ), false( ), and not( )). A detailed reference covering each function is not appropriate here, but a few useful examples are.

Get the number of para elements in the document:

```
count(//para)
```

Create a fully qualified URL based on the relative_url attribute of the context node:

```
concat('http://mysite.com/', @relative_url )
```

Replace dashes with underscores in the text of the context node:

```
translate(., '-', '_' )
```

Get the scheme name from a fully qualified URL:

```
substring-before(@url, '://' )
```

Quickly, get the total number of items ordered:

```
sum(/order/products/item/quantity)
```

In addition to the core functions provided by XPath, XSLT adds several additional functions to make the task of transforming documents easier, or more robust. Two are especially useful; the document( ) (which provides a way to include all or part of separate XML documents into the current result), and the key( ) function (which offers an easy way to select specific nodes from larger, regularly-shaped sets by using part of the individual nodes as a lookup). You can find a complete listing of XSLT's functions on the Web at http://www.w3.org/TR/xslt#add-func.

Many functions provided by XPath and XSLT can be useful for transforming the source document's content; others are only useful when combined with XPath predicate expressions.

5.1.1.3 Predicates

Predicate expressions provide the ability to examine the properties of a nodeset in a way similar to the WHERE clause in SQL. When a predicate expression is used, only those nodes that meet the criteria established by the predicate are selected. Predicates are set off from the rest of the expression using square brackets and can appear after any node identifier in the larger XPath expression. When the predicate expression evaluates to an integer, the node at that position is returned. The following all select the second div child of the context node:

```
div[2]

div[1 + 1]

div[position( ) = 2]

div[position( ) > 1 and position < 3]
```

More than one predicate can be used in a given expression. The following returns a nodeset containing the title of the first section of the fifth chapter of a book in DocBook XML format.

```
/book/chapter[5]/sect1[1]/title
```

Predicate expressions may also contain function calls. The following selects all descendants of the current node that contain significant text data but no child elements:

```
.//*[string-length(normalize-space(text( ))) > 0 and count(child::*) = 0]
```

5.1.2 Stylesheet Templates

The basic building block of an XSLT stylesheet is the xsl:template element. The xsl:template element is used to create the output of the transformation. A template is invoked either by matching nodes in the source document against a pattern contained by the template element's match attribute, or by giving the template a name via the name attribute and calling it explicitly.

The xsl:template element's match attribute takes a pattern expression that determines to which nodes in the source tree the template will be applied. These match rules can be evaluated and invoked from within other templates using the xsl:apply-templates element. The pattern expressions take the form of an XPath location expression that may also include predicate expressions. For example, the following tells the XSLT processor to apply that template to all div elements that have body parents:

```
<xsl:template match="body/div">

 . . .

</xsl:template>
```

You can also add predicates to your patterns for more fine-tuned matching:

```
<xsl:template match="body/div[@class='special']">

  . . .

</xsl:template>
```

By adding the @class='special' predicate, this template would only be applied to the subset of those elements matched by the previous example that have a class attribute with the value special.

Templates that contain match rules are invoked using the xsl:apply-templates element. If the xsl:apply-templates element's optionalselect attribute is present, the nodes returned by its pattern expression are evaluated one at a time against each pattern expression in the stylesheet's templates. When a match is found, the nodes are processed by the matching template. (If no match is found, a built-in template is used, and the children of the selected nodes are processed, and so on, recursively.) If the select attribute is not present, all child nodes of the current node are evaluated. This allows straightforward recursive processing of tree-shaped data.

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:template match="/">

   <!-- root template, always matches -->

   <html>

      <xsl:apply-templates/>

   </html>

  </xsl:template>


  <xsl:template match="article">

    <!-- matches element nodes named 'article' -->

    <body>

      <xsl:apply-templates/>

    </body>

  </xsl:template>


  <xsl:template match="para">

    <!-- matches element nodes named 'para' -->

    <p>

      <xsl:apply-templates/>
    </p>
  </xsl:template>

  <xsl:template match="emphasis">
    <!-- matches element nodes named 'emphasis' -->
    <em>
      <xsl:apply-templates/>
    </em>
  </xsl:template>

</xsl:stylesheet>
```

Applying this stylesheet to the following XML document:

```
<?xml version="1.0"?>
<article>
 <para>
 I was <emphasis>not</emphasis> pleased with the result.
 </para>
</article>
```

gives the following result:

```
<?xml version="1.0"?>
<html>
 <body>
 <p>
  I was <em>not</em> pleased with the result.
 </p>
 </body>
</html>
```

The name attribute provides a way to explicitly invoke a given template from within other templates using the xsl:call-template element. For example, the following inserts the standard disclaimer contained in the template named document_footer at the bottom of an HTML page:

```
<xsl:template match="/">
   . . .  more processing here  . . .
    <xsl:call-template name="document_footer"/>
    </body>

  </html>

</xsl:template>



<xsl:template name="document_footer">

  <div class="footer">

    <p>copyright © 2001 Initech LLC. All rights reserved.</p>

  </div>

</xsl:template>
```

The xsl:template element's mode attribute provides a way to have template rules that have the same match expression (match the same nodeset) but process the matched nodes in a very different way. For example, the following snippet shows two templates whose expressions match the element nodes named section. One displays the main view of the document (the default), and the other builds a table of contents:

```
<xsl:template match="article">

  <!-- create the table of contents first -->

  <div class="toc">

    <xsl:apply-templates select="section" mode="toc"/>

  </div>

  <!-- then process the document as usual -->

  <xsl:apply-templates select="section"/>

</xsl:template>



<xsl:template match="section">

  <div class="section">

    <h2>

      <a name="{generate-id(title)}">

        <xsl:value-of select="title"/>

      </a>

    </h2>

    <xsl:apply-templates/>

  </div>

</xsl:template>



<xsl:template match="section" mode="toc">

  <a href="#{generate-id(title)}">

    <xsl:value-of select="title"/>

  </a>

  <br />

  <xsl:apply-templates select="section" mode="toc"/>

</xsl:template>
```

5.1.3 Loop Constructs

Adding to its flexibility, XSLT borrows the concepts of iterative loops and conditional processing from traditional programming languages. The xsl:for-each element is used for iterating over the nodes in the nodeset returned by the expression contained in its select attribute. In spirit, this corresponds to Perl's foreach (@some_list) loop.

```
<xsl:template match="para">

  <xsl:for-each select="xlink">

    . . .  do something with each xlink child of the current para element

  </xsl:foreach>

</xsl:template>
```

In the earlier example showing how template modes are used, you created two templates for the section elements: one for processing those nodes in the context of the main body of the document, and one for building the table of contents. You could just as easily have used an xsl:for-each element to create the table of contents:

```
<xsl:template match="article">

  <!-- create the table of contents first -->

  <div class="toc">

    <xsl:for-each select=".//section">

      <a href="#{generate-id(title)}">

        <xsl:value-of select="title"/>

      </a>

      <br />

    </xsl:for-each>

  </div>

  <!-- then process the document as usual -->

  <xsl:apply-templates select="section"/>

</xsl:template>
```

So, which type of processing is better: iteration or recursion? There is no hard-and-fast rule. The answer depends largely on the shape of the node trees being processed. Generally, an iterative approach using xsl:for-each is appropriate for nodesets that contain regularly shaped data structures (such as line items in a product list), while recursion tends to work better for irregular trees containing mixed content (elements that can have both text data and child elements, such as articles or books). XSLT's processing model is founded on the notion of recursion (process the current node, apply templates to all or some of the current node's children, and so on). The point is that one size does not fit all. Having a working understanding of both styles of processing is key to the efficient and professional use of XSLT.

5.1.4 Conditional Blocks

Conditional "if/then" processing is available in XSLT using xsl:if, xsl:choose, and their associated child elements.

The xsl:if element offers an all-or-nothing approach to conditional processing. If the expression passed to the processor through thetest attribute evaluates to *true*, the block is processed; otherwise, it is skipped altogether.

Consider the following template. It prints a list of an employee's coworkers, adding the appropriate commas in between the coworker's names (plus an and just before the final name) by testing the position( ) of each coworker child:

```
<xsl:template match="employee">
  <p>
    Our employee <xsl:value-of select="first-name"/> works with:
    <xsl:for-each select="coworker">
      <xsl:value-of select="."/>
      <xsl:if test="position( ) != last( )">, </xsl:if>
      <xsl:if test="position( ) = last( )-1"> and </xsl:if>
    </xsl:for-each>
    on a daily basis.
  </p>
</xsl:template>
```

In cases in which you need to emulate the if-then-else block or switch statement found in most programming languages, use the xsl:choose element. An xsl:choose must contain one or more xsl:when elements and may contain an optional xsl:otherwise element. The test attribute of each xsl:when is evaluated in turn and contents processed for the first expression that evaluates to true. If none of the test conditions return a true value and an xsl:otherwise element is included, the contents of that element are processed:

```
<xsl:template match="article">
  <xsl:choose>
    <xsl:when test="$page-view='toc'">
      <xsl:apply-templates select="section" mode="toc"/>
    </xsl:when>
    <xsl:when test="$page-view='summary'">
      <xsl:apply-templates select="abstract" mode="summary"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-templates select="section"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

5.1.5 Parameters and Variables

XSLT offers a way to capture and reuse arbitrary chunks of data during processing via the xsl:param and xsl:variable elements. In both cases, a unique name is given to the parameter or variable using a name attribute, and the contents can be accessed elsewhere in the stylesheet by prepending a $ (dollar sign) to that name. Therefore, the value of a variable declaration whose name attribute is myVar will be accessible later as $myVar.

The xsl:param element serves two purposes: it provides a mechanism for passing simple key/value data to the stylesheet from the outside, and it offers a way to pass information between templates within the stylesheet. One benefit of using XSLT in an environment such as AxKit is that all HTTP parameters are available from within your stylesheets via xsl:param elements:

```
<?xml version="1.0"?>

<xsl:stylesheet

     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

     version="1.0">

<xsl:param name="user"/>


<xsl:template match="/">

   . . .

  <p>Greetings, <xsl:value-of select="$user"/>.

  Welcome to our site.</p>

   . . .

</xsl:template>

 . . .
```

The result of a request to a document that is transformed by this stylesheet, such as http://myhost.tld/mypage.xml?user=ahclem (or a POST request that contains a defined "user" parameter), contains the following:

```
  <p>Greetings, ahclem. Welcome to our site.</p>
```

Default values can be set by adding text content to the xsl:param element:

```
<xsl:param name="user">Mystery Guest</xsl:param>
```

or by passing a valid XPath expression to its select attribute:

```
<xsl:param name="user" select="'Mystery Guest'"/>

<xsl:param name="my-value" select="/path/to/default"/>
```

Parameters can also be used to pass data to other templates during the transformation process. In this second form, a template rule is invoked using the xsl:call-template element, whose required name attribute must correspond to the name attribute of an xsl:template, or by using an xsl:apply-templates that is expected to match one of the template's match attributes. In both cases, data can be passed to the template via one or more xsl:with-param elements, which are then available inside its invoked template using xsl:param elements. The result returned by the called template is inserted into the transformation's output at the point that the template is called.

Calling named templates and passing in parameters is the closest thing that XSLT has to the user-defined functions or subroutines provided in traditional programming languages. It can be employed to create reusable pseudofunctions:

```
<xsl:template match="guestlist">

  <xsl:for-each select="visitor">

    <xsl:call-template name="lastname-first">

      <xsl:with-param name="fullname" select="name"/>

    </xsl:call-template>

  </xsl:for-each>

   . . .

</xsl:template>

 . . .
```

```
<xsl:template name="lastname-first">
  <xsl:param name="fullname"/>
  <xsl:value-of select="$fullname/lastname"/>
  <xsl:text>, </xsl:text>
  <xsl:value-of select="$fullname/firstname"/>
  <xsl:if test="$fullname/middle-initial">
    <xsl:text> </xsl:text>
    <xsl:value-of select="$fullname/middle-initial"/>
    <xsl:text>.</xsl:text>
  </xsl:if>
</xsl:template>
```

The xsl:variable element is similar to xsl:param in that it can be assigned an arbitrary value such as a nodeset or string. Unlike parameters, though, variables only provide a way to store chunks of data; they are not used to pass information in from the environment or between templates.

They can be useful for such things as creating shortcuts to complex nodesets:

```
<xsl:varable name="super-stars"
            select="/roster/players[points-per-game > 25]"
/>
```

or setting default values for data that may be missing from a given part of the document:

```
<xsl:varable name="username">
 <xsl:choose>
  <xsl:when test="/application/session/username">
    <xsl:value-of select="/application/session/username"/>
  </xsl:when>
  <xsl:otherwise>Unknown User</xsl:otherwise>
 </xsl:choose>
</xsl:variable>
```

Once a variable or parameter has been assigned a value, it becomes read-only. This behavior trips most web developers who are used to doing such things as:

```
my $grand_total = 0;
foreach my $row (@order_data) {
        $grand_total += $row->{quantity} * $row->{price};
}
print "Order Total: $grand_total";
```

There is a way around this limitation, but it requires creating a template that recursively consumes a nodeset while passing the sum of the previous value and current value back to itself through a parameter as each node is processed:

```
<xsl:template match="/">
<root>
   . . .
  Order total:
  <xsl:call-template name="price-total">
    <xsl:with-param name="items" select="order/item"/>
  </xsl:call-template>
</root>
</xsl:template>

<xsl:template name="price-total">
  <xsl:param name="items"/>
  <xsl:param name="total">0</xsl:param>
  <xsl:choose>
    <xsl:when test="$items">
      <xsl:call-template name="price-total">
        <xsl:with-param name="items"
                              select="$items[position( ) != 1]"/>
        <xsl:with-param name="total"
                        select="$total + $items[position( )=1]/quantity/text( ) *
$items[position( )=1]/price/text( )"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="format-number($total, '#,##0.00')"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

If you are more used to thinking in Perl, the following snippet illustrates the same principle:

```
my $order_total = &price_total('0', @order_items);

sub price_total {
    my ($total, @items) = @_;
        if (@items) {
                my $data = shift (@items);
                &price_total($total + $data->{price} * $data->{quantity}, @items);
        }
        else {
```

```
        return $total;

    }

}
```

This short introduction to XSLT's syntax and features really only touches the surface of what it can achieve. If what you read here intrigues you, I strongly recommend picking up one of the many fine books that cover the language in much greater depth.

5.2 A Brief XSLT Cookbook

As I mentioned in the introduction, the goal of this chapter is to provide just enough information to allow you to be productive as quickly as possible with XSLT and AxKit. To that end, we will complete our whirlwind tour by looking at how XSLT can be used for several tasks that web developers are commonly asked to perform.

5.2.1 Delivering Browser-Friendly HTML

5.2.1.1 Problem

Your stylesheets work fine, but older HTML browsers are choking on tags such as <br/> and <img/>.

5.2.1.2 Solution

Use the xsl:output element and set its method attribute to "html":

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output method="html" />

  . . .
```

5.2.1.3 Discussion

The xsl:output element offers an easy way to control the formatting of the result of a given transformation. Other valid values for the method attribute include "text", and "xml" (the default). This element offers several other useful options, including the ability to set the encoding of the result document (via the encoding attribute) and the ability to add a document type declaration to the output (using the doctype-system and doctype-public attributes).

5.2.2 Alternating Colors in HTML Table Rows

5.2.2.1 Problem

You are transforming a document that consists of a long list of line items into HTML. You want to make the background of every other row a different color so that the page is more readable.

5.2.2.2 Solution

Use an xsl:if element that tests the value of the current node's position and conditionally adds the proper attribute to the output:

```
<xsl:template match="recordset/item">

  <tr>

    <xsl:if test="position( ) mod 2 != 1">

      <xsl:attribute name="bgcolor">#eeeeee</xsl:attribute>

    </xsl:if>

    <xsl:apply-templates/>

  </tr>

</xsl:template>
```

5.2.2.3 Discussion

Here, a combination of the `position( )` function and the `mod` operator are used to test whether to add the `bgcolor` attribute to the surrounding table row. The expression `position( ) mod 2 != 1` evaluates to true only for nodes in odd-numbered positions in the larger nodeset, so the `bgcolor` attribute is added only to every other row (starting with the first row).

5.2.3 Using XSLT Parameters with POST and GET Data

5.2.3.1 Problem

You want to make your stylesheets more dynamic by being able to access the `POST` and `GET` parameters that are part of a given HTTP request.

5.2.3.2 Solution

Use an `xsl:param` element and use the name of the desired field as the name of the parameter:

```
<xsl:param name="my-url-param"/>
```

5.2.3.3 Discussion

Unless explicitly configured to behave otherwise, all HTTP POST and GET parameters are available from within your stylesheets via top-level `xsl:param` elements. You only need to declare a top-level `xsl:param` whose `name` attribute corresponds to the name of the request field.

If you have caching turned on for documents being transformed by a stylesheet that uses a parameter extracted from the query string, be sure to add the following line to your *.htaccess* or other configuration file:

```
AxAddPlugin Apache::AxKit::Plugin::QueryStringCache
```

This plug-in makes AxKit's internal caching mechanism smarter with respect to query string parameters. More cache files are created on the disk (and therefore take more space), but it allows you to safely use the values of query string parameters inside your stylesheet without serving stale data or having to turn caching off altogether.

5.2.4 Creating a "Breadcrumb" Navigation Bar

5.2.4.1 Problem

You want to add context-sensitive navigation that allows visitors to easily climb the document hierarchy from the current folder (often called a *breadcrumb bar*).

5.2.4.2 Solution

Use AxKit's `AddXSLParams::Request` plug-in to pass the URL path of the source XML document to your XSLT stylesheet as an `xsl:param` and process that string into the desired HTML hyperlinks:

```
<?xml version="1.0"?>

<xsl:stylesheet

   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

   version="1.0">
```

```
<xsl:param name="request.uri"/>

<xsl:variable name="breadcrumb-delimiter">
  <xsl:text> :: </xsl:text>
</xsl:variable>

<xsl:template name="breadcrumb-nav">
<xsl:param name="url"/>
<xsl:param name="step-url"/>
<xsl:param name="links"/>

<xsl:choose>
  <xsl:when test="contains($url, '/')">

    <xsl:variable name="step">
      <xsl:choose>
        <xsl:when test="starts-with($url, '/')">
          <xsl:value-of select="'home'"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="substring-before( $url, '/')"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:variable>

    <xsl:variable name="local-url">
      <xsl:choose>
        <xsl:when test="$step='home'">
          <xsl:value-of select="'/'"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="concat( $step-url, $step, '/')"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:variable>

    <!-- call this template again to process the rest of the url -->
    <xsl:call-template name="breadcrumb-nav">
      <xsl:with-param name="url">
        <xsl:value-of select="substring-after( $url, '/')"/>
      </xsl:with-param>
      <xsl:with-param name="step-url">
        <xsl:value-of select="$local-url"/>
      </xsl:with-param>
      <xsl:with-param name="links">
        <xsl:copy-of select="$links"/>
        <xsl:value-of select="$breadcrumb-delimiter"/>
        <a href="{$local-url}"><xsl:value-of select="$step"/></a>
      </xsl:with-param>
    </xsl:call-template>

  </xsl:when>

  <!-- if you make it to here, all the url steps
       have been extracted. cover the last case
       and return the result -->

  <xsl:otherwise>
    <xsl:copy-of select="$links"/>
    <xsl:value-of select="$breadcrumb-delimiter"/>
    <a href="{concat($step-url, $url)}" >
      <xsl:value-of select="$url"/>
    </a>
  </xsl:otherwise>

</xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

5.2.4.3 Discussion

AxKit makes all form and query parameters available to the XSLT stylesheet via top-level xsl:param elements by default. This is not the only data that can be accessed in this way. AxKit's plug-in modules can pass *any* simple key/value pair as a stylesheet parameter. You learn more about this in Custom Plug-ins in Chapter 8, but in this case, you do not need a no custom plug-in; you can simply use the existing Apache::AxKit::Plugin::AddXSLParams::Request plug-in to pass the URI of the content source into your stylesheet. When added to the processing chain, this verbosely named class provides the ability to select sets of commonly used request and server information (HTTP headers, cookie data server hostnames, etc.) and makes them available as XSLT parameters. We will not dig into details of this plug-in's parameter groups or their naming conventions (see the installed documentation for more information), but suffice it to say, as with form and query string data, if the name of one of this plug-in's selected fields matches the name attribute of a top-level xsl:para element in your stylesheet, then the data is made available via that parameter. To access the request URI as an XSLT parameter, you need to first add and configure the plug-in via one of the web server's configuration files:

```
# Add the plug-in to the processing chain

AxAddPlugin Apache::AxKit::Plugin::AddXSLParams::Request



# Tell the plugin which group of data you are interested in

PerlSetVar AxAddXSLParamGroups "Request-Common"
```

With this in place, you can access the URI fron the current request using the following top-level parameter in your stylesheet:

```
<xsl:param name="request.uri"/>
```

Now that you can access the URL, you still need to process that string to generate the breadcrumb bar for the top of your pages. To do this, create a named template that recursively consumes the URL string and produces a series of HTML links separated by the text string defined by the $breadcrumb-delimiter variable.

A stylesheet such as the one in this example is a good candidate for inclusion in a reusable library of common templates. With that stylesheet imported, you can simply call the breadcrumb-nav template (passing the URI parameter as an argument) at the location in the page that you want the breadcrumb bar to appear. For example, the following inserts the link bar into its own div element at the top of the result's body element:

```
<?xml version="1.0"?>

<xsl:stylesheet

    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

    version="1.0">



<xsl:import href="/style/common/breadcrumb.xsl"/>



<xsl:template match="article">

<html>

  <body>

    <div class="breadcrumb">

       <xsl:call-template name="breadcrumb-nav">

         <xsl:with-param name="url">

         <xsl:value-of select="$request.uri"/>

       </xsl:with-param>

      </xsl:call-template>

    </div>

    <!-- main content continues.. -->

  </body>

</html>

</xsl:template>
```

With the template above as a part of your stylesheet, the resulting HTML generated from this snippet may look something like this:

```
<html>

  <body>

  <div class="breadcrumb">

    <a href="/">home</a> :: <a href="/samples/">samples</a> ::

    <a href="/samples/mypage.xml">mypage.xml</a>

  </div>

  <!-- main content continues.. -->

  </body>

</html>
```

The breadcrumb-nav template that does the real work is fairly verbose, and truly, you may have slimmed it down a bit by using an extension function. It does, however, illustrate the functional recursive template technique introduced earlier in this chapter in "Parameters and Variables." It underscores the fact that some tasks, most notably up-translating text strings into markup, are tricky but possible.

5.2.5 Passing Markup Through Untransformed

5.2.5.1 Problem

You have certain elements in your source documents that you want to appear untransformed in the result of a stylesheet transformation.

5.2.5.2 Solution

Create a template to match the nodes you want to copy, and use xsl:copy and xsl:copy-of.

Pass through a copy of the selected nodes, including their attributes and text children without including any descendant elements:

```
<xsl:template match="para">

  <xsl:copy>

    <xsl:copy-of select="@*|text( )"/>

  </xsl:copy>

</xsl:template>
```

Copy the selected nodes and all of their descendants as is:

```
<xsl:template match="article/articleinfo">

  <xsl:copy-of select="."/>

</xsl:template>
```

Copy the selected nodes, and pass their descendants on for further processing:

```
<xsl:template match="article/articleinfo">
   <xsl:copy>
     <xsl:copy-of select="@*"/>
     <xsl:apply-templates />
   </xsl:copy>
</xsl:template>
```

5.2.5.3 Discussion

The significant difference between xsl:copy and xsl:copy-of is that the former creates a shallow copy of the node being processed, while the latter creates a deep copy that includes all attributes, text nodes, and descendant elements. The xsl:copy element offers finer control over exactly which parts of a node are copied, but puts the onus on the stylesheet author to explicitly descend into the nodeset's structure to extract the parts that are to be copied into the result. The xsl:copy-of makes it easy to include complex nodesets in the result but relies on the fact you do not want to process any descendant nodes contained by that nodeset.

As a general rule, use xsl:copy-of when you want to include the entire nodeset without examining its contents, and a combination of xsl:copy and xsl:copy-of with xsl:apply-templates when the nodeset may contain other elements that you want to transform.

The following passes a defined subset of HTML elements through the stylesheet untransformed, while processing any non-HTML descendants according to the other templates contained in the stylesheet:

```
<xsl:template match="p|i|b|font|em|blockquote|span|pre|br|div|ul|ol|li|
                     img|script|html|head|meta|a|
                     table|tr|td|th|
                     form|input|select|option|textarea|
                     embed">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates />
  </xsl:copy>
</xsl:template>
```

Rather than listing all the elements by name as you do here, it's far more common to use what is popularly called an identity transformation template that copies through all nodes that do not have a more specific template that matches. This offers a way to operate on only the nodes that you need to, while passing the rest through undisturbed:

```
  <xsl:template match="node( )|@*">
    <xsl:copy>
      <xsl:apply-templates select="@*|node( )" />
    </xsl:copy>
  </xsl:template>
```

The template is rather common in processing chains in which an upstream stylesheet may have generated HTML (or whatever the client expects), and the last stylesheet in the chain wants to add a common header and footer to the body element while passing the rest of the content through, as is.

5.2.6 Including External Documents

5.2.6.1 Problem

You want to include all or part of another XML document in the result of an XSLT transformation.

5.2.6.2 Solution

Use the XSLT document( ) function.

Include the entire document as is:

```
<xsl:copy-of select="document('include.xml')"/>
```

Process the contents of the bookinfo element in the included document with the template rules in the current stylesheet:

```
<xsl:apply-templates select="document('book.xml')/bookinfo/*"/>
```

5.2.6.3 Discussion

XSLT's document( ) function offers a flexible mechanism for including all or some nodes in a separate XML document in the result of a transformation. It is especially useful for including common site elements such as headers, footers, and global navigation. The result of a call to document( ) is a nodeset containing all or some content of the external file. The resulting nodeset can be assigned to an xsl:variable and reused within the stylesheet:

```
<xsl:variable name="common_nav" select="document('sitenav.xml')"/>

<xsl:apply-templates select="$common_nav/*" mode="rightnav"/>

  . . .

<xsl:apply-templates select="$common_nav/*" mode="footerenav"/>
```

You can use the document( ) function for much more than adding chunks of common boilerplate. For example, you can include details about the products in a customer's order by looking up the product details in a separate file:

```
<xsl:template match="/">

  <xsl:variable name="item_details"

               select="document('productdetail.xml')"/>


  <xsl:for-each select="order/item">

    <xsl:variable name="lookup">

      <xsl:value-of select="@product_id"/>

    </xsl:variable>

    <item product_id="{$lookup}">

      <xsl:copy-of select="$item_details/productlist/item[@product_id=$lookup]/*"/>

    </item>

  </xsl:for-each>

  . . .

</xsl:template>
```

The way the document( ) function treats relative URIs often trips the unwary. Stylesheet authors typically expect the file path to be evaluated in the context of the document being transformed—it is not. It is processed in the context of the stylesheet, not the source document. When you type select="document('header.xml')", remember that the XSLT processor looks in the same directory as the stylesheet for the file *header.xml*.

You can alter this default behavior, however, by using the two-argument form of document( ). In the two-argument form, the second argument is a nodeset. The URI of the source document that originally contained that nodeset is used to set context in which any relative path contained by the first argument is evaluated.

To resolve a relative file path in the context of the source document, rather than the stylesheet, use:

```
<xsl:copy-of select="document('widgets/footer.xml', /)"/>
```

This includes the contents of the file *footer.xml* from the directory *widgets* in the same directory as the source document. It works because the second argument (/) represents the root of the document currently being transformed. Therefore, the relative path contained in the first argument is resolved in the context of that URI.

5.2.7 Dividing Large Datasets into Pages

5.2.7.1 Problem

You have a large XML source document containing many records. You want to display only a few at a time.

5.2.7.2 Solution

Use the XPath position( ) function to select a smaller subset of the records stored in the document.

Use position( ) in the select expression of an xsl:for-each loop:

```
<xsl:template match="/">
<recordset>
  <xsl:for-each select="recordset/record[position ( ) >= $start and position( )
<= $end]">
    <xsl:copy-of select="."/>
  </xsl:for-each>
</recordset>
```

5.2.7.3 Discussion

Displaying a limited set of all records contained in a large document (*pagination*) is one of the most common XML publishing tasks. Selecting a given set of records is as easy as selecting the proper element nodes whose position falls within the desired range.

Using top-level xsl:param elements that allow the range and context to be passed dynamically to the stylesheet via the query string makes navigating large documents fairly straightforward. Example 5-1 demonstrates one way to navigate a large set of records using the type of interface popularized by major web search engines.

Example 5-1. An XSLT stylesheet for paginating large records

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="html" />

<xsl:param name="start">1</xsl:param>
```

```
<xsl:param name="perpage">10</xsl:param>

<xsl:variable name="totalitems" select="count(//item)"/>

<xsl:variable name="end">

  <xsl:choose>

    <xsl:when test="($start + $perpage) > $totalitems">

      <xsl:value-of select="$totalitems"/>

    </xsl:when>

    <xsl:otherwise>

      <xsl:value-of select="$start + $perpage - 1"/>

    </xsl:otherwise>

  </xsl:choose>

</xsl:variable>


<!-- begin root template -->

<xsl:template match="/">


  <h3>

    Showing records

    <xsl:value-of select="$start"/> - <xsl:value-of select="$end"/> of

    <xsl:value-of select="$totalitems"/>

  </h3>


  <table border="0" cellpadding="0" cellspacing="0">

    <tr><th>Product ID</th><th>Description</th><th>Price</th></tr>

    <xsl:for-each select="/order/item[position( ) >= $start and position( ) <= $end]">

      <tr>

        <!-- make every other row gray -->

        <xsl:if test="position( ) mod 2 != 1">

          <xsl:attribute name="bgcolor">eeeeee</xsl:attribute>

        </xsl:if>

        <td><xsl:value-of select="@product-id"/></td>

        <td><xsl:value-of select="name"/></td>

        <td><xsl:value-of select="price"/></td>

      </tr>

    </xsl:for-each>

  </table>


  <!-- if there are records before the block you are viewing, provide a 'prev.' link -->

  <xsl:if test="$start > 1">
```

```
    <a href="paginate.xml?start={$start - $perpage};perpage={$perpage}">prev.</a>
  </xsl:if>

  <!-- process *all* the <item> elements in the document to
       build the navbar -->
  <xsl:apply-templates select="/order/item"/>

  <!-- if there are more records, provide a 'next' link -->
  <xsl:if test="$totalitems > $end">
    <a href="paginate.xml?start={$end + 1};perpage={$perpage}">next</a>
  </xsl:if>

</xsl:template>
<!-- end root template -->

<!-- the 'item' template that builds the numbered navbar links -->
<xsl:template match="item">
  <xsl:if test="position( ) mod $perpage = 1 or $perpage = 1">
    <xsl:variable name="pagenum">
      <xsl:value-of select="ceiling(position( ) div $perpage)"/>
    </xsl:variable>
    <a href="paginate.xml?start={position( )};perpage={$perpage}">
      <xsl:value-of select="$pagenum"/>
      <!-- force whitespace in between the numbered links -->
      <xsl:text> </xsl:text>
    </a>
  </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Remember, XSLT is not, nor does it pretend to be, a general-purpose programming language. If the XML documents you are publishing require a significant amount of processing that is beyond the reach of XSLT, or if you are bending over backwards too often to get XSLT to do what you need, consider using XPathScript to transform your documents. Similar to XSLT in many respects, XPathScript adds the ability to process the data contained in your documents using Perl and its myriad functions and modules.

Chapter 6. Transforming XML Content with XPathScript

XSLT is a good solution for many cases, but it is certainly not the only game in town when it comes to transforming XML documents. XSLT does a fantastic job of performing recursive transformations of tree-shaped data structures based on declarative rules, but it's not a general-purpose programming language, and the solutions to some common web-publishing tasks (most notably, anything involving text processing) can seem cumbersome and obtuse. Ironically, the people who seem to have the hardest time picking up XSLT are more experienced web developers with a strong history in a language such as Perl or Python. It's not that XSLT is bad or wrong; it just relies on patterns unfamiliar to many web coders, and many approaches that work just fine in Perl simply do not fit into XSLT's model.

Created expressly to bridge the gap between "just print it, already" web-programming techniques and the tree-based operations found in most XML processing tools, the XPathScript language seeks to offer the best of both worlds—powerful declarative rule-based transformations such as those available in XSLT, combined with full access to the flexibility of Perl and its many modules.

Here are some reasons why you may want to use XPathScript:

- You know Perl and do not want to learn XSLT just to work with AxKit.
- Your stylesheets would benefit from dynamic transformations based on external libraries, fine-grained access to the Apache server environment, database queries, etc.
- Using XSLT is working out fine for the most part, but you want to pre- or postprocess your documents in a Perlish way to simplify or avoid certain hairy transformations.

Essentially, if you have a strong background in Perl development and the prospect of learning XSLT to transform XML content in AxKit seems undesirable—or if the types of transformations you are doing seem beyond the reasonable reach of XSLT— then XPathScript is probably the tool for you.

6.1 XPathScript Basics

An XPathScript document's basic syntax is similar to that found in Active Server Pages and similar technologies, where literal output is separated from functional code blocks using the familiar <% . . . %> pseudotags as delimiters. As with Apache::ASP, the embedded language is Perl and the mixture of code and markup is processed, returning the content as it is encountered from the top down, while replacing the code blocks with the markup they return to produce the complete result:

```
<html>
 <body>
 <p>Greetings. The local time for this server is <% print scalar localtime; %>
 </p>
 </body>
</html>
```

Appending an equal sign «=» to the opening delimiter offers a shortcut that sends the interpolated values returned from any code it encloses directly to the output. The following would send the same result to the browser as shown in the previous example (the print statement is gone):

```
<html>
 <body>
 <p>Greetings. The local time for this server is <%= scalar localtime %>
 </p>
 </body>
</html>
```

XML elements and attributes can be added to the result from within code blocks by simply returning the appropriate textual representation:

```
 <select name="day">
 <%
   foreach my $day ( qw(Sun Mon Tues Wed Thu Fri Sat) ) {
      print "<option value='$day'>$day</option>";
   }
 %>
 </select>
```

In terms of lexical scoping, an XPathScript stylesheet behaves like any Perl script in that all variables and subroutines not enclosed within a block, or explicitly declared as part of another package, become part package::main and are available throughout the entire stylesheet. Among other things, this allows you to break your stylesheets into logical sections that can provide a somewhat cleaner division between functional code and markup.

```
<%
my @fish_names = qw( tuna halibut salmon scrod );
%>
<html>
  <body>
    <form>
      <p>Choose Your Favorite Fish</p>
      <select name="fave_fish">
      <%
         # @fish_names is still in scope here
         foreach my $fish ( sort(@fish_names) ) {
            print "<option value='$fish'>$fish</option>";
         }
      %>
      </select>
    </form>
  </body>
</html>
```

Any similarity between XPathScript and various Server Pages technologies ends at this superficial syntactic level. Remember, XPathScript was designed as an alternative to XSLT, and as such, it provides both a means for selecting and extracting information for an XML source as well as a means for creating and applying declarative rules for transforming that content—that is, unlike ASP, XPathScript documents are *stylesheets* that are applied to a source XML document to create a transformed result, not simply a source of dynamically generated content. The typical XPathScript stylesheet takes the form shown in Example 6-1.

Example 6-1. Structure of a typical XPathScript stylesheet

```
<%
    # Perl code block that imports any required modules,
    # and performs any required initialization
    use Some::Package;

    # Declarative template rules defined via the
    # special "template hash" $t
    $t->{some:element}->{pre}  = '<new>';
    $t->{some:element}->{post} = '</new>';
    # and so on..
 %>
  <root>
      <!-- begin literal output -->
    <child>
      <%
        # A mixture of literal output and code blocks that
        # call XPathScript functions to select data from
        # or apply templates to the source XML
      %>
    </child>
  </root>
```

From this, you can see that a typical XPathScript stylesheet consists of two parts: a Perl code block that contains any initialization required for the current transformation as well as the template rule configuration via XPathScript's special template hash, $t, and a block of literal output that provides the overall structure of the result and contains escaped code blocks through which the contents of the source XML document are accessed.

6.1.1 Accessing Document Content

Like XSLT, XPathScript uses the XPath language as the mechanism for selecting and evaluating the nodes in an XML document. (We touched on the basics of XPath in Section 5.1 in Chapter 5, and I will not duplicate that introduction here.) Recall that XPath provides a compact syntax for accessing and evaluating the contents of an XML document using a combination of location paths and function evaluation. In XPathScript, these XPath expressions are passed as arguments to one of a handful of Perl subroutines, implemented by the XPathScript processor, that perform the desired actions. The following sections provide an introduction to each of these subroutines.

6.1.1.1 findvalue( )

As the name suggests, the *findvalue* function offers a way to select the value from a given node in the source XML document. The required first argument to this function is the XPath location path that will be evaluated to select the node whose value will be returned.

```
<html>
  <head>
    <title><%= findvalue('/article/artheader/title') %></title>
  </head>
</html>
```

An optional second argument may be passed to the findvalue( ) function. This argument is expected to be a node from the current document and will be used to provide the context in which the expression contained in the first argument will be evaluated.

```
<%
foreach my $chapter ( findnodes('//chapter') ) {
  # select the title of each chapter
  my $title = findvalue('title', $chapter);
}
%>
```

This can be useful, but it is far more common to use the object-oriented interface, calling *findvalue* on the nodes themselves, rather than passing that node as the second argument. Calling *findvalue* as a method on the node has the effect of limiting the value returned to the contents of that specific node and gives the same result as calling *findvalue* as a function and passing in the given node as the second argument:

```
<%

foreach my $chapter ( findnodes('//chapter') ) {

    # Same as above

    my $title = $chapter->findvalue('title');

}

%>
```

As with XSLT's xsl:value-of element, if more than one node is returned from evaluating the given expression, the text descendants of all nodes selected will be concatenated (in document order) into a single string. For example, applying findvalue('para') to the following XML snippet:

```
<para>

    Well, <emphasis>I</emphasis> wouldn't say that.

</para>
```

produces this somewhat more ambiguous snippet:

```
Well, I wouldn't say that.
```

Unlike other XPathScript subroutines designed to select nodes from the source XML document, XPath expressions passed to *findvalue( )* are not limited to location path expressions:

```
<p>

    This document contains <%= findvalue('count(//section)') %> sections.

</p>
```

6.1.1.2 findnodes( )

While *findvalue( )* is used for retrieving the values of the selected nodes, the *findnodes( )* function is used to select the nodes themselves. In a list context, *findnodes( )* returns a list of all nodes that match the XPath expression passed as the first argument; in a scalar context, it returns an XML::XPath::NodeSet object. As with *findvalue( )*, a node may be passed as the optional second argument to constrain the context in which the XPath expression will be evaluated.

```
# print the 'id' attributes from all 'product' elements calling findnodes( ) in a list

context

foreach my $product ( findnodes('/productlist/product') ) {

    print $product->findvalue('@id') . "\n";

}


# the same, but calling findnodes( ) in a scalar context to return an XML::XPath::NodeSet

object

my $products = findnodes('/productlist/product');


foreach my $product ( $products->get_nodelist ) {
    print $product->findvalue('@id') . "\n";

}
```

6.1.1.3 findnodes_as_string( )

The *findnodes_as_string( )* function works just like *findnodes( )*, but it returns the textual representation of the selected nodes rather than a nodeset object. It is used primarily to copy larger document fragments, as is, into the result of the transformation. For example, the following copies the entire contents of the file */include/common_header.xhtml* into the result of the current process:

```
<body>

   <!-- insert the common header -->

   <%= findnodes_as_string('document("/include/common_header.xhtml")') %>

    . . .

</body>
```

Taken together, these subroutines provide direct access to the contents of the source XML document from within a stylesheet's code blocks. They are most useful for cases in which the types of transformations required would benefit from the ability to use Perl's control structures, operators, and functions to generate the result based on nodes from the source document. However, this is only half of the story; XPathScript also offers the ability to set up and apply template rules that will generate new content as each element node in a given set is visited by the XPathScript processor.

6.1.2 Declarative Templates

Declarative templates provide a way to easily set up transformations for all instances of certain elements in the source document. Template rules are defined by adding keys to a global Perl hash reference, $t. During processing, the names of selected nodes are evaluated against the key names in the hash that $t refers to. When a match is found, the node is transformed based on the contents of that template.

The result of applying a template to a matching element is governed by the values assigned to one or more of a specific set of subkeys for each key in the template hash. For example, the value assigned to the pre subkey is added to the output just *before* the matching element is processed, while the value assigned to post key is appended to the output just *after* the node is processed. Adding the following to your XPathScript stylesheet has the effect of replacing all section elements with div elements with a class attribute with the value section:

```
<%

$t->{'section'}{pre} = '<div class="section">';

$t->{'section'}{post} = '</div>';

%>
```

If you are familiar with XSLT, the above corresponds to the following template rule:

```
<xsl:template match="section">

    <div class="section">

        <xsl:apply-templates/>

    </div>

</xsl:template>
```

XPathScript template matches are based solely on the element's name rather than the evaluation of more complex XPath expressions used in XSLT. For example, the first template below matches all instances of the product element, while the second would *never* match, even though a node may exist in the source document that matches the given expression:

```
<%

# convert <product> elements to list items

$t->{'product'}{pre} = "<li>";

$t->{'product'}{post} = "</li>";


# will never match since only element names are tested

$t->{'/products/product'}{pre} = "<li>";

$t->{'products/product'}{post} = "</li>";

%>
```

The fact that matches are based on the element name only is the major difference between declarative templates in XPathScript and those in XSLT. While this may appear to limit XPathScript's power, you will see that XPathScript provides other facilities for more fine-tuned control over the templates' matching behavior. (See Section 6.2, later in this chapter.)

XPathScript is not implemented in XML, so there is really no way to bind an XML namespace URI to a prefix, according to the rules of XML. This means that if you are creating template rules or location expressions to match element and attribute names that use XML namespaces and prefixes, you must use the same literal string prefix used in the source document. Compare this to XSLT, in which a prefix in the stylesheet and the one in the source document can vary as long as they are bound to the same URI.

Now you have a general idea of how to create template rules. Let's examine how to apply those templates to the source content.

6.1.2.1 Applying templates

Templates are applied to the source content using the *apply_templates( )* function. This function accepts a single optional argument that can either be an XPath expression that selects the appropriate nodes to be transformed, or an XML::XPath::NodeSet object containing those nodes. If no argument is passed, the templates are tested for matches against *all* nodes in the current document.

```
# Apply template rules to all <product> elements and their children

print apply_templates('/productlist/product');


# The same, but pass a nodeset instead

my $product_nodes = findnodes('/productlist/product');

print apply_templates( $product_nodes );
```

The XPath expression passed to the *apply_templates* function can be as specific as the case requires, and Perl scalars will be interpolated if the expression is passed as a double-quoted string. Also, the «@» must be escaped in the following, so the processor does not confuse the XPath attribute selection syntax with a Perl array and attempt to interpolate it (this is not needed if the expression is passed as a single-quoted string):

```
<%

my $id = $some_object->get_id( );

my $product = findnodes("/productlist/product[\@id=$id]");


if ( $product->size( ) > 0 ) {
    print apply_templates( $product );

}

else {

   print "<p> Sorry, no product ID match for $id in the current document.</p>";

}

%>
```

When applying templates, you must use Perl's built-in *print* function or the <%= opening delimiter shortcut for the result of the template processing to appear in the final output:

```
<table>

<%

 # part of a much longer Perl block

 if ( $some_condition ) {

     print apply_templates('/productlist/product');

 }

%>

</table>


# The same, but using the <%= delimiter shortcut


<table>

    <%= apply_templates('/productlist/product') %>

</table>
```

In both cases, the result of any template processing appears in the final result at the same location that the print statement or special opening delimiter is found.

6.1.2.2 Importing templates

The *import_template* function provides a way to add template rules to the current stylesheet by importing an external stylesheet. This function's single required argument is the DocumentRoot-relative path to the stylesheet that you want to import.

```
<%

import_template('/path/to/included.xps')->( );

%>
```

Since an XPathScript stylesheet is really a sort of fancy preprocessed Perl script, and the template hash is a global variable within that script, imported templates have the potential to override or add to template rules defined in the current stylesheet. That is, the element hashes will be merged, but since the template hash can only have one unique subkey for each template rule, if your stylesheet has a rule such as $t->{'para'}{pre} = '<p>';, and the imported stylesheet also has a rule to match para elements and defines a pre subkey, then the subkey definition that appears *last* in the stylesheet (e.g., the one that assigns a value to the given subkey last) takes precedence. If different subkeys are defined for the same element match, both subkeys are applied during processing.

6.1.2.3 Expression interpolation

Template rules may also contain information selected from the document tree by adding XPath expressions delimited by curly braces «{ }» to the rule's value. The contents of the braces are passed to the *findvalue( )* function in which the expression is evaluated in the context of the current matching node, and the result is added to the output. For example, the following copies the value of the url attribute from the current <ulink> element into the href attribute of the newly created HTML hyperlink:

```
$t->{ulink}{pre}  = '<a href="{@url}">';

$t->{ulink}{post} = '</a>';
```

To save resources, this sort of interpolation is not turned on by default. It must be explicitly switched on by adding the AxXPSInterpolate configuration directive to your *httpd.conf* or other server configuration file:

```
AxXPSInterpolate 1
```

6.2 The Template Hash: A Closer Look

Understanding how the template hash works is crucial to the effective use of XPathScript. First, the template hash is a bit magical in that (unlike all other Perl variables you may use in your stylesheet) you need not initialize this hash in your stylesheet. It is declared invisibly during execution by the XPathScript processor. This means that you cannot safely initialize a scalar named $t to the top level of your XPathScript stylesheets without causing a conflict. Second, the template hash really takes the form of a hash of hashes. The names given to the top-level keys define the element names that will be matched when *apply_templates( )* is called, and the values for those keys are themselves hashes whose predefined keys determine how the given element will be processed if a match is found. A typical rule takes the following form:

$t->{<element name>}{<sub-key name>} = $some_value;

element name is the full name (including the namespace prefix) of the element you want to match, and *sub-key name* is one of eight special keys that the XPathScript processor uses to determine how to build the output for matching cases.

Here are all these special subkeys and their associated behaviors:

pre

The scalar value assigned to this key is added to the output just before the matching element is processed.

post

The value assigned is appended to the output just after the matching element is processed.

prechildren

The scalar value assigned is sent to the output before any child elements of the matching element are visited.

postchildren

The scalar value assigned is added to the output after all children of the matching element are processed.

prechild

The value assigned is added to the output before *each* child of the matching element is processed.

postchild

The value assigned is added to the output after each child of the matching element is processed.

showtag

If defined, this key has the effect of copying the current matching element's start and end tags into the output. By default, the start/end tags for all matching elements are stripped.

testcode

The value assigned to this key is expected to reference a subroutine that controls how the matching element is processed.

6.2.1 The Joys of testcode

The testcode key is far and away the most powerful and interesting of the template rule subkeys. The value assigned to this key references a Perl subroutine that handles the output. Each time a match is found for the top-level key, this subroutine is called and passes two arguments: the element node for the current match and a localized template hash reference that can be used to set the other subkeys for the current template. The private template hash can be used in the same way as the global template hash, but its effects are limited to the current node, and there is no need to add the top-level key containing the element name.

```
# Set the pre and post keys for an <item> elements template in the typical way

$t->{item}{pre} = '<li>';

$t->{item}{pre} = '</li>';



# The same, but via testcode

$t->{item}{testcode} = sub {

    my ($node, $template ) = @_;

    $template->{pre}  =  '<li>';

    $template->{post} = '</li>';

};
```

> Perl code executed inside the *mod_perl* environment needs to be a bit stricter than that allowed in typical CGI scripts. Specifically, you should avoid creating closures (subroutines that reference lexical variables outside the scope of the subroutine itself) when creating your testcode subroutines, or you will certainly get unexpected results from your XPathScript stylesheets. For more detailed information about closures and *mod_perl*, see the reference pages at http://perl.apache.org/docs/general/perl_reference/perl_reference.html#.

The testcode option also provides a way to conditionally process a given element based on the properties of the element node itself. In XSLT, this kind of property examination often happens when the XPath expression for a given template rule is evaluated, but in XPathScript, template matches are made only against the element name, and you need to examine the node properties (or other conditions) from within the template's testcode subroutine.

Suppose that you are transforming DocBook XML documents into HTML. Even if you are using the simplified subset (*sdocbook.dtd*), the title element can validly appear in 22 different contexts; thus, a single, simple template rule based on the element name cannot suffice. The following shows how you could begin to approach the problem:

```
$t->{title}{testcode} = sub {

    my ($node, $template ) = @_;



    # Get the parent element's name

    my $parent_name = $node->findvalue('name(..)');



    # now alter the local template hash to cover the various cases.

    if ( $parent_name eq 'section' ) {

        my $level = $node->findvalue('count(ancestor::section)');

        $template->{pre}  = "<h$level>";

        $template->{post} = "</h$level>";

        return 1;

    }

    elsif ( $parent_name =~ m/sect(\d+)$/ ) {

        my $level = $1;
        $template->{pre}  = "<h$level>";
        $ttemplate->{post} = "</h$level>";
         return 1;
    }
    else {
        return 0;
    }
};
```

Here, you grab the name of the current title element's parent element and examine that value to provide the context for your conditional processing. If the parent is a section element, you count the number of nested sections and use that to determine the level for the HTML header element that will be used to render the content. If the parent name matches the regular expression /sect(\d+)$/, the level of the header is defined by extracting the numeric value from the element name itself (i.e., sect1, sect2, etc.). Obviously, this does not cover the 22 possible cases in which a title element can appear, but it provides a good start that you can build on later, as needed.

You return different values from your title element's testcode subroutine, depending on the context. This is an important option, since the value returned from the testcode subroutine controls how the XPathScript processor reacts when a matching node is found. A return value of 1 indicates to the XPathScript processor that you want to process the current node and its children, while returning 0 states that you do *not* want to process the current node (nor its children). The following is the list of possible testcode return values:

1

The default behavior for all templates; returning 1 from the testcode subroutine tells the XPathScript processor to process the current node and descend into any child nodes for additional template matches.

-1

A return value of -1 signals the XPathScript engine to process the current node but skip all children and their descendants.

0

Returning 0 from any point in the testcode subroutine tells the XPathScript processor to skip the current node and its descendants altogether.

$string

If a string is returned from the testcode subroutine, it is expected to contain a valid XPath expression that will be used to select the next set of nodes to process.

Given the access that the testcode option provides to the node being selected and the variety of return codes that control what the XPathScript processor does *after* the associated subroutine has been executed, you can safely say that XPathScript's element-name-only template matching rule behavior is in no way inferior to XSLT's template matching rules (in which more sophisticated XPath expressions may be used to decide when a template is applied). The only difference is that, in XPathScript, any additional node properties are examined via the given testcode subroutine, not tested by the match rule itself.

6.2.2 The Special "Catch-All" Template

There is one exception to the rule that template matches can only be made against element names: the special "catch-all" template that is invoked by using a top-level key in the template hash with the name «*». The subkeys added to the catch-all template are invoked for every element node that does not already have an explicit entry in the template hash. Among other uses, this allows you to explicitly prune all elements from the current node that do not have an associated template rule.

```
<%

# Main template rules here . . .



# But block all unexpected elements

$t->{'*'}->{testcode} = sub { return 0; };

%>
```

If you want to be a little fancier, you can log the rogue elements in the Apache *error.log* for debugging during development:

```
<%

# Block and log all unexpected elements

$->{'*'}->{testcode} = sub {

    my ($node, $template) = @_;

    AxKit::Debug(10, "Unexpected element '" . $node->getName . "'

found during XPathScript transformation.");

    return 0;

};

%>
```

6.3 XPathScript Cookbook

Just as we concluded our look at XSLT with a few recipes that offer solutions to common tasks, we will do the same here with XPathScript. Since most XSLT tips are easy to re-create using XPathScript, I will not repeat the same recipes but will focus instead on things unique to XPathScript (or simply not possible with vanilla XSLT 1.0).

6.3.1 Accessing Client Request and Server Data

6.3.1.1 Problem

You need to access information about the current request (POSTed form data, cookies, etc.) or other parts of the Apache HTTP server API from within your stylesheet.

6.3.1.2 Solution

Use the Apache object that is passed as the first argument to the top level of every XPathScript stylesheet transformation.

```
<%

# at the top lexical level of your XPathScript stylesheet:

my $r = shift;

# $r now contains the same Apache object passed to mod_perl handler scripts.

%>
```

6.3.1.3 Discussion

One benefit of running XPathScript inside of AxKit is that each stylesheet can directly access the Apache server environment via the same Apache object that gives *mod_perl* its power and flexibility. This object can be used to examine (and in many cases, control) virtually every aspect of the Apache HTTP Server, from user-submitted form and query data to outgoing headers and host configurations.

6.3.1.4 Accessing form and query data

```
<%

use Apache::Request;

my $r = shift;

my $cgi = Apache::Request->instance($r);

%>

<html>

  <body>

    <p>

     You said that your favorite fish is <%= $cgi->param('fave_fish') %>

    </p>

  </body>

</html>
```

6.3.1.5 Setting cookies

```
<%
use Apache::Cookie;
my $r = shift;
my $out_cookie = Apache::Cookie->new( $r,
    -name => 'mycookie',
    -value => 'somevalue'
);
$out_cookie->bake;
%>
<html>
    <!--Stylesheet contents  -->
</html>
```

6.3.1.6 Redirecting the client

```
<%
use Apache::Constants qw(REDIRECT OK);
my $r = shift;

if ( $some_condition =  = 1 ) {
    $r->headers_out->set(Location => '/some/other.xml');
    $r->status(REDIRECT);
    $r->send_http_header;
}
%>
<html>
    <!--Default stylesheet contents  -->
</html>
```

6.3.2 Generating Fresh Dynamic Content

6.3.2.1 Problem

Your XPathScript stylesheet generates or transforms content based on runtime logic, but AxKit is serving the cached result of a previous transformation.

6.3.2.2 Solution

Pass a nonzero value to the *no_cache( )* method on the Apache object to turn off caching for the current resource:

```
<%

my $r = shift;

$r->no_cache( 1 );

%>

<html>

    <!--Default stylesheet contents  -->

</html>
```

6.3.2.3 Discussion

AxKit's aggressive default caching behavior helps keep it speedy and easy to set up for many common publishing cases, but sometimes, your stylesheets will need to generate or transform content based on data unavailable until runtime. In these cases, the stylesheet behaves as expected for the first request, but once the cache is created, the result of all transformations in the current processing chain is sent directly to the client, and the stylesheet containing the dynamic logic is not applied again until the source XML file (or one of the stylesheet documents) is changed.

Setting the AxNoCache configuration directive to On can do the trick, but setting it up on a resource-by-resource basis can be cumbersome. It gets even trickier if several possible transformation chains can be applied to a given resource and only some of them could benefit from turning caching off altogether.

The most direct way to ensure that your dynamic stylesheet is always applied is to simply pass a true value to $r->no_cache( ) from within your XPathScript stylesheet itself. Be aware that any resources transformed using that stylesheet will *never* be cached (and the entire transformation chain will be run for each request), but if you're generating or transforming content dynamically, that is probably what you want to happen in any case.

6.3.3 Importing Templates Dynamically

6.3.3.1 Problem

You want to apply different sets of template rules based on runtime data or aspects of document content beyond the root element name or DOCTYPE declaration.

6.3.3.2 Solution

Use a simple wrapper stylesheet containing a call to the *import_template( )* function, and generate the path to the imported stylesheet dynamically:

```
<%

my $r = shift;

$r->no_cache( 1 );



my $import_path = '/styles/myapp/';



# Select the import based on the presence of an <error>

# element anywhere in the source document.



if ( findvalue('//error') ) {

    $import_path  .= 'error.xps';
}
else {
    $import_path .= 'default.xps';
}

import_template($import_path)->( );
%>
<html>
    <%= apply_templates( ) %>
</html>
```

6.3.3.3 Discussion

The combination of AxKit's styling directives and StyleChooser plug-ins offers the ability to create chains of transformations, then select the appropriate chain to apply at request time based on a wide variety of environmental factors (the type of client making the connection, the root element name of the source XML document, etc.). However, sometimes there are cases in which explicitly defining (then selecting) a processing chain for all possible sets of conditions can get tedious.

Using the AxAddDynamicProcessor directive (that lets you write a Perl class that will generate the steps in the processing chain dynamically, at request time) offers a solution. However, it can seem like overkill for a lot of cases, and even then, you cannot readily access the content of the source XML document. Letting logic in the XPathScript stylesheet itself decide which set of imported templates to apply allows you to set up a single processing chain while still applying the appropriate styles in response to the environment.

6.3.4 Tokenizing Text into Elements

6.3.4.1 Problem

You have elements in your source XML document with text content that you want to tokenize into child elements or mixed content (containing both text data and child elements).

6.3.4.2 Solution

Use a template with a testcode subroutine that operates on the element's text children directly, and use simple Perl text-handling logic to create the new elements:

```
$t->{'my:element'}{testcode} = sub {
    my ($node, $t) = @_;

    foreach my $child ( $node->getChildNodes ) {
        if ( $child->getNodeType =  = TEXT_NODE ) {
            my $text = $child->getValue( );
            # Process $text as a string, adding pointy brackets as needed
            # to create tokenized  mixed content
            print $text;
        }
        else {
            # otherwise, apply templates manually to the child node

            apply_templates( $node );

        }

        # You have already processed this node and its children, so tell

        # the XPathScript processor not to.

        return 0;

}
```

6.3.4.3 Discussion

Carving up text nodes into tokenized mixed content, sometimes called *up-translation*, can be a tricky proposition (especially using XSLT), but it is not as crazy as it may sound. For example, you may be given documents that contain dates marked up as a date element containing a single text child, but it would simplify processing if the year, month, and day components were defined individually:

```
<!-- What you get -->

<date>2003-06-07</date>



<!-- What you wish you were getting -->

<date>

    <year>2003</year>

    <month>06</month>

    <day>07</day>

</date>
```

Here, the solution is to select the value of the date element, split that value on the «-» character using Perl's built-in *split* function, then print the appropriate textual representation of the new elements:

```
$t->{date}{testcode} = sub {

    my ($node, $t) = @_;

    my $date_string = $node->getValue( );

    my ($year, $month, $day) = spilt(/-/, $date_string);

    print "<date><year>$year</year><month>$month</month><day>$day</day></date>";

    return 0;

};
```

You must return 0 from the testcode subroutine, since you have already printed all the output you need for the current element, and you do not want the XPathScript processor to descend into the old text node and try to add it to the output. Be aware, too, that by taking control of the entire output and returning 0 from the testcode subroutine, you render the other template options (pre, post, prechild, showtag, etc.) useless for the current element. That is, by printing the output and returning 0 you are essentially saying to the XPathScript processor "Do not process this node. I want to do it myself," so the other rules are never applied.

Things get more interesting when the element in question may already have mixed content, and you need to make sure that the child nodes are processed in the typical way. In this case, you need to manually iterate over all the children of the current node and examine the node type of each child. You can then operate on the text nodes, as needed, while explicitly calling *apply_templates* for all other types of nodes.

Example 6-2 illustrates how to cope with the task of tokenizing character data when the elements being transformed already have a mixed content model. This stylesheet examines the text contents of all p elements and their children for the presence of a keyword passed in via a posted form or query string parameter named matchword. A simple Perl regular expression is used to find the matches. When one is found, it is replaced in the result by the same word wrapped in a span element whose style class renders that word in the browser in bold text with a yellow background.

Example 6-2. XPathScript stylesheet for highlighting keywords in XHTML

```
<%
use Apache::Request;
my $r = shift;
$r->no_cache( 1 );
my $cgi = Apache::Request->instance( $r );

sub tokenizer {
    my ($element, $t) = @_;

    if ( my $match_word = $cgi->param('matchword') ) {
        my $name = $element->getName( );

        print "<$name>";
        foreach my $node ( $element->getChildNodes ) {
            if ( $node->getNodeType( ) =  = TEXT_NODE ) {
                my $text = $node->getValue( );
                $text =~ s|\b($match_word)\b|<span class="matched">$1</span> |g;
                print $text;
            }
            else {
                print apply_templates( $node );
            }
        }
        print "</$name>";
        return 0;
    }
    else {
        $t->{showtag} = 1;        return 1;
    }
}

# apply the 'tokenizer' sub to all 'p' elements
# and their children
$t->{'*'}{testcode} = sub {
    my ( $node, $t ) = @_;
    if ( $node->findnodes('ancestor-or-self::p') ) {
       tokenizer( $node, $t );
    }
    else {
        return 1;
    }
};
%>
<html>
  <head>
    <style>
    span.matched { background: yellow; font-weight: bold;}
    </style>
  </head>
  <body>
  <form action="highlight.xml" method="GET">
    <input name="matchword" type="text">
    <input type="submit">
  </form>
  <%= apply_templates( ); %>
  </body>
</html>
```

Certainly, XPathScript is not to everyone's taste. For some, its visible reliance on Perl and the fact that it does not use an XML syntax are enough to send them scrambling for the hills. For others, having an XML syntax for their processing tools (in addition to the content source) is going too far, and those very same properties are what make XPathScript desirable. Reality is that one size never fits all. The fact that AxKit supports a variety of transformation languages is part of what makes it a viable, professional development environment.

Similarly, do not let the fact that XSLT and XPathScript were featured in the last two chapters dissuade you from investigating the other options that AxKit makes available. The goal here has been simply to introduce two of the more popular choices in an effort to give you the means to be productive with AxKit as quickly as possible, not to recommend one technology over another. For example, you may find that one of the other generic transformation tools such as SAX filter chains (Apache::AxKit::Language SAXMachines) or Petal (Apache::AxKit::Language::Petal), is better suited to your needs. In any case, remember that picking one transformative Language module does not mean abandoning all others—AxKit allows you to pick the tool that best suits the task, even mixing different languages within a single processing chain.

Chapter 7. Serving Dynamic XML Content

So far, we have examined how AxKit can be used to transform static XML documents. Transforming such documents via XSLT stylesheets or another means that can alter, create, or include content conditionally provides a certain level of dynamism, but most modern web sites include features or resources that need to change for each unique request. An altered version of a static source is not enough, the *source itself* must be generated programatically.

By now, you are surely aware of the value of separating content from presentation, and how that value increases when the documents being served are marked up using semantically rich grammar that intimately captures (or communicates) the meaning of the information they contain. The same principle holds true when creating XML-based applications—the key difference is that markup generated in an application context should attempt to most accurately reflect the structure and roles of the resources associated with (or required by) the current state of the application, rather than those of a static narrative document.

By generating only the data relevant to the current state of the application, developers working on the backend libraries have simpler and better defined targets to hit. Since the markup being generated does not include any presentational elements, the grammar for the document being produced is usually greatly simplified. Among other things, this means that the generated content can be evaluated in detail using nothing more than a validating XML parser and a Document Type Definition (for instance, one of the XML schema languages) to verify the result. Hence, acceptance testing can become fully deterministic—the coder knows her job is done when the generated grammar passes a validity test, not when it seems to render correctly in one or another client application.

Similarly, having a simplified, well-defined application grammar offers stylesheet authors the ability to develop and test their stylesheets using sample documents that accurately reflect the data that will be generated when the application goes into production. This means that application development can push ahead asynchronously—the stylesheets that will render the application content can be completed at the same time (or even before) the code that generates the content is written.

Also having the ability to reuse the same content to meet the needs and expectations of a variety of clients is a key benefit of an XML application environment. Being able to deliver the same essay as either HTML or PDF is a cool feature; being able to offer the same online *application* to a variety of clients can directly affect the profitability of a web-based enterprise.

In short, the benefits of separating content from interface are especially noticeable when the markup in question is being generated dynamically in the context of an online application. In this chapter, I introduce a few of the tools and techniques that AxKit offers for creating content dynamically and explain how applying transformative styles to that generated content can be used to create flexible, sophisticated online applications.

7.1 Introduction to eXtensible Server Pages

Originally created by the developers of AxKit's sister Apache project, Cocoon, eXtensible Server Pages (XSP) offers an XML-centric implementation of the Server Pages model. Unlike XSLT or XPathScript, an XSP document is not a separate stylesheet that is applied to an XML document to transform it; rather, XSP is concerned solely with *generating* content. Literal markup is mixed with functional code and elements from the XSP grammar within a single document to create an XML instance dynamically when that document is passed through the XSP processor. In AxKit's implementation, the embedded programming language is Perl.

As with AxKit's other Language modules, support for XSP must be explicitly enabled using the AxAddStyleMap configuration directive:

```
AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP
```

XSP diverges from the norm a bit in how documents are associated with the XSP processor. Unlike XSLT or XPathScript, XSP is self-contained, and there is no external stylesheet to apply. This means that when setting up the style processing directives for XSP, you use the literal string NULL, whereas if you use another Language module, you would typically put the path to the stylesheet.

```
# The familiar pattern, using XSLT
AxAddProcessor text/xsl /styles/somestyle.xsl

# In XSP there is no stylesheet, so the
# string NULL is used instead
AxAddProcessor application/x-xsp NULL

# The same, but as a conditional processor, based
# on the top-level element name and its namespace URI
AxAddRootProcessor application/x-xsp NULL {http://apache.org/xsp/core/v1}page
```

7.1.1 XSP Basics

XSP's basic syntactic requirements are very simple: every valid XSP document must be a well-formed XML document, and all XSP elements must be bound to the namespace URI http://apache.org/xsp/core/v1 (binding that URI to the prefix xsp is conventional, but certainly not required). In contrast to XPathScript, Active Server Pages, PHP, and similar technologies that use special pseudo tags (such as <% . . . %>, for example) as delimiters to separate code from literal markup, XSP uses an XML application grammar that separates literal and processed output through the use of specific XML elements bound to the XSP namespace. For example, the xsp:logic element creates a block that can contain any block of free-form Perl code, while the xsp:expr element interpolates a single expression into its literal result. The following shows a minimal, valid XSP document that inserts the current time into the content via an xsp:expr element:

```
<?xml version="1.0"?>
<application>
  <time>
     <xsp:expr xmlns:xsp="http://apache.org/xsp/core/v1">scalar localtime( )</xsp:expr>.
  </time>
</application>
```

Although XSP elements and code can be embedded into any XML document and passed to the XSP processor, an XSP page more commonly contains: a top-level xsp:page element containing optional xsp:structure and xsp:logic elements, and a required non-XSP element that becomes the top-level element of the result generated by the XSP processor (this is often called the *user-root* element). The following illustrates this basic structure:

```
<?xml version="1.0"?>

<!--  The  top-level <xsp:page> element, bound to the XSP namespace URI -->

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">



    <!--  optional <xsp:structure> used to import external Perl modules for use

in the current page-->

    <xsp:structure>

        <xsp:import>Some::Perl::Module</xsp:import>

    </xsp:structure>



    <!-- class-level logic (subroutines, global initialization, etc.) -->

    <xsp:logic>

        sub now {

            return scalar localtime( );

        }

    </xsp:logic>



    <!-- The following <application> element is the user-root

      that will become the top-level element after processing -->

    <application>

        <time>

            <xsp:expr>now( )</xsp:expr>

        </time>

    </application>

</xsp:page>
```

The following shows the result returned from both of the above examples:

```
<?xml version="1.0"?>

<application>

  <time>Thu Feb 12 20:26:11 2004</time>

</application>
```

Before we dig further into the details of XSP's syntax, it is important to note that the most effective use of XSP extends the Server Pages model beyond the code-mixed-with-HTML mess that may have driven you to look for an alternative environment such as AxKit in the first place. Yes, XSP allows and even encourages the mixture of XML markup and Perl code, but this is not the same as having code hardwired to generate *presentational* markup (which is arguably the true weakness found in most uses of the Server Pages technologies, not simply the fact that code and markup appear in the same document). To get the maximum benefit from XSP and AxKit, it is best to make sure that the markup that your XSP pages generate conforms to a grammar that best reflects the semantics and state of the application, irrespective of how that content may be presented. As long as your XSP-based applications meet this criteria, the syntactic details of *how* that data may be generated become solely a matter of personal taste.

For many web developers, the XML-influenced approach to generating dynamic content requires a little bit of mental adjustment. One tends to think of generating *pages* or *screens*, rather than constructing semantically meaningful data structures. It helps to remember, though, that the result generated by the XSP processor is really the *beginning* of the processing chain, not the end. In most cases, at least one transformative style will be applied to the XML created, and it is the stylesheet's job to transform the generated data into a usable interface. Therefore, you are free to generate simpler markup that reflects only the application's state and data structures.

Now that this conceptual foundation is laid, let's get to the business of introducing XSP. First, here is a summary of the grammar as a whole:

xsp:page

The optional top-level element for an XSP document. If present, it must be the top-level element of the document.

xsp:structure

A wrapper element for one or more xsp:import elements. May appear only as a child of a top-level xsp:page element and before the opening tag of the document's user-root element.

xsp:import

The text content of this element is expected to contain the name of a Perl class to be loaded during processing (similar to Perl's *use* function). This element may only appear as a child of the xsp:structure element.

xsp:logic

The content of this element is expected to be a free-form Perl block that is evaluated during processing.

xsp:expr

The content of this element is expected to be a Perl expression that is interpolated during processing. The result is added to the output as a literal string.

xsp:content

The content of this element is expected to be a well-formed XML fragment and is added, as is, to the resulting output. Although it may appear anywhere inside the user-root element, it most often appears as the child of an xsp:logic element, as a convenient way to escape out of the code to generate markup.

xsp:element

Creates an XML element whose name is defined by the required name attribute.

xsp:comment

The contents of this element will be wrapped in an XML comment in the resulting output.

xsp:pi

Will be expanded into an XML processing instruction, whose target is defined by the required target attribute.

XSP's grammar is quite small—just nine elements—but these elements, plus Perl's conditional statements, loops, and other logical constructs, can be combined to create very sophisticated XML content.

7.1.1.1 Generating content

First, it is important to keep in mind that all non-XSP elements, not otherwise skipped over based on conditional logic, are passed through the XSP processor verbatim into the generated result:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

<application>

    <user-info/>

</application>

</xsp:page>
```

Running this through the XSP processor gives the following predictable result:

```
<application>

    <user-info/>

</application>
```

Free-form Perl code can be embedded any place inside an XSP document using the xsp:logic element. At the same time, single elements or well-balanced chunks can be generated from within those logic blocks by wrapping the output in an xsp:content element. Also, the xsp:expr element can be used to interpolate a simple Perl scalar expression into a literal string:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

<application>

    <user-info>

    <xsp:logic>

        my $connection = $r->connection;

        my $ip = $connection->remote_ip;


        <xsp:content>

            <ip-address><xsp:expr>$ip</xsp:expr></ip-address>

        </xsp:content>


        if ( $ip =~ /^192\./ ) {

            <xsp:content>

              <is-local>true</is-local>

            </xsp:content>

        }

        else {

            <xsp:content>

              <is-local>false</is-local>

            </xsp:content>

        }
    </xsp:logic>

    </user-info>

</application>

</xsp:page>
```

In most cases, the XSP processor is smart enough to tell the difference between Perl code and literal markup that is meant to be part of the generated result. This makes the xsp:content element optional. Therefore, you could trim the previous example into the following and get the same result:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

<application>

    <user-info>

    <xsp:logic>

        my $connection = $r->connection;

        my $ip = $connection->remote_ip;


        <ip-address><xsp:expr>$ip</xsp:expr></ip-address>


        if ( $ip =~ /^192\./ ) {

              <is-local>true</is-local>

        }

        else {

              <is-local>false</is-local>

        }

    </xsp:logic>

    </user-info>

</application>

</xsp:page>
```

There are a few things to remember about xsp:logic elements. First, any logic block that appears *outside* of the user-root element is considered a special, global block that the XSP processor only interprets *once*. This global section is intended to provide a handy place to declare any subroutines or constant variables that your XSP page may need, while saving the overhead associated with reevaluating the code for every request. This provides a performance boost, but the behavior of the special, global case can sometimes snare the unwary. The following two XSP pages exemplify a common trap:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

    <!-- logic block outside of the user-root -->

    <xsp:logic>

        my $time = scalar localtime( );
    </xsp:logic>

<application>

    <time><xsp:expr>$time</xsp:expr></time>

</application>

</xsp:page>
```

The $time scalar variable is set in an xsp:logic block outside of the user-root application element—that is, in the special, "global" section. Compare that with this very similar example:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

<application>

    <!-- the same block, but inside the user-root -->

    <xsp:logic>

        my $time = scalar localtime( );

    </xsp:logic>

    <time><xsp:expr>$time</xsp:expr></time>

</application>

</xsp:page>
```

The two appear identical, but in the second example, the $time scalar is set in a logic block *within* the user-root element. You may expect that the two pages would return the same results, but remember that any logic block that appears outside of the user-root element is a special case that is only evaluated once. So the result returned from the first page shows the same time string no matter how many times the page is requested, while the second updates the time string for each request.

However, it is perfectly safe to put subroutines in the global block. In this case, it doesn't matter that the code is only evaluated once. The subroutine still *executes* at request time. So if you don't want to put the xsp:logic element inside the user-root, the following still works as you'd expect:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

    <!-- logic block outside of the user-root -->

    <xsp:logic>

        sub now { return scalar localtime( ); }

    </xsp:logic>

<application>

    <time><xsp:expr>now( )</xsp:expr></time>

</application>

</xsp:page>
```

The second thing to keep in mind when creating inline xsp:logic blocks concerns possible syntactic conflicts between Perl and XML. Certain common Perl expressions can cause XML well-formedness errors unless they are properly escaped. The specific characters you have to watch out for are «&» and «<».

```
<xsp:logic>

    # Parser sees '<' and thinks it's the beginning of a new XML element
    if ( $this_number < $that_number ) {  }

    my $here_doc = <<"TARGET";


    # Parser sees '&' and thinks it's the beginning of an XML entity reference

    if ( $this_condition && $that_condition ) {  }

    my $val = &sone_function( );

</logic>
```

The preferred solution here is to wrap the contents of the xsp:logic element that contain the conflicting characters in a CDATA (character data) section that the XML parser passes over without trying to parse it:

```
<xsp:logic><![CDATA[

   # You are now free to use & and < to your heart's content


]]><xsp:logic>
```

Using a CDATA section to tell the XML parser to skip the contents of the logic block also tells the parser to ignore any XSP elements or literal output, as well, so be sure to break out of the CDATA if you want to return content.

New XML elements may also be generated through use of the xsp:element element. The string passed to this element's required name attribute becomes the name of the newly generated element. The value set for the name attribute is usually a literal string, but you can also pass in a Perl scalar expression by wrapping it in curly braces. This is similar to XSLT's attribute value templates and offers the ability to produce elements whose names are generated programmatically at request time.

```
<params>

  <xsp:logic>

  my %query_params = $r->args;

  foreach my $key ( keys( %query_params )) {

      <xsp:element name="{$key}"><xsp:expr>$query_params{$key}</xsp:expr></xsp:element>

  }

  </xsp:logic>

</params>
```

Similarly, XML attributes can be generated via the xsp:attribute element. This element's required name attribute becomes the name of the newly created attribute, while the contents of this element become the attribute's value.

```
<params>

  <xsp:logic>

  my %query_params = $r->args;

  foreach my $key ( keys( %query_params )) {

      <param>

        <xsp:attribute name="name"><xsp:expr>$key</xsp:expr></xsp:attribute>

        <xsp:attribute name="value"><xsp:expr>$query_params{$key}</xsp:expr></xsp:attribute>
      </param>

  }

  </xsp:logic>

</params>
```

Finally, comments and processing instructions can be generated using the xsp:comment and xsp:pi elements, respectively. Generating comments can be especially useful when debugging your XSP while it is still in development. Rather than dumping information to the host's error log, you can simply produce a comment for nonfatal but unexpected errors, and the result appears directly in the output in a way that's easily distinguished, visibly, from the rest of the content:

```
<xsp:logic>

    if ($var eq 'this' ) {

        <this><xsp:expr>$var</xsp:expr></this>

     }

     else {

         <xsp:comment>Was expecting $var to be 'this' but got <xsp:expr>$var</xsp:expr
> instead.</xsp:comment>

     }

</xsp:logic>
```

7.1.1.2 Using Perl modules in XSP pages

Perl's popularity as a programming language has as much to do with the army of useful, open source extension modules freely available from the Comprehensive Perl Archive Network (CPAN) as it does with any particular feature of the language itself. Once installed (details about installing modules from CPAN can be found by running *perldoc CPAN* at a shell prompt on any machine on which Perl is installed), these modules can be used from within your XSP pages by adding an xsp:structure element as a direct child of the top-level xsp:page element, then adding an xsp:import element containing the name of the Perl package for each module that you want to use in your page:

```
<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">


  <xsp:structure>

                      <xsp:import>Geo::IP</xsp:import>

  </xsp:structure>


<application>

    <user-info>

    <xsp:logic>

        my $connection = $r->connection;

        my $ip = $connection->remote_ip;


                        my $mapper = Geo::IP->new( );
        <ip-address><xsp:expr>$ip</xsp:expr></ip-address>
        <country><xsp:expr>$mapper->country_name_by_addr($ip)</xsp:expr></country>
    </xsp:logic>
    </user-info>
</application>
</xsp:page>
```

You should now have a pretty good idea about XSP's page-level syntax and structure. There's more to the story, though. From what you have learned so far, you can generate application content quickly, but each page exists only unto itself, and the solutions are not reusable. Fortunately, AxKit's XSP implementation allows you to extend this basic framework to include your own custom application grammar extensions that can be reused in any number of pages to meet a variety of needs. These extensions are called *tag libraries*.

7.1.2 XSP Tag Libraries

XSP tag libraries (or taglibs, for short) provide a means to extend XSP's functionality by mapping XML elements in a specific, custom XML grammar to Perl code that expands and replaces those elements with other markup generated dynamically, performs a programmatic task behind the scenes, or most often, both. A tag library's elements are distinguished from literal output and elements in the XSP core grammar by binding those elements to a specific XML namespace. The following shows a simple XSP page that employs the Util tag library (available on CPAN) to return the current time:

```
<?xml version="1.0"?>
<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1"
       xmlns:util="http://apache.org/xsp/util/v1"

  <application>
      <time>
        <util:time/>
      </time>
  </application>
</xsp:page>
```

XSP taglibs can be implemented in one of two ways: as a special XSLT, XPathScript, or other stylesheet (called a *logicsheet*) used to preprocess the content, expanding the elements in the taglib grammar into the XSP elements and Perl codes that implement the taglib's behavior before the document is passed on the XSP processor, or more commonly, as a special Perl module registered with AxKit and used to extend the functionality of the XSP processor itself. You learn how to create both types of taglibs later in this chapter. For now, let's focus on some existing module-based taglibs and how they can be used to extend your XSP pages. The following explains just a few of the more popular or interesting XSP taglibs available for use in your XSP applications:

AxKit::XSP::Wiki

A complete Wiki application implemented in XSP. Uses POD as the page syntax.

AxKit::XSP::WebUtils

A general-purpose utility class that provides access to request and server data. Among other things, you can get or set HTTP headers, issue client redirects, examine whether the current request is using secure HTTP connections, and much more.

AxKit::XSP::Sendmail

Send email using this simple XSP interface to Perl's popular Mail::Sendmail module.

AxKit::XSP::ESQL

Feature-rich taglib for selecting data from any relational database supported by Perl's DBI.

AxKit::XSP::Util

Offers several utility functions for including content from local files, remote URI, and interpolated expressions, as well as the ability to get the current date and time in a variety of formats.

AxKit::XSP::LDAP

An easy-to-use XSP interface for retrieving data from LDAP servers.

AxKit::XSP::PerForm

Robust XSP helper class for creating and validating data entry applications.

AxKit::XSP::Swish

XSP interface to the popular site search/indexing tool, Swish-e.

There are many benefits of using XSP taglibs, but the most important is that they provide XSP developers with the ability to create generic, reusable libraries that implement common features or solve common problems.

7.1.2.1 Installing module-based XSP taglibs

Installing module-based XSP taglibs is exactly the same as installing any Perl module. If the taglib you want to install is available from CPAN, installation can be achieved quickly and painlessly using the CPAN shell that installs with Perl itself. Simply ask your systems administrator to install the taglib. (Be sure to provide her with the exact name of the taglib module you wish to install.) Or if you have permission, become root (superuser) and enter the following at a shell prompt:

```
perl -MCPAN -e shell

install Some::XSP::Taglib
```

Once the module is installed, you must first register the taglib with AxKit before you can start using it with your XSP applications. This is achieved by adding an `AxAddXSPTaglib` configuration directive to your *httpd.conf* or other Apache configuration file and passing the Perl package name of the taglib module as the sole argument. The following registers the Param and ESQL taglibs with AxKit:

```
AxAddXSPTaglib AxKit::XSP::Param

AxAddXSPTaglib AxKit::XSP::ESQL
```

Adding the taglib modules in this way does two things: it configures AxKit to load the module code (similar to Perl's built-in `use` statement), and it registers the unique XML namespace associated with that taglib grammar so that the XSP processor knows to dispatch the processing of elements in that namespace to the taglib's Perl package and not to simply pass them through as literal content.

Once a taglib module is registered, you may use the elements from its grammar in any XSP page. You need only to make sure to bind those elements to the XML namespace URI associated with that tag library; the XSP processor handles the rest.

The XSP taglib modules are already available from CPAN and cover an interesting set of cases. They are fantastic for adding reusable features to your own applications with a minimum of effort. However, there will surely be cases when what's out there does not meet your needs, and you should implement your own taglibs. Now that you have an idea about what XSP taglibs are and how they are used, let's examine how to create your own custom tag libraries.

7.1.2.2 Writing logicsheet taglibs

Module-based XSP taglibs (which we will examine shortly) are by far more common, but we would be remiss not to touch on the alternative, *logicsheet taglibs*, first. As I mentioned earlier, a logicsheet taglib is an XSLT, XPathScript, or other stylesheet applied to the source XSP document before it is handed to the XSP processor. During this transformation, elements in the tag library's grammar are expanded and replaced by the core XSP elements and Perl code that are required to implement the taglib's behavior.

Suppose that while designing our next XSP application, you discover that certain bits of environmental data are needed throughout each stage in the application. You decide to encapsulate access to that information in a tag library rather than duplicate the code in each XSP document. Specifically, you need to access the server's environment variable (%ENV), you need to know whether the current user has logged in (based on the presence of an HTTP cookie), and you want a list of links to simplify the creation of the ubiquitous "breadcrumb" navigation bar. Given these requirements, you could define your taglib grammar in the following way:

> The XML namespace URI for this grammar is http://localhost/xsp/myapp, and the preferred namespace prefix is myapp.

myapp:env

> This element is replaced by an env element containing a list of field elements, each containing name and value elements that reflect the name and value of each field in the %ENV hash.

myapp:breadcrumb

> This element is replaced by a breadcrumb element that contains an ordered list of link elements. Each link element represents a link in the breadcrumb chain and contains both an href attribute containing the URI of that step and a title attribute that may be used when creating a hyperlink for that step.

myapp:logincheck

> This element is replaced by a logged-in element whose text contents are either true or false depending on the presence or absence of an HTTP cookie. If the value of the logged-in element is true, a username element containing the current user's name will also appear.

An XSP page using this taglib grammar may look something like Example 7-1.

Example 7-1. minimal_myapp.xsp

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1"

          xmlns:myapp="http://localhost/myapp">

<xsp:structure>

   <xsp:import>Apache::Cookie</xsp:import>

</xsp:structure>

<application>

  <meta>

    <myapp:breadcrumb/>

    <myapp:env/>

    <user>

      <myapp:logincheck/>
    </user>

  </meta>

</application>

</xsp:page>
```

Now you must create the logicsheet that expands the elements in the MyApp grammar into the XSP elements and Perl code that actually make the taglib work as you expect. Example 7-2 shows one possible way to do this, using XSLT to implement the logicsheet.

Example 7-2. myapp.xsl : the MyApp taglib as a preprocessed XSLT logicsheet

```
<?xml version="1.0"?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsp="http://apache.org/xsp/core/v1"
  xmlns:myapp="http://localhost/xsp/myapp"
>

<xsl:template match="myapp:env">
<env>
  <xsp:logic>
    foreach my $key (keys(%ENV)) {
        <field>
          <name><xsp:expr>$key</xsp:expr></name>
          <value><xsp:expr>$ENV{$key}</xsp:expr></value>
        </field>
    }
  </xsp:logic>
</env>
</xsl:template>

<xsl:template match="myapp:breadcrumb">
  <breadcrumb>
  <xsp:logic>
    my $path ='/';

    <link title="home" href="/"/>

    my @steps = split '/', $r->uri;
    for ( my $i = 0; @steps > $i; $i++ ) {
        my $step = $steps[$i];
        next unless length $step;
        $path .= $step;
        $path .= '/' unless $i =  = $#steps;

        <link>
          <xsp:attribute name="title">
             <xsp:expr>$step</xsp:expr>
          </xsp:attribute>

          <xsp:attribute name="href">
              <xsp:expr>$path</xsp:expr>
          </xsp:attribute>
       </link>
    }
  </xsp:logic>
  </breadcrumb>
</xsl:template>

<xsl:template match="myapp:logincheck">
<xsp:logic>
    my $cookie = Apache::Cookie->fetch;
    if ( defined( $cookie->{'username'} ))  {
        my $username = $cookie->{'username'}->value;
        <logged-in>true</logged-in>
        <username><xsp:expr>$username</xsp:expr></username>
    }
    else {
        <logged-in>false</logged-in>
    }
</xsp:logic>
</xsl:template>

<xsl:template match="*">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates />
  </xsl:copy>
</xsl:template>
```

This logicsheet taglib is really just a plain XSLT stylesheet that replaces the elements in the MyApp grammar with the code, XSP elements and literal markup required to implement the taglib. Obviously, for this to work, you must apply the *myapp.xsl* stylesheet to the source document:

```
<FIles minimal_myapp.xsp>

  AxAddProcessor text/xsl /styles/myapp.xsl

</Files>
```

With this snippet added to your configuration file, a request for the *minimal_myapp.xsp* document gives the following result:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1" xmlns:myapp="http://localhost/myapp">

<xsp:structure>

   <xsp:import>Apache::Cookie</xsp:import>

</xsp:structure>

<application>

  <meta>

    <breadcrumb><xsp:logic>

    my $path ='/';


    <link title="home" href="/"/>


    my @steps = split '/', $r->uri;

    for ( my $i = 0; @steps > $i; $i++ ) {

        my $step = $steps[$i];

        next unless length $step;

        $path .= $step;

        $path .= '/' unless $i =  = $#steps;


        <link>

            <xsp:attribute name="title">

                <xsp:expr>$step</xsp:expr>

            </xsp:attribute>

            <xsp:attribute name="href">

                <xsp:expr>$path</xsp:expr>

            </xsp:attribute>

        </link>


    }
  </xsp:logic></breadcrumb>
    <env><xsp:logic>
    foreach my $key (keys(%ENV)) {
        <field>
            <name><xsp:expr>$key</xsp:expr></name>
            <value><xsp:expr>$ENV{$key}</xsp:expr></value>
        </field>
    }
  </xsp:logic></env>
    <user>
      <xsp:logic>

    my $cookie = Apache::Cookie->fetch;
    if ( defined( $cookie->{'myapp_username'} ))  {
        my $username = $cookie->{'myapp_username'}->value;
        <logged-in>true</logged-in>
        <username><xsp:expr>$username</xsp:expr></username>
    }
    else {
        <logged-in>false</logged-in>
    }
</xsp:logic>
    </user>
  </meta>
</application>
</xsp:page>
```

As expected, the elements from the MyApp grammar are replaced and all other content is copied through verbatim. You only need to configure AxKit to apply the XSP processor to this intermediate step to get the final result:

```
<Files minimal_myapp.xsp>
  # Preprocess using the XSLT logicsheet
  AxAddProcessor text/xsl /styles/myapp.xsl

  # And send the result to the XSP processor
  AxAddProcessor application/x-xsp NULL
</Files>
```

This simple two-step processing chain delivers the final, desired result:

```
<?xml version="1.0" encoding="UTF-8"?>
<application>
    <meta>
        <breadcrumb>
          <link title="home" href="/"/>
          <link title="axkitbook" href="/axkitbook/"/>
          <link title="samples" href="/axkitbook/samples/"/>
          <link title="chapt07" href="/axkitbook/samples/chapt07/"/>
          <link title="logicsheet.xml" href="/axkitbook/samples/chapt07/minimal_myapp"/>
        </breadcrumb>
        <env>
          <field>
           <name>PATH_INFO</name>
           <value/>
         </field>
         <field>
           <name>PATH</name>
           <value>/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/kip/bin</value>
        </field>
        <field>
           <name>GATEWAY_INTERFACE</name>
           <value>CGI-Perl/1.1</value>
        </field>
       <field>
          <name>MOD_PERL</name>
          <value>mod_perl/1.29</value>
       </field>
    </env>
    <user>
      <logged-in>true</logged-in>
      <username>ubu</username>
    </user>
  </meta>
</application>
```

If this were a real-world XSP page, you would certainly include more that just the bits of metadata that you have here. Also, you would probably have a content or state element that contains the data associated with the current state of whatever application you were implementing, but this example shows enough of the basics to get you up and running.

7.1.2.3 Writing module-based taglibs using TaglibHelper

Logicsheet-based taglibs such as the one you just created can be handy, but in practice, most XSP tag libraries are implemented as Perl modules that are then registered with the XSP processor. There are several benefits to using module-based taglibs rather than logicsheets. First, module-based taglibs do not require the XSP source to be preprocessed before execution; this not only saves the overhead associated with adding an additional transformation, but it often greatly simplifies the style processor configuration needed to serve the XSP pages throughout a given site. Also, module-based taglibs offer additional performance, since the Perl code that implements the taglib functions will be cached in memory, so the XSP processor is able to dispatch the handling of the taglib elements to the external module as quickly as it processes the elements in the core XSP grammar.

While it is possible to write taglib modules that interact directly with the low-level components of AxKit's XSP engine, this approach is repetitive and error-prone, and requires intimate knowledge of the XSP processor's internals. It is usually better to use one of the helper modules available to streamline and simplify the process of writing custom tag libraries. In Example 7-3, the MyApp grammar from the logicsheet sample is reimplemented as a Perl module using Steve Willer's Apache::AxKit::XSP::TaglibHelper module that ships with the core AxKit distribution.

Example 7-3. MyApp.pm

```
package TaglibHelper::MyApp;


use strict;

use Apache::AxKit::Language::XSP::TaglibHelper;

use Apache::Cookie;


use vars qw( @ISA $NS @EXPORT_TAGLIB );

@ISA = qw( Apache::AxKit::Language::XSP::TaglibHelper );

$NS = 'http://localhost/xsp/myapp';


@EXPORT_TAGLIB = (

    'logincheck( )',

    'env( ):listtag=env:itemtag=field',

    'breadcrumb( ):as_xml=1',

);


sub logincheck {

    my $cookie = Apache::Cookie->fetch;

    my $out = {  };


    if ( defined( $cookie->{'username'} ))  {

        my $username = $cookie->{'username'}->value;

        $out->{'logged-in'} = 'true';

        $out->{username} = $username;

    }

    else {

        $out->{'logged-in'} = 'false';
    }
    return $out;
}

sub env {
    my @out = ( );
    foreach ( keys( %ENV ) ) {
        push @out, { name => $_, value => $ENV{$_}};
    }
    return \@out;
}

sub breadcrumb {
    my $out = '<breadcrumb>';
    $out .=   '<link title="home" href="/"/>';

    my $path ='/';
    my $r = AxKit::Apache->request( );

    my @steps = split '/', $r->uri;
    for ( my $i = 0; @steps > $i; $i++ ) {
        my $step = $steps[$i];
        next unless length $step;
        $out .= qq|<link title="$step" href="$path"/>|;
    }

    $out .= '</breadcrumb>';
    return $out;
}
1;
```

There are several things to note here. First, the module is a subclass of the TaglibHelper class itself. This allows taglib module authors to implement only the functions required to react to the tags in that specific taglib while hiding the tedious low-level processing. Next, TaglibHelper works by allowing subclass authors to write simple Perl subroutines whose names match the local name of the elements in the taglib's grammar. Each time a given element from the taglib being implemented is encountered by the XSP processor, the matching subroutine is called.

The arguments passed to the module's subroutines and the way the values returned from those subroutines will be processed are determined by the function specifications passed to the `@EXPORT_TAGLIB` array. Each element in this array takes the form of a string that follows the pattern illustrated here:

```
tagname([argument specification])[: additional options ]
```

While `tagname` is the local name (unprefixed) of the taglib element to match, `argument specifications` is an optional comma-separated list of any child elements of the taglib element that should be passed as arguments to the taglib subroutine, and additional options provide additional information about how the data returned from the taglib subroutine is processed and included in the result of the XSP process. Look back at the @EXPORT_TAGLIB array for your MyApp taglib module:

```
@EXPORT_TAGLIB = (
    'logincheck( )',
    'env( ):listtag=env:itemtag=field',
    'breadcrumb( ):as_xml=1',
);
```

This tells the TaglibHelper parent class to look for three elements from the taglib's namespace, logincheck, env, and breadcrumb. It further indicates that the *logincheck* subroutine expects no arguments (note the empty argument specification) and that the subroutine returns a simple Perl data structure that should be serialized to XML. The specification for the env subroutine shows that it too expects no arguments, that it will return a list reference whose entries should be named field (itemtag=field), and that the entire list should be wrapped in an env element (listtag=env). Finally, the specification for the breadcrumb indicates that it also expects no arguments and that the result returned is a well-balanced chunk of XML that should be parsed and included in the final result (as_xml=1).

With this module installed, you only need to register it by name via the AxAddXSPTaglib directive. You can begin using the tags from the MyApp grammar in your XSP pages without having to preprocess them using the logicsheet you created earlier.

```
# Load and register the taglib module
AxAddXSPTaglib TaglibHelper::MyApp

<Files minimal_myapp.xsp>
  # No need to preprocess, just send the source directly to the XSP processor
  AxAddProcessor application/x-xsp NULL
</Files>
```

The result returned from a request to *minimal_myapp.xsp* using your new taglib module is exactly the same as the result returned for the previous logicsheet taglib, so I will not duplicate the output here.

In the same way that Perl's many extension modules increase the power and value of the language as a whole by providing out-of-the-box solutions for common tasks, the judicious use of XSP tag libraries can add significant value to your XSP applications by reducing common application features to a handful of editor-friendly XML elements. If you are serious about building XSP applications with AxKit, I strongly suggest spending the time to learn the ins and outs of the TaglibHelper class or one of the other similar classes (SimpleTaglib or ObjectTaglib) that seek to streamline the process of writing custom XSP tag libraries—the time spent will more than pay for itself in the long run.

7.1.3 XSP Debugging Tips

When you are first starting out with XSP, it can be hard to track down precisely where something went wrong with your page or taglib code. The reason is that, to make XSP pages fast, flexible, and cache-friendly, AxKit's XSP engine is quite complex. (It's actually a SAX Handler that dispatches events to the core event handlers and various taglib modules while dynamically constructing an intermediate Perl class that uses the DOM interface to generate the results!) A lot of advanced Perl magic happens behind the scenes. There are, however, several AxKit configuration directives that you can use to make debugging your XSP applications more straightforward.

First, be sure to set the AxDebugLevel directive to maximum (10). This dumps copious amounts of information about what AxKit is doing behind the scenes to your web host's error log. Next, be sure that the AxStackTrace directive is set to On. This causes the AxKit error handling mechanism to produce a deep stack trace in the case of a fatal exception.

Also, be sure to set up the AxErrorStylesheet directive properly. As with the style processing directives, this option accepts the MIME type associated with one of the Language processing modules and the path to a stylesheet that will be applied to the XML error document that AxKit produces if this directive is present. For example, the following configures AxKit to apply the */styles/axkit_error.xps* stylesheet to the generated XML stack trace in the case of a fatal processing error:

```
AxErrorStylesheet application/x-xpathscript /styles/axkit_error.xps
```

This sends the transformed version of the Perl error and stack trace to the requesting client instead of the disappointing default 500 Internal Server Error that Apache usually offers. This saves you development time by obviating the need to look at the error log every time something blows up. See the entry about the AxErrorStylesheet directive in Appendix A for details about the XML that AxKit produces in these cases.

Finally, AxKit offers the helpful AxTraceIntermediate directive explicitly to help debug XSP pages and complex processing chains. This directive takes the path to a directory as its sole argument. Then, when a document is requested, AxKit generates a series of files in that directory (with the file extensions 0-n)—one for each transformation in the processing chain. If the chain includes an XSP process, a special *.XSP* file is generated that shows the intermediate Perl class that AxKit generates to implement the XSP page. This invaluable tool can save hours of groping for subtle bugs in your XSP code. To ensure that the results are easy to read, combine this directive with the AxDebugTidy On directive to format the intermediate files for better readability.

A word of caution, though—with the exception of the AxErrorStylesheet directive (which is good to use in any case), each directive listed here has a negative impact on AxKit's performance and should, therefore, only be used in a development environment, not on a production server.

7.2 Other Dynamic XML Techniques

While XSP is the dominant dynamic XML tool, there are other options, which may be appropriate in some cases.

7.2.1 Aggregate Data URIs

An *aggregate data URI* is simply a way to combine content from more than one source and present it as a single URI-addressable resource. While the uses for this technique are many, the most obvious and intuitive use is building collections of metadata about all or some documents on a given site. Suppose you are publishing an online newsletter. In each publishing period, you publish a few articles while archiving the old ones. Obviously, one feature that such a site needs is an article index—you do not want the older content to be forgotten and rendered useless just because it is not featured in this period's issue. Furthermore, you know from experience that such indexes are more useful to your visitors when they contain more than just a list of hyperlinked titles. They need a greater level of detail (date published, article summary, etc.) about how the archived articles are presented in the index. When faced with this task, developers traditionally go one of two ways: they either choose to create and maintain the index by hand, or they store the archived articles in a database and use a script to generate the index dynamically. Using XML, AxKit, and aggregate URIs offers a novel, third choice: you can set up alternate styles for the articles themselves to automatically extract the metadata you need, then combine those article-specific bits of data into a single resource that is the content source for the online index.

The advantage here is that filtered aggregated results are cached on disk after the first request—saving the overhead of reading, parsing, and transforming each individual document.

Example 7-4 shows the stylesheet that extracts the metadata from the articles. To keep things simple, assume that all the articles are marked up using the Simplified DocBook grammar.

Example 7-4. sdocbook_meta.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    version="1.0"
>
<xsl:param name="request.uri.hostname"/>
<xsl:param name="request.uri"/>
<xsl:variable name="fullURI" select="concat('http://', $request.uri.hostname, $$

<xsl:template match="article">
<rdf:RDF>
  <rdf:Description rdf:about="{$fullURI}">
    <dc:format>text/xml</dc:format>
    <dc:title><xsl:value-of select="title|articleinfo/title"/></dc:title>
    <xsl:apply-templates select="articleinfo"/>
  </rdf:Description>
</rdf:RDF>
</xsl:template>

<xsl:template match="publishername">
  <dc:publisher><xsl:value-of select="."/></dc:publisher>
</xsl:template>

<xsl:template match="authorgroup">
<dc:creator>
  <rdf:Bag>
    <xsl:for-each select="author">
      <rdf:li>
        <xsl:value-of select="concat(firstname, ' ', surname)"/>
      </rdf:li>
    </xsl:for-each>
  </rdf:Bag>
</dc:creator>
</xsl:template>

<xsl:template match="author">
  <dc:creator>
    <xsl:value-of select="concat(firstname, ' ', surname)"/>
  </dc:creator>
</xsl:template>

<xsl:template match="articleinfo">
  <xsl:if test="copyright|legalinfo">
    <dc:rights>
      <xsl:if test="copyright">
        Copyright <xsl:apply-templates select="copyright"/>
        <xsl:value-of select="legalinfo"/>
      </xsl:if>
    </dc:rights>
  </xsl:if>
  <xsl:variable name="subjects"
       select="keywordset/keyword|subjectset/subject/subjectterm"/>
   <xsl:if test="count($subjects)">
```

```
    <dc:subject>
      <xsl:choose>
        <xsl:when test="count($subjects) > 1 ">
          <rdf:Bag>
            <xsl:for-each select="$subjects">
              <rdf:li><xsl:value-of select="."/></rdf:li>
            </xsl:for-each>
          </rdf:Bag>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="$subjects"/>
        </xsl:otherwise>
      </xsl:choose>
    </dc:subject>
  </xsl:if>
  <xsl:apply-templates select="./*[local-name( ) != 'copyright']"/>
</xsl:template>

<xsl:template match="copyright">
  <xsl:value-of select="year"/>
  <xsl:text> </xsl:text>
  <xsl:value-of select="holder"/>
</xsl:template>

<xsl:template match="abstract">
  <dc:description><xsl:value-of select="."/></dc:description>
</xsl:template>

<xsl:template match="pubdate">
  <dc:date><xsl:value-of select="."/></dc:date>
</xsl:template>

</xsl:stylesheet>
```

This simple stylesheet extracts the title, author's name, publisher, date, abstract summary, and several other useful bits of information from a Simplified DocBook article. It presents the information as an RDF/XML document using elements from the Dublin Core Metadata Initiative's suggested grammar. This a nice feature that will probably prove useful beyond your immediate goal, but you are not done yet. You need a way to pull the metadata for each article into a single resource. You can achieve this in several ways. Example 7-5 shows a simple RDF/XML document that lists the articles you want to include into the metadata aggregate.

Example 7-5. article_list.rdf

```
<?xml version="1.0"?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

 <rdf:Seq rdf:about="http://localhost/articles/"

      xml:base="http://localhost/articles/">

   <rdf:li rdf:resource="smithee.dkb" />

   <rdf:li rdf:resource="goldenage.dkb" />

   <rdf:li rdf:resource="nonlinear.dkb" />

 </rdf:Seq>

</rdf:RDF>
```

Note the use of the xml:base attribute. This not only saves you from having to type out the full URL for each article, but it provides the base URI when resolving the included documents. Example 7-6 shows the XSLT stylesheet that handles the task of combining the individual metadata results in one document.

Example 7-6. sdocbook_meta_aggregate.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet

     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

     xmlns:dc="http://purl.org/dc/elements/1.1/"

     version="1.0"

>


<xsl:template match="/">

<rdf:RDF>

  <xsl:apply-templates/>

</rdf:RDF>

</xsl:template>


<xsl:template match="rdf:li">

  <xsl:variable name="uri" select="concat(@rdf:resource, '?style=meta')"/>

  <xsl:copy-of select="document($uri, .)/rdf:RDF/*"/>

</xsl:template>


</xsl:stylesheet>
```

By using the two-argument form of the *document( )* function, the XSLT processor uses the base URI that you set using xml:base attribute in the source document to resolve the document's location. In this case, that means that the XSLT processor will make an HTTP request back to your web host for each article, specifying the meta style in the query string, which, in turn, will cause AxKit to apply the *sdocbook_meta.xsl* stylesheet to that document when returning the result. The sum of all of this is that you can now access all metadata from all the articles listed in the *article_files.rdf* file via a single URI. The result looks something like this:

```
<?xml version="1.0"?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

           xmlns:dc="http://purl.org/dc/elements/1.1/">

  <rdf:Description rdf:about="http://myhost.tld/articles/smithee.dkb">

    <dc:format>text/xml</dc:format>

    <dc:title>The History of Alan Smithee</dc:title>

    <dc:rights> Copyright 2001 John Q. Pundette, Esq.</dc:rights>

    <dc:subject>

      <rdf:Bag>

        <rdf:li>History of Cinema</rdf:li>

        <rdf:li>Film Directors</rdf:li>

      </rdf:Bag>

    </dc:subject>

    <dc:creator>John Pundette</dc:creator>

    <dc:date>2001</dc:date>

    <dc:description> Alan Smithee is credited with directing some of the worst films of all

      time-- and that's just the way he likes it. Find out more about this enigmatic and

      controversial figure. </dc:description>

  </rdf:Description>

  <rdf:Description rdf:about="http://myhost.tld/articles/goldenage.dkb">

    <dc:format>text/xml</dc:format>

    <dc:title/>

    <dc:rights> Copyright 2001 John Q. Pundette, Esq.</dc:rights>

    <dc:creator>John Pundette</dc:creator>

    <dc:date>2001</dc:date>

    <dc:description> Hollywood is back with a vengeance, and this time, it's

personal. </dc:description>

  </rdf:Description>

  <rdf:Description rdf:about="http://myhost.tld/articles/nonlinear.dkb">

    <dc:format>text/xml</dc:format>

    <dc:title>Some Assembly Required: Why We Need More Non-Linear Plots</dc:title>

    <dc:rights> Copyright 2004 Indy Filmsnob</dc:rights>

    <dc:creator>Indiana Filmsnob</dc:creator>

    <dc:date>2004</dc:date>

    <dc:description> Modern viewers have had enough of the Hansel and Gretel

School of plot

      development. Let's mix it up, people! </dc:description>
    </rdf:Description>

</rdf:RDF>
```

With this result, you have all the data you need to create a rich, meaningful index for all articles on your site; you only need to process this document with another stylesheet to generate browsable HTML. It does not stop there, though. For example, you may also reuse that same aggregate source to extract all information about articles written by a single author, or ones published during a specific date range. All you need to do is apply the appropriate stylesheets to the aggregate source to get what you need. The following configuration snippet ties together what you have done so far and shows some possibilities for filtering the aggregated metadata:

```
# These rules apply to the /articles/ directory

<Directory /articles/>


    # Add the query string StyleChooser

    AxAddPlugin Apache::AxKit::StyleChooser::QueryString


    # Transform sdocbook into HTML

    <AxStyleName html>

        AxAddProcessor text/xsl /styles/sdocbook_html.xsl

    </AxStyleName>


    # Extract the article's metadata as RDF

    <AxStyleName meta>

         AxAddProcessor text/xsl /styles/sdocbook_meta.xsl

    </AxStyleName>


    # HTML is the default style

    AxStyle html


    # Special rules apply to the metadata aggregate list

    <Files article_list.rdf>

        # The other styles never apply here

        AxResetProcessors

        AxAddProcessor text/xsl /styles/sdocbook_meta_aggregate.xsl


        # Show the aggregated data as an HTML index

        <AxStyleName html>

            AxAddProcessor text/xsl /styles/rdf_aggregate2html.xsl

        </AxStyleName>


        # Reuse the aggregated data for inclusions that filter the data

        # based on params passed that specify author, date range, etc.
        <AxStyleName filtered>
            AxAddProcessor text/xsl /styles/rdf_aggregate_filter.xsl
        </AxStyleName>

        # HTML is the default
        AxStyle html
    </Files>

</Directory>
```

This strategy still requires someone to manually update the *article_list.rdf* file as new articles are added. If you want to make the whole thing more or less maintenance free, you could use an XSP page to create the list of articles. (See Example 7-7.)

Example 7-7. article_list.xsp

```
<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1"
          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<xsp:structure>
  <xsp:import>File::Find::Rule::XPath</xsp:import>
</xsp:structure>

<xsp:logic>
    my $web_base = 'http://localhost/';
    my $finder = File::Find::Rule::XPath->new( );
    $finder->name('*.xml', '*.dkb');
    $finder->relative(1);
    $finder->xpath('/article');

    my @files = $finder->in( $r->document_root . '/articles/');
</xsp:logic>
<rdf:RDF>
 <rdf:Seq rdf:about="http://localhost/articles/"
      xml:base="http://localhost/articles/">
  <xsp:logic>
   foreach my $path ( @files ) {
      <rdf:li rdf:resource="{$path}"/>
    }
   </xsp:logic>
  </rdf:Seq>
</rdf:RDF>
</xsp:page>
```

This deceptively simple XSP page uses Grant McLean's File::Find::Rule::XPath module from CPAN to search the host's */articles/* directory (and its subdirectories) for all files with a *.dbk* or *.xml* extension and those containing a top-level article element. It generates a document with the same structure as the previous static file, except the list is created dynamically.

There is a trade-off here, though. Generating the list dynamically frees you from updating the list of articles, but it also means that all the aggregation processing happens *for every request*. With the static version, the result is cached on the first request, and the process is not repeated until the list of files changes.

To be sure, most aggregate data URI are not as complex as the one you created here. In fact, most of the time, a few simple XInclude elements do the trick. However, combining transformed resources into one resource that is also transformed to meet a specific need illustrates a powerful technique that would be difficult, if not impossible, to achieve outside a dynamic flexible XML environment such as AxKit.

7.2.2 Application ContentProviders

XSP offers a rich environment in which to generate dynamic XML content, but it is not to everyone's taste. For some, using markup application grammar and tag libraries, which map XML elements to Perl code that generates markup during processing, adds an undesirable layer of indirection to their day-to-day coding; others simply prefer to generate content within a familiar, pure Perl environment. As usual, AxKit's modularity offers an alternative. If XSP is not your style, you may consider creating or using an *application ContentProvider* instead.

In the most generic sense, an application ContentProvider is nothing more that an alternative Provider class that generates its content dynamically, based on one or more conditions (in contrast to the default Provider that reads the content from a file on the disk). More often, though, an application Provider is a bridge between an existing application environment and AxKit.

To illustrate this utility and the simplicity that using an application ContentProvider can offer, we will examine a simple two-state web application using the SAWA (Simple API for Web Applications) application environment. We will not delve into SAWA in detail here; just know that it implements the Model, View, Controller design pattern on top of a flexible dispatching mechanism that fires a series of registered event handler methods (across multiple component classes, if need be) in response to the current state of the application. The typical SAWA application consists of two parts: a base application class in which the processing logic for the current application state takes place, and an output class in which the concrete interface that represents state is generated. SAWA has the added benefit of an output class that is also an AxKit Provider. This gives SAWA both a way to pass content directly into AxKit as well as a means to control which transformative styles AxKit applies to the content returned.

To show how SAWA and AxKit can work together, we will create a simple application that accepts a bit of user input and then generate a random pairing for the data entered—essentially, a variation of the typical "Find Your (Gangster/Past Life/Hip-Hop DJ) Name" forms that pop up regularly on the Web. Example 7-8 shows the base application class.

Example 7-8. MovieName::Base.pm

```
package MovieName::Base;
use SAWA::Constants;
use SAWA::Event::Simple;
use strict;


our @ISA = qw( SAWA::Event::Simple );


BEGIN { srand(time( ) ^ ($$ + ($$ << 15))) }
```

```
sub registerEvents {
    return qw/ complete /;
}

sub event_default {
     my $self    = shift;
     my $ctxt   = shift;
     $ctxt->{style_name} = 'moviename_prompt';
     return OK;
 }

sub event_complete {
    my $self    = shift;
    my $ctxt    = shift;

    my @param_names = $self->query->param;

    my @first_names = map { $self->query->param("$_") } grep { $_ =~ /^first\./ }
@param_names;
    my @last_names  = map { $self->query->param("$_") } grep { $_ =~ /^last\./ }
@param_names;

    if ( ((scalar @first_names) + (scalar @last_names) != 6)
       || ( grep { length =  = 0 } @first_names, @last_names ) ) {

        $ctxt->{message} = 'All fields must be filled in. Please try again.';
        $ctxt->{style_name} = 'moviename_prompt';
        return OK;
    }

    $ctxt->{first_name} = $first_names[ int( rand( @first_names )) ];
    $ctxt->{last_name} = $last_names[ int( rand( @last_names )) ];
    $ctxt->{message} = 'Congratulations! Your Movie Star name has been magically determined!';
    $ctxt->{style_name} = 'moviename_complete';
    return OK;
}

1;
```

Two event-handler methods are presented here. The first is the *event_default( )* handler, which is called if no registered event is called. It simply sets the value for the style_name key in the global context hash reference (that is passed to every handler method) to the value stylea_prompt. (You'll see the effect this has when we examine the output class.) Next is the .*event_complete( )* that is called when the client submitted data. In this case, after a little error checking, the handler also sets the style_name key and adds the randomly chosen first and last names to the context hash. Example 7-9 shows the output class in which things get a bit more interesting.

Example 7-9. MovieName::Output.pm

```
package MovieName::Output;

use XML::LibXML;

use SAWA::Constants;

use SAWA::Output::XML::AxKit;

our @ISA = qw/ SAWA::Output::XML::AxKit /;


sub get_style_name {

    my $self = shift;

    my $ctxt = shift;

    $self->style_name( $ctxt->{style_name}  );

    return OK;

}


sub get_document {

    my $self = shift;

    my $ctxt = shift;

    my $dom  = XML::LibXML::Document->new( );

    my $root = $dom->createElement( 'application' );

    $dom->setDocumentElement( $root );


    my $msg_element = $dom->createElement( 'message' );

    $msg_element->appendChild( $dom->createTextNode( $ctxt->{message} ) );


    my $fname_element = $dom->createElement( 'first_name' );

    $fname_element->appendChild( $dom->createTextNode( $ctxt->{first_name} ) );


    my $lname_element = $dom->createElement( 'last_name' );

    $lname_element->appendChild( $dom->createTextNode( $ctxt->{last_name} ) );


    $root->appendChild( $fname_element );

    $root->appendChild( $lname_element );
    $root->appendChild( $msg_element );


    $self->document( $dom );

    return OK;

}


1;
```

First, this class is a subclass of SAWA::Output::XML::AxKit, which together with Apache::AxKit::Provider::SAWA, provides the bridge between SAWA and AxKit. In practical terms, this means that the data you pass to class member accessors from these output methods are forwarded directly to AxKit. In the case of the *get_style_name( )* event handler, you set the class member style_name to the value selected by the event hander in the Base.pm application class. The name selected (moviename_prompt or moviename_complete) corresponds to the AxKit-named style blocks whose processing directives are used to transform the content returned.

The *get_document( )* event handler simply generates an XML::LibXML::Document instance by which you create the XML (via the DOM interface) that is passed off AxKit to transform. (This SAWA class also allows you to return the content as textual XML markup, but then AxKit has to parse it before transformation. Returning a document object saves a little bit of parsing overhead).

Now you have seen how the style is selected and the content created. Let's look at the configuration that enables this to work seamlessly with AxKit:

```
# Load the AxKit<->SAWA "bridge"

PerlModule Apache::AxKit::Provider::SAWA


<Location /moviename>

    # Set AxKit as the handler for this virtual URI

    SetHandler axkit

    AxResetProcessors


    # Add the request params plug-in so XSLT can access request data

    AxAddPlugin Apache::AxKit::Plugin::AddXSLParams::Request

    PerlSetVar AxAddXSLParamGroups "Request-Common"


    # Set the "bridge" as the AxKit ContentProvider for this resource

    AxContentProvider Apache::AxKit::Provider::SAWA


    # Add out two SAWA modules to its dispatcher pipeline

    SawaAddPipe MovieName::Base

    SawaAddPipe MovieName::Output


    # Set up the AxKit  named styles that your SAWA application will select to

    # create the View for the current application state.

    <AxStyleName moviename_prompt>

        AxAddProcessor text/xsl /styles/moviename/prompt.xsl
        AxAddProcessor text/xsl /styles/moviename/common.xsl

    </AxStyleName>


    <AxStyleName moviename_complete>

        AxAddProcessor text/xsl /styles/moviename/complete.xsl

        AxAddProcessor text/xsl /styles/moviename/common.xsl

    </AxStyleName>


</Location>
```

This configuration block loads the Apache::AxKit::Provider::SAWA class as a PerlModule (required, since it implements its own set of custom Apache configuration directives). It then creates a virtual URI, /moviename, which uses that module as the AxKit ContentProvider. Next, the SAWA application and output classes are added to SAWA's pipeline, and the AxKit-named style blocks that the application will select to transform the content are added.

Your little application has two states: the default state (moviename_prompt), which prompts the users to enter the data that will be used to randomly generate their movie star name, and the complete state (moviename_complete), in which the generated result is presented. SAWA determines the application state, then passes the name of the style that reflects that state to AxKit, which performs the actual content transformation.

Let's take a quick look at the stylesheets themselves before moving on. First, Example 7-10 shows the style associated with the default, prompt state.

Example 7-10. prompt.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:import href="params.xsl"/>


<xsl:template match="/">

  <xsl:apply-templates/>

</xsl:template>


<xsl:template match="application">

  <application>

  <xsl:apply-templates/>

  <body>

    <form name="prompt" action="{$request.uri}" method="post">

      <div class="appmain">

        <div class="row">

          <span class="label">Name of classy hotel:</span>

          <span class="inputw">

              <input name="last.hotel" type="text" value="{$last.hotel}"/>

          </span>
```

```
</div>

<div class="row">

  <span class="label">

   Name a street you lived on when you were a teenager:</span>

  <span class="inputw">

      <input name="last.street" type="text" value="{$last.street}"/>

  </span>

</div>

<div class="row">

  <span class="label">Your mother's maiden name:</span>

  <span class="inputw">

      <input name="last.maiden" type="text" value="{$last.maiden}"/>

  </span>

</div>

<div class="row">

  <span class="label">Your middle name:</span>

  <span class="inputw">

      <input name="first.middle_name" type="text" value="{$first.middle_name}"/>

  </span>

</div>

<div class="row">

  <span class="label">The name of your favorite pet:</span>

  <span class="inputw">

      <input name="first.pet" type="text" value="{$first.pet}"/>

  </span>

</div>

<div class="row">

  <span class="label">Name of your favorite model of car:</span>

  <span class="inputw">

      <input name="first.car" type="text" value="{$first.car}"/>

  </span>

</div>

<div style="padding-top: 10px; text-align: center; clear: both;">

  <input name="complete" type="hidden" value="1"/>

  <input type="submit" value="Generate Name"/>

  <input type="reset" value="Start Over"/>

</div>
```

```
      </div>
    </form>
  </body>
  </application>
</xsl:template>


</xsl:stylesheet>
```

As you may expect, this stylesheet does little more than create the data entry interface that visitors will use to type in the information that will be shuffled to select their movie star name. Example 7-11 shows the stylesheet associated with the complete application state.

Example 7-11. complete.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:import href="params.xsl"/>


<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>


<xsl:template match="application">
  <application>
  <xsl:apply-templates/>
  <body>
   <div>
     <p>
       Your new <i>nom d'cinema</i> is:
       <b>
         <xsl:value-of select="/application/first_name"/>
         <xsl:text> </xsl:text>
         <xsl:value-of select="/application/last_name"/>
       </b>
     </p>
    </div>
    <form name="prompt" action="{$request.uri}" method="post">
      <input name="last.hotel" type="hidden" value="{$last.hotel}"/>
      <input name="last.street" type="hidden" value="{$last.street}"/>
      <input name="last.maiden" type="hidden" value="{$last.maiden}"/>
      <input name="first.middle_name" type="hidden" value="{$first.middle_name}"/>
      <input name="first.pet" type="hidden" value="{$first.pet}"/>

      <input name="first.car" type="hidden" value="{$first.car}"/>

      <input name="complete" type="hidden" value="1"/>

      <div>

        <input type="submit" value="Regenerate"/>

        <input type="button" value="Start Over" onClick="location='{$request.uri}'"/>

      </div>

    </form>

  </body>

  </application>

</xsl:template>


</xsl:stylesheet>
```

This stylesheet presents the randomly selected movie star name (passed through in the XML returned from the SAWA output class) along with a hidden form that allows the user to regenerate a new name using the same set of input data. Also, each of the state-specific stylesheets imports the *params.xsl* stylesheet. (See Example 7-12.) This is really just a tiny stylesheet fragment that allows the other stylesheets to access the CGI and other request parameters needed for the two HTML forms.

Example 7-12. params.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">


<!-- Global Params -->

<xsl:param name="request.uri"/>

<xsl:param name="last.hotel"/>

<xsl:param name="last.street"/>

<xsl:param name="last.maiden"/>

<xsl:param name="first.middle_name"/>

<xsl:param name="first.pet"/>

<xsl:param name="first.car"/>


</xsl:stylesheet>
```

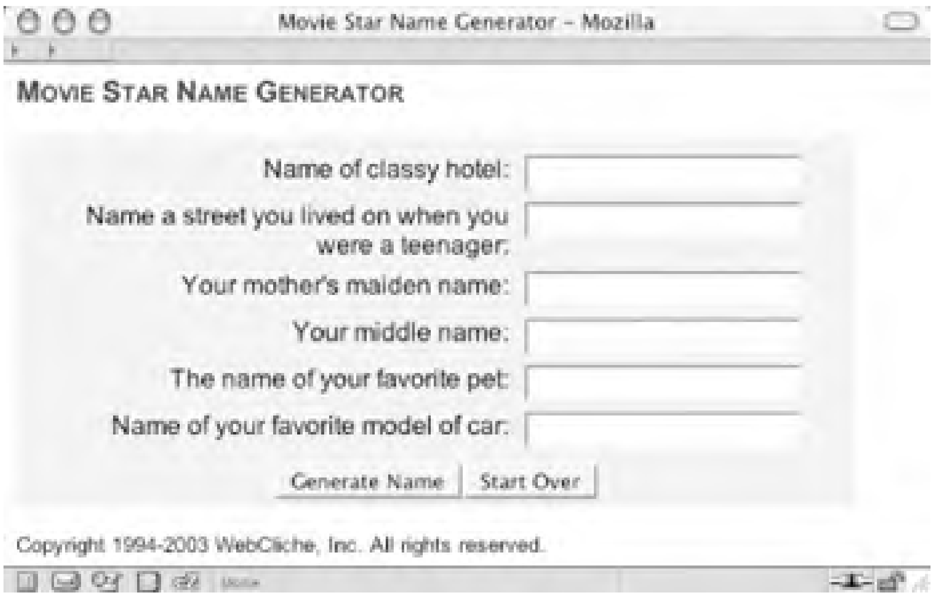
From the earlier configuration, note that the two state-specific stylesheets are first in their respective AxKit style processing chains and that both chains pass their content through to a final *common_html.xsl* stylesheet. (See Example 7-13.) This small stylesheet simply passes through the result of the previous XML to HTML transformations, while adding the global header and footer common to both application states.

Example 7-13. common_html.xsl

```
<![CDATA[
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
  <head>
    <title>Movie Star Name Generator</title>
  </head>
  <xsl:apply-templates select="/application/body"/>
</html>
</xsl:template>

<xsl:template match="body">
<body>
  <div class="pagehead">
    Movie Star Name Generator
  </div>
  <div class="message">
    <p>
      <xsl:apply-templates select="//message"/>
    </p>
  </div>
  <xsl:apply-templates select="./*"/>
  <div class="legal">
    Copyright 1994-2003 WebCliche, Inc. All rights reserved.
  </div>
</body>
</xsl:template>

  <!-- copy the HTML from the previous transformation verbatim -->
  <xsl:template match="node( )|@*">
    <xsl:copy>
      <xsl:apply-templates select="@*|node( )" />
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

That's all there is to it. A couple of tiny Perl classes, a few small XSLT stylesheets, and the application is complete. Let's look at the result. Chapter 7 shows the prompt. Figure 7-2 shows the completed result.

Figure 7-1. Movie star name generator: state two (prompt)



Movie Star Name Generator - Mozilla

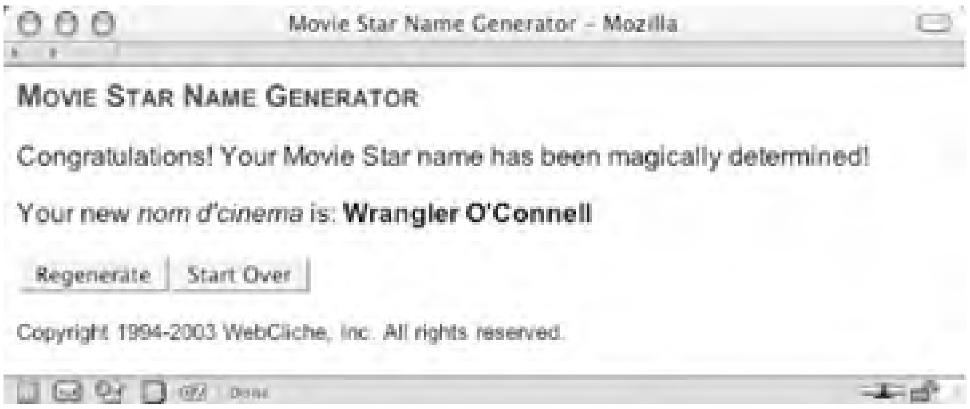
MOVIE STAR NAME GENERATOR

Name of classy hotel:	
Name a street you lived on when you were a teenager:	
Your mother's maiden name:	
Your middle name:	
The name of your favorite pet:	
Name of your favorite model of car:	

Generate Name | Start Over

Copyright 1994-2003 WebCliche, Inc. All rights reserved.

Figure 7-2. Movie star name generator: state two (complete)



Movie Star Name Generator - Mozilla

MOVIE STAR NAME GENERATOR

Congratulations! Your Movie Star name has been magically determined!

Your new *nom d'cinema* is: **Wrangler O'Connell**

Regenerate | Start Over

Copyright 1994-2003 WebCliche, Inc. All rights reserved.

You may wonder why you might combine both SAWA and AxKit in the same application, given that each offers features that the other possesses. It's really a personal choice. Having used both extensively, for me, the combination is a natural fit. SAWA's simple modularity and Perl's pure environment are well suited for generating XML content. This, plus AxKit's flexible styling options and built-in caching facilities, add up to a winning combination. In any case, whether you favor XSP, application ContentProviders, aggregate data URIs, or a combination of all three, Axkit lets developers choose the tools and techniques that suit them best.

Chapter 8. Extending AxKit

For many developers, AxKit will be a black box through which they transform and serve XML content. They will neither know nor care about its lower-level components or how things work behind the curtain. That's fine. After all, one need never know anything about lasers to use a CD player for its intended purpose. If this "I don't care, so long as it works" approach characterizes your interest in AxKit, you may want to skim the architectural overview presented shortly and skip the gory details that make up the balance of this chapter. If, on the other hand, you are the type of person who takes the "serviceable by authorized technician only" sticker on a CD player as a personal challenge, then this chapter is for you.

8.1 AxKit's Architecture

AxKit implements a modular design in which several smaller components are combined to create the larger application. Each component type has an associated configuration option that allows the default components to be changed at runtime on a resource-by-resource basis. In practice, this means that AxKit is extremely malleable and extensible—if one or another component does not do exactly what you want for a given situation, you are free to swap in a new component that does, while still taking advantage of the rest of what AxKit has to offer. In fact, AxKit's continued popularity is arguably due, in large part, to the fact that it provides a set of stock components that covers a great many common cases while still giving developers the ability to easily address special requirements without reinventing the wheel. (See Figure 8-1.)

Plug-ins

Plug-in modules stand between the user's request and the bulk of AxKit's processing cycle. They are used primarily to process the incoming request and to set any internal properties that will help AxKit respond appropriately.

Providers

Provider modules handle the task of getting the source for the documents and stylesheets that are used to serve a given request. Each resource has a ContentProvider that is responsible for getting the XML content and a StyleProvider that fetches the source of any stylesheets to be applied during processing.

Languages

Language modules provide different ways to transform the XML content being served. Often, an AxKit Language module is a thin wrapper around an existing XML processing library. For example, XPathScript is made available to AxKit via Apache::AxKit::Language::XPathScript, while Apache::AxKit::Language::LibXSLT offers XSLT transformations by creating an interface to the Gnome Project's libxslt library.

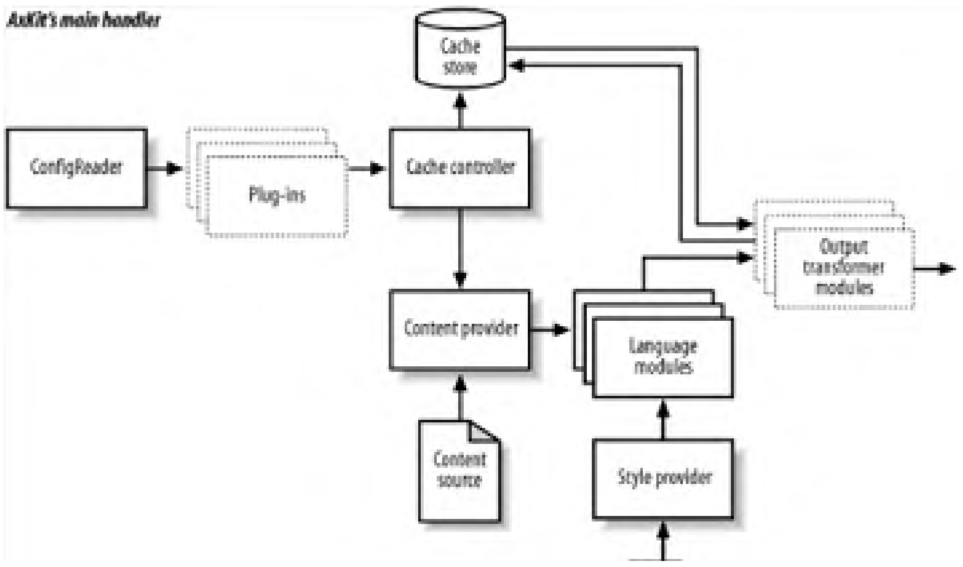
ConfigReaders

ConfigReader modules control how and from where AxKit gets its runtime configuration information.

Caches

Caching modules determine how, where, and under what conditions various resources are cached and delivered to the requesting client.

Figure 8-1. AxKit's order of execution



From Figure 8-1, you can see that first, AxKit initializes the ConfigReader class to get the runtime options needed to handle the client's request. Next, any plug-in modules configured for the current resource are loaded and run (in the order returned from the ConfigReader). Following that, the Provider and Cache modules are initialized. If all caching conditions are met, the pretransformed result is sent to the client, and processing ends there. If the resource is not cached (or the cache has expired, or caching is turned off) the Provider class responsible for fetching the source XML content is loaded, as well as the Language modules associated with the transformations to be applied to that content. Each Language module is then called, and transformed content is sent to the client. There are additional levels of detail that we will touch on while examining the component classes individually, but from the highest level, this is how AxKit works.

Before we dive into the details of extending and replacing AxKit's default components, you should be sure to have a basic understanding of Perl and its capabilities as an object-oriented language. Specifically, you should feel comfortable creating new Perl modules, especially those that are inherited subclasses of other existing modules, and the techniques for installing those modules into places where the Perl interpreter can find them. Of course, since AxKit is implemented on top of *mod_perl*, the more you know about that uniquely powerful and flexible environment, the more creative and productive you are likely to be from the very start.

The Apache configuration directives shown in each of the following sections to demonstrate how to swap in a new component type presume that the default Config- Reader component is being used. Obviously, if you implement a different way for AxKit to get its configuration data, you need to use your own custom interface to set these options.

8.2 Custom Plug-ins

Referring to the order of execution introduced earlier, you can see that plug-ins are the first AxKit components to run after the ConfigReader is created. This makes a plug-in the most logical place to put custom code that alters AxKit's processing behavior based on data from the user's request. In fact, the StyleChooser and MediaChooser modules that ship with AxKit are really only plug-ins with conventional names that hope to provide a clue about their intended use.

If the default ConfigReader class is used, plug-ins are added via the AxAddPlugin directive. Plug-ins are run in the order that they appear in the configuration file:

```
AxAddPlugin Apache::AxKit::Plugin::Passthru


# Allow Cookie-based style selection the 'forums' directory

<Directory forums>

  AxAddPlugin Apache::AxKit::StyleChooser::Cookie

</Directory>


# Reset the plug-ins list and add a new set for a particular application

<LocationMatch /about/contact>

  AxResetPlugins

  AxAddPlugin MyApp::Plugin

</LocationMatch>
```

8.2.1 Plug-in API

All AxKit plug-in modules must implement one function, *handler( )*, which is passed an Apache::Request instance for the current client request as its sole argument. Access to this request object gives plug-ins the same power to examine and alter the incoming request as a standard *mod_perl* content handler. Most often, though, they are used to set one or both of the preferred_style or preferred_media properties in the request object's pnotes table that AxKit uses later to determine which styles apply while delivering the content.

Example 8-1 shows one possible way to provide a friendlier environment for a site's international or multilingual visitors by selecting a different preferred style based on the relationship between the requesting client's Accept-Language header and the host's internal language priority list.

Example 8-1. StyleChooser::AcceptLanguage

```
package StyleChooser::AcceptLanguage;

use strict;

use Apache::Constants qw(OK);

use Cookbook::LanguagePriority;


sub handler {

    my $r = shift;

    my $lp = Cookbook::LanguagePriority->get( $r );
    my $lang_header = $r->header_in('accept-language');
    my $chosen_language = undef;

    if ( defined( $lang_header ) ) {
        $chosen_language = best_language_match( $lang_header, @{$lp->priority} );
    }
    $chosen_language ||= $r->dir_config('DefaultStyleLanguage');

    # Modifies the preferred style name
    # to <stylename>_<language>
    my $current_style = $r->notes('preferred_style') || 'default';
    $r->notes('preferred_style', $current_style . '_' . $chosen_language );

    return OK;
}

sub best_language_match {
     my $accept_lang = shift;
     my @priority_list = @_;

     # The Schwartzian Transform in action
     my @user_languages =
     map $_->[0],
     sort { $b->[1] <=> $a->[1] }
     map {
        my ( $lang, $range ) = split/\s*;\s*/, $_;
        $range = defined( $range ) && $range =~ /^q=["']?(.+?)["']?$/ ? $1 : 1;
        $lang =~ s/^(.+?)-(.+?)$/$1/;
        [ $lang, $range ];
     } split /\s*,\s*/, $accept_lang;

    foreach my $lang ( @user_languages ) {
        for ( @priority_list ) {
            return $_ if $_ eq $lang;
        }
    }
}


1;
```

With this module installed, you only need to add the appropriate AxAddPlugin directive and named styles to your configuration. Again, the named styles take the form of <stylename>_<language> that your plug-in produces.

```
# Add the AcceptLanguage plug-in
AxAddPlugin StyleChooser::AcceptLanguage


# Set the default language:(referenced by the plug-in)
PerlSetVar DefaultStyleLanguage en


# Now the styles
<AxStyleName style1_en>
    AxAddProcessor text/xsl /styles/style_one.xsl
    AxAddProcessor text/xsl /styles/english.xsl
</AxStyleName>


<AxStyleName style1_de>
    AxAddProcessor text/xsl /styles/style_one.xsl
    AxAddProcessor text/xsl /styles/german.xsl
</AxStyleName>


<AxStyleName style2_en>
    AxAddProcessor text/xsl /styles/style_two.xsl
    AxAddProcessor text/xsl /styles/english.xsl
</AxStyleName>


<AxStyleName style2_de>
    AxAddProcessor text/xsl /styles/style_two.xsl
    AxAddProcessor text/xsl /styles/german.xsl
</AxStyleName>
...
```

8.3 Custom Providers

By default, AxKit presumes that the documents that it is serving and processing are stored as plain files on the filesystem. While this is typically adequate for most cases, in some situations you may need to get data from another source. In AxKit, the mechanism for slurping in data for further processing is called a Provider.

Providers come in two flavors: ContentProviders and StyleProviders. As their names suggest, ContentProviders are responsible for fetching the source of the content being delivered, while StyleProviders handle getting the source of the stylesheets to be used to transform that content. The default class for both types, Apache::AxKit::Provider::File, reads data from XML sources on the filesystem. Alternate classes can be configured for both the ContentProvider and the StyleProvider for a given resource using the corresponding AxKit configuration directive:

```
# Set each type of Provider explicitly

AxContentProvider My::Provider

AxStyleProvider My::Other::Provider



# Or, configure both to use the same alternate class

AxProvider My::Generic::Provider
```

Custom Providers can be used to fetch content from non-XML data, to get XML data from sources other than flat files, or a combination of the two. In some cases, you are looking to take advantage of Perl's capable XML tools and other data processing facilities to *generate* an XML instance based on another source of data. In others, you simply want to read in XML for a source other than a plain file on the disk. For example, some Providers may use a SAX generator class to dynamically generate an XML document from a directory listing or Excel spreadsheet, while others may be used to extract existing XML content from a zip archive or relational database.

But wait, as you saw in Chapter 7, AxKit offers several fine options for generating dynamic content, so why would you use a Provider instead of a taglib? There is no hard and fast rule, but, in general, defining the real *source* of the content decides the matter. For example, a shopping cart application that includes a list of products from a database is probably best implemented through a taglib, while a content management system that returns a complete document from the same database may best be integrated into AxKit as a custom ContentProvider. That is, in the shopping cart page, the product list is only one component that is *included* into the content, while the data returned from the CMS *defines* the content. This distinction may seem a bit arbitrary, and from a technical point of view, it is, given that either task could be achieved by either means. Spending a few minutes considering the best approach for the task at hand can save hours of development time in the long run.

8.3.1 Provider API

Generally, a Provider is expected to offer access to the sources of the content or stylesheets that are associated with the current request, as well as certain key pieces of metadata about those sources. The data for the given resource must be returned from one of the *get_fh* (get filehandle), get_strref (get string reference), or get_dom (get Document Object instance) methods. These methods are simply variations that allow the source to be passed to AxKit in different data formats. Only one must be implemented for the Provider to work. All custom Providers should be inherited subclasses of AxKit's base Provider class Apache::AxKit::Provider or one of its subclasses:

```
package My::Provider;



use strict;

use Apache::AxKit::Provider

use vars qw( @ISA );

@ISA = qw( Apache::AxKit::Provider );
```

```
 # Override some class methods and

 # add few of your own.


1;
```

8.3.1.1 init( )

Called before all other methods, the *init* method gives the Provider a chance to perform any initialization logic needed to prepare for further processing. It is most commonly used for things such as instantiating any objects that the Provider needs to handle the request, initializing instance variables, and so on. In classes that inherit from AxKit's base Provider class (and most should), it is passed the same arguments that were passed to the constructor for the current instance:

```
sub init {

    my $self = shift;

    my %args = @_;


    $self->{content_application} = My::App->new( );

    # and so on . . .

}
```

8.3.1.2 process( )

The *process* method is used to communicate whether or not the Provider provides content for the given resource. It is passed no arguments (apart from the instance reference passed as the first argument to all Perl methods) and is expected to return 1 (or any nonzero value) to indicate that all conditions are met for the Provider to handle the request and 0 or undef, otherwise. For cases in which the Provider cannot continue, it is strongly recommended that an appropriate exception be thrown, providing an explanation as to what may have gone wrong:

```
sub process {

    my $self = shift;


    my $uri = $self->apache_request->uri( );


    # Get the data based on some URI-to-data mapping method

    # implemented elsewhere in your custom Provider

    my $data = $self->map_uri_to_data( $uri );


    if ( defined( $data ) ) {

        $self->{data} = $data;

        return 1;

    }

    else {

        throw Apache::AxKit::Exception::Error( -text => "No data associated
with URI '$uri'." );

    }

}
```

8.3.1.3 mtime( )

Used with AxKit's caching mechanism, the *mtime* method is expected to return the last modification time in seconds, since the epoch, for the current resource. If the document being provided will always be dynamic (based on user input, etc.), returning the result of Perl's built-in *time( )* function ensures that the data is never cached unexpectedly:

```
sub mtime {

    return time( ); # content is always fresh.

}
```

Implementing *mtime* correctly for cases in which the data being provided is not a plain file on the disk but is an aggregate of data from more than one source can be tricky. For example, if the content is being built as the result of an SQL query that joins several tables that may have been updated at different times, how does one determine the true last modification time for that resource? The answer is always very application-specific, and I will avoid making dubious generalizations here. It is enough to say that being able to take advantage of AxKit's caching facilities wherever possible and appropriate is a *huge* performance gain. The time spent implementing *mtime* is usually worth the investment.

8.3.1.4 get_styles( )

Called only on ContentProvider classes, the *get_styles* method is responsible for returning the final list of processors to be applied to the given resource. It is expected to return a reference to a list of style definitions that AxKit uses to transform the content. Styles are applied in the order that they appear—the first style is applied to the source content, the second to the result of the first, and so on. The style definitions take the form of an anonymous HASH reference containing two required keys: `href` whose value contains the DocumentRoot-relative path to the stylesheet to be applied, and `type`, whose value declares the MIME type associated with the Language processor to be used to apply the stylesheet:

```
my @styles = ( { type => "text/xsl",

                 href => "/styles/style1.xsl"

               },

                type => "text/xsl",

                href => "/styles/style2.xsl"

               }

             );
```

In the default ContentProviders, *get_styles( )* is used to map the current preferred style and media to any `xml-stylesheet` processing instructions contained in the source XML. If no matching styles are found, the ConfigReader's *GetMatchingProcessors* method is called, the document's root element name and Document Type Definition are evaluated against all `AxAdd*Processor` configuration directives in the current scope, and any matching styles are used instead. In all, *get_styles* is a crucial method whose default implementation provides much of AxKit's expected behavior. It should be overridden only with caution and a clear purpose.

That said, some Providers, most notably those implementing a bridge between AxKit and a content creation application that needs to define one or more stylesheet transformations to create the "view" of a given set of data for a particular application state, may need explicit control over the list of styles to apply to the content. In these cases, *get_styles* offers the most direct, least ambiguous way to define the styles to be applied. Overriding the default implementation of this method does not mean abandoning the use of the preferred style and media properties that an upstream plug-in may have set—these values are passed in as arguments to *get_styles*. The following shows how an application-based Provider may conditionally override the current list of styles, while still falling back to any default styles defined via configuration directive or `xml-stylesheet` processing instruction:

```
package My::Provider;

use vars qw( @ISA );

@ISA = qw( Apache::AxKit::Provider );


 . . .


sub get_styles {

    my $self = shift;

    my ( $preferred_media_name, $preferred_style_name ) = @_;


    my $app = $self->{some_content_application};

    my @style_list = $app->get_axkit_styles( $preferred_media_name,

$preferred_style_name );


    # If your application returned styles, use those; otherwise, fall back to the

    # default implementation in your parent Provider class.

    if ( scalar( @style_list > 0 ) ) {

        return \@style_list; # you return a reference, not the list itself.

    }

    else {

        return $self->SUPER::get_styles( $preferred_media_name, $preferred_style_name );

    }

}
```

Or, here's how an application-driven ContentProvider may alter the preferred media and style properties while letting the default Provider handle the low-level details:

```
sub get_styles {

    my $self = shift;

    my ( $preferred_media_name, $preferred_style_name ) = @_;


    my $app = $self->{some_content_application};


    my $new_preferred_style = $app->get_axkit_stylename( ) || $preferred_style_name;

    my $new_preferred_media = $app->get_axkit_medianame( ) || $preferred_media_name;


    return $self->SUPER::get_styles( $new_preferred_media, $new_preferred_style );

}
```

8.3.1.5 get_strref( )

One of three methods available for returning content, *get_strref* (get string reference) offers the ability to return the XML content for the current resource as a reference to a scalar containing the entire document as a string. For example, the following shows how a custom ContentProvider built on XML::Generator::DBI (which generates SAX events from the result of a database query) may implement *get_strref* to return a generated XML instance:

```
sub get_strref {
    my $self = shift;
    my $content = undef;

    my $writer = XML::SAX::Writer->new( Output => \$output );

    my $generator = XML::Generator::DBI->new(
                            Handler => $writer,
                            dbh => $self->{db_handle}
                            );

    eval {
        $generator->execute( $self->{sql_statement} );
    };

    if ( my $error = $@ ) {
        throw Apache::AxKit::Exception::Error( -text => "Error generating XML: $error" );
    }

    if ( length( $content ) ) {
        # you return a reference, not the scalar itself
        return \$content;
    }
    else {
        throw Apache::AxKit::Exception::Error( -text => "No data
was returned from SQL $self->{sql_statement}." );
    }

}
```

8.3.1.6 get_fh( )

Similar to *get_strref*, the *get_fh* (get filehandle) method offers a way to return data as an open filehandle. In some circumstances too complex to detail here, a filehandle requires fewer system resources than a scalar variable that contains the same document as a plain string; *get_fh* offers a way to take advantage of that optimization.

```
 # As above, but return a filehandle instead

sub get_fh {

    my $self = shift;


    # Use the Apache-friendly way to create a new filehandle

    my $handle = $self->apache_request->gensym( );


    my $writer = XML::SAX::Writer->new( Output => \$handle );


    my $generator = XML::Generator::DBI->new(

                              Handler => $writer,

                              dbh => $self->{db_handle}

                              );


    eval {

        $generator->execute( $self->{sql_statement} );

    };


    if ( my $error = $@ ) {

        throw Apache::AxKit::Exception::Error( -text => "Error generating XML: $error" );

    }


    return $handle;

}
```

8.3.1.7 get_dom( )

The *get_dom* method offers a way to return the XML data for the current resource as an XML::LibXML::Document instance. It is used most often as a means to pass the content from application frameworks such as SAWA and CGI::XMLApplication without incurring the overhead of serializing that DOM object via its *toString* method and reparsing it once it is passed into AxKit.

```
sub get_strref {

    my $self = shift;

    my $content = $self->{XML_APP}->getDom( );


    unless ( $content ) {

        throw Apache::AxKit::Exception::Error( -text => "Error generating XML,

no document object returned" );

    }
    return $content;

}
```

8.3.1.8 key( )

Called throughout AxKit, the Provider's *key* method should return a string that can be used as a persistent, unique identifier for the current resource. It is used extensively by AxKit's default caching mechanism (along with *mtime*) to both look up content that may be cached on the disk or to create the ID for a new cache entry if caching is turned on and none previously existed.

In the default file Provider, *key* simply returns the filename associated with the current request, which is sufficient in most cases. File-based alternate Providers are encouraged to do the same, or to inherit from Apache::AxKit::Provider::File and avoid implementing the *key* method altogether. In cases in which there is no one-to-one mapping between the current request URI and a file on the filesystem, a smarter *key* method is almost always required.

Suppose you use a content management system for part of your site that stores the source XML documents in a relational database. You are now faced with creating the public interface to that data. Let's go a step further and say that your CMS offers an internal hierarchical mapping that allows content objects to be selected using a path interface. Setting up the interface is easy. You only need to create a virtual URL with a `<Location>` directive and set your custom Provider as the ContentProvider for that URL. Then you can simply use any additional path information from the incoming request as the path passed to your application to retrieve the content. You must explicitly set the cache directory for this resource, since, by default, AxKit attempts to write the cache to the same directory as the requested content—in this case, a directory that does not actually exist.

```
# virtual URI for public side of your CMS

<Location /cmsapp/content>

  AxContentProvider My::CMS::Provider

  # always set an explicit cache for virtual URIs

  AxCacheDir /.mycachedir

</Location>
```

Given that all requests for content within this resource always have the same value from the request object's *filename*, you cannot just use the same strategy as the default Providers. You must use the full URI (including the additional path information) in the string returned to create a unique cache key for each document in the document store:

```
sub key {

    my $self = shift;

    my $r = $self->apache_request( );



    return $r->uri( );

}
```

You can achieve the same effect using a unique property from the content object itself:

```
sub key {

    my $self = shift;

    return $self->{content_object}->id( );

}
```

8.3.1.9 exists( )

This method is expected to return 1 (or any nonzero value) if the resource exists and is readable and 0 or undef, otherwise. Typically, a class member added to the current instance during *init* or *process* can be examined and the appropriate value returned.

```
sub process {
    my $self = shift;

    my $uri = $self->apache_request->uri( );
    my $data = $self->map_uri_to_data( $uri );

    if ( defined( $data ) ) {
        $self->{data} = $data;
                 $self->{exists} = 1;
        return 1;
    }
    else {
        throw Apache::AxKit::Exception::Error( -text => "No data associated with
URI '$uri'." );
    }
}

sub exists {
    my $self = shift;
    if ( defined( $self->{exists} ) ) {
       return 1;
    }
    else {
       return 0;
    }
}
```

It is worth mentioning again: as subclasses of one of AxKit's default Providers, most custom Providers only ever need to implement a few of these methods. Often, implementing both the *process* method to fetch and preprocess the data from the given source and one of the *get_** methods to return that data to AxKit are all that is required for a working Provider. To bring this all together, a simple Provider allows you to transparently serve both content and stylesheets from *zip* archives. (See Example 8-2.)

Example 8-2. Provider::Zip

```
package Provider::Zip;

use strict;
use vars qw($VERSION @ISA);

use Apache::AxKit::Provider::File;
use Archive::Zip qw(:ERROR_CODES);

# Inherit from the default file Provider class
@ISA = ('Apache::AxKit::Provider::File');

Archive::Zip::setErrorHandler(\&_error_handler);

sub _error_handler {
    my $error = shift;
    AxKit::Debug(3, $error);
}

sub exists {
    my $self = shift;
    return defined( $self->{zip_member} );
}

sub mtime {
    my $self = shift;
    return $self->{zip_member}->{lastModFileDateTime};
}

sub process {
    my $self = shift;
    my $zip = Archive::Zip->new( );
    if ($zip->read($self->{file}) != AZ_OK) {
        throw Apache::AxKit::Exception::IO (-text => "Couldn't read archive file
'$self->{file}'");
    }

    my $r = $self->apache_request;
    my $member;

    my $path_info = $r->path_info;
    $path_info =~ s|^/||;

    if ( $self->{zip_uri} ) {
        $member = $zip->memberNamed($self->{zip_uri});
    }
    else {
        if ($path_info) {
            $member = $zip->memberNamed($path_info);
        }
        else {
            $member = $zip->memberNamed('index.xml') || $zip->memberNamed('index.xsp');
        }
    }

    unless ( $member ) {
        throw Apache::AxKit::Exception::Declined(
                -text => "Document could not be retrieved from $self->{file}"
                );
    }

    $self->{zip_member} = $member;
    return 1;
}

sub get_strref {
    my $self = shift;

    my ($data, $status) = $self->{zip_member}->contents( );

    my $r = $self->apache_request( );

    if ($status != AZ_OK) {
        throw Apache::AxKit::Exception::Error(
                -text => "Document could not be retrieved from $self->{file}: $status"
                );
    }

    # Allow images to be served correctly
    if ( $r->path_info =~ /\.(png|gif|jpg)$/ ) {
        my $image_type = $1;
        $r->content_type( 'image/' . $image_type );
        $r->send_http_header( );
        $r->print( $data );
        throw Apache::AxKit::Exception::Declined(
                -text => "Image detected, skipping further processing."
                );
    }

    return \$data;
}

1;
```

Obviously, a production-worthy implementation would be a bit more complex, but the basic functionality exists. Once this custom Provider is installed, you only need to configure AxKit to process zip archives and then to set up special Alias and Location directives for each zip to make browsing the archived content seem transparent:

```
# Set AxKit to process zip archives
AddHandler axkit .zip

# Add an Alias, so the zipped content appears
# to be a native part of the site.
Alias /help /www/sites/myaxkithost/zips/helpdocs.zip

# And set AxKit to use Provider::Zip to fetch both
# content and stylesheets for the zipped help docs.

<Location /help>
  AxProvider Provider::Zip
</Location>
```

With these directives in place, a request to http:://localhost/help/index.xml causes AxKit to extract the file *index.xml* from the top level of the *helpdoc.zip* archive. In addition, any xml-stylesheet processing instructions found in that document whose href attribute pointed to a document at or below that same level in the archive also cause that stylesheet document to be extracted and applied at request time.

8.4 Custom Language Modules

AxKit Language modules provide various ways to transform XML content during delivery. For example, Apache::AxKit::Language::XSP provides AxKit's eXtensible Server Pages implementation, while Apache::AxKit::Language::LibXSLT is one of two Language modules that offers the ability to transform content using XSLT stylesheets. Usually, a Language module is simply a wrapper around the implementation of proven or promising XML transformation technology that allows AxKit to pass data to that processor and capture the result for delivery or further processing.

New Language modules are added to AxKit at runtime using the AxAddStyleMap directive. This directive requires two arguments: the MIME type used to associate transformations with the given Language processor and the Perl package name of the module that implements that processor:

```
# Add the Language module

AxAddStyleMap application/x-mylang Language::MyCustomLanguage



# Then, later, an AddProcessor directive that uses your new Language to apply

# the 'default.mlg' stylesheet

AxAddProcessor application/x-mylang /styles/default.mlg
```

8.4.1 Language API

The Language module interface consists of a single *handler* method expected to encapsulate the entire transformation process. This method is passed, in order: the current Apache object, a reference to the ContentProvider that will pull (or has pulled) in the original XML source, a reference to the StyleProvider that the Language module can use to get sources for any stylesheet to be applied, and a Boolean flag that indicates whether the current Language processor is the last entry in the processing pipeline.

Access to the source content (the data being transformed) varies based on two factors: whether the current Language processor is first in the processing chain, and, if not, whether the previous Language processor stored its result as a string or as an XML::LibXML::Document instance. If the given Language processor is first in the pipeline, it must call one of the ContentProvider instance's accessors methods, *get_fh( )*, *get_strref( )*, or *get_dom( )* to fetch the content to transform—the method used should be based on which is the most efficient for the processor associated with the Language module to consume. If the current Language module is *not* the first processor in the chain, the content was stored by the preceding processor in the Apache instance's pnotes table, either in the xml_string or dom_tree fields, depending on what the previous Language module returned. It sounds a little messy and convoluted, but this approach provides Language modules that are implemented on top of XML::LibXML with an easy way to accept and return preprocessed document instances (avoiding the overhead of serializing the objects to a string and reparsing them at each stage), while still allowing those modules based on other XML libraries to simply consume the content as a string.

```
sub handler {

    my ($r, $content_provider, $style_provider, $last_in_chain ) = @_;



    # Your processor wants its XML as a string



    my $content;



    # Always check for previous processing, first

    if (my $string = $r->pnotes('xml_string') ) {

        $content = $string;

        delete $r->pnotes('xml_string');
    }
    elsif ( my $dom = $r->pnotes('dom_tree') ) {
        $content = $dom->toString;
        delete $r->pnotes('dom_tree');
    }
    # If you make it this far, you are the first in the processing chain
    # and need to use the ContentProvider
    else {
        $content = $content_provider->get_strref;
    }

    my $output

    # The real processing happens here, ending with
    # $output containing the result of the transformation.

    $r->pnotes('xml_string', $output );
    return Apache::Constants::OK;
}
```

What to do with the result of the Language module's transformation also varies. Language modules whose result is returned as a string containing a well-formed XML document should store that result in the xml_string field in the Apache object's pnotes table. If the result is an XML::LibXML::Document instance, it may be stored in the pnotes dom_tree field instead. Modules that store their results in pnotes in this way are most generic, since they can appear at any stage in the processing pipeline.

However, some Language modules should not use pnotes to store their results because of the type of content generated during the transformation process. For example, the AxPoint Language module that ships with AxKit generates PDF slideshows from XML content—a result that would choke any downstream processors expecting to consume a well-formed XML document. In cases in which the result of the transformation cannot be an XML document (or XML::LibXML::Document instance), the Language module is expected to use the Apache object to send the result directly to the requesting client. While doing so limits these modules to the last position in the processing chain, it means that practically any format can be generated (PDF document, binary image, RTF text, etc.). Whether the content is stored in pnotes for further processing or sent directly to the client, the *handler* method should always return OK from the Apache::Constants class to let AxKit know that the process completed successfully

```
sub handler {
    my ($r, $content_provider, $style_provider, $last_in_chain ) = @_;


    unless ( $last_in_chain ) {
        throw Apache::AxKit::Exception::Error(
          -text => _ _PACKAGE_ _ . " is not a generic Language module and may only
appear as the last processor in the chain."
          );
    }


    my $result;
    # The transformation happens here, storing a binary

    # or other non-XML result in $result.


    # You are the last in the chain, so you can

    # send the data directly to the client using the Apache object.

    $r->print( $result );


    return Apache::Constants::OK;

}
```

As an example of an AxKit Language module, let's create an interface to Petr Cimprich's Perl implementation of STX (Streaming Transformations for XML), XML::STX. The code in Example 8-3 reflects an early beta implementation of STX. The AxKit interface used for the final implementation is likely to be quite different.

Example 8-3. Language::STX

```
package Language::STX;


use strict;

use vars qw( @ISA );

use XML::STX;

use XML::SAX::ParserFactory;

use Apache::AxKit::Language;

use XML::SAX::Writer;


@ISA = qw( Apache::AxKit::Language );


sub handler {

    my $class = shift;

    my ($r, $xml, $style, $last_in_chain) = @_;


    my ($xmlstring, $xml_doc);


    if (my $dom = $r->pnotes('dom_tree')) {

        $xml_doc = $dom;

    }

    else {

        $xmlstring = $r->pnotes('xml_string');

        delete $r->pnotes( )->{'xml_string'};

    }
    my $stx_style = undef;

    # get the source for the STX stylesheet from the StyleProvider and parse
    # it to get the compiled XSLT processor.
    my $stx_compiler = XML::STX::Compiler->new( );

    my $parser = XML::SAX::ParserFactory->parser( Handler => $stx_compiler );

    my $style_stringref = $style->get_strref( );

    eval {
        $stx_style = $parser->parse_string( $$style_stringref );
    };


    if ( my $error = $@ ) {
        throw Apache::AxKit::Exception::Error( -text => "Error parsing
STX stylesheet: $error ." );
    }

    my $result = '';
    my $error;
    my $output_handler = XML::SAX::Writer->new( Output => \$result );
    my $stx_handler = XML::STX->new( Handler => $output_handler, Sheet => $stx_style );

    if ( $xml_doc ) {
        eval {
            require XML::LibXML::SAX::Parser;
            my $p = XML::LibXML::SAX::Parser->new( Handler => $stx_handler );
            $p->generate( $xml_doc );
        };
    }
    else {
        my $p = XML::SAX::ParserFactory->parser( Handler => $stx_handler );

        if ($xmlstring) {
            eval {
                $p->parse_string( $xmlstring );
            };
            $error = $@ if $@;
        }
        else {
            $xmlstring = ${$xml->get_strref( )};
            eval {
                $p->parse_string( $xmlstring );
            };
            $error = $@ if $@;
        }
    }

    if ( length( $error ) ) {
        throw Apache::AxKit::Exception::Error(
                    -text => "STX Processing Error: $error"
                    );
    }

    delete $r->pnotes( )->{'dom_tree'};
    $r->pnotes('xml_string', $result);
    return Apache::Constants::OK;
}

1;
```

8.5 Custom ConfigReaders

By default, AxKit gets all its runtime configuration information from a set of extensions to Apache's standard configuration directives. In fact, whether the task simply required adding a new XSP taglib or setting up a complex processor-to-resource style mapping, all example applications that we have examined throughout this book rely on the fact that the default ConfigReader is being used. This does not have to be the case. Apart from the top level PerlModule directive that loads AxKit in the first place and an AxConfigreader directive that loads the new class responsible for loading the configuration data, AxKit's setup can be completely uncoupled from the Apache configuration system.

Reasons for implementing a custom ConfigReader vary. For example, the people responsible for setting up style-to-resource mappings (the part of an AxKit configuration that changes most often) may have limited or no access to the Apache configuration files. Here, providing a different way to set up those mappings while avoiding possible risks associated with doling out write access to *httpd.conf* and *.htaccess* files (or, worse, forcing developers to nag the webmaster for every change) makes everyone's life a bit easier. Or, you may decide that an XML-based configuration system such as the sitemaps in Apache Cocoon 2 fits your needs best—all you need to do is implement a ConfigReader that can extract and process the relevant AxKit configuration information from your XML configuration documents. Custom ConfigReaders are swapped in using the AxConfigReader directive:

```
AxConfigReader My::Custom::ConfigReader
```

8.5.1 ConfigReader API

The ConfigReader class implements a small army of methods responsible for providing AxKit with all its runtime configuration data. From the ContentProvider and StyleProvider classes used to get the data for the current request, to the list of plug-ins that will run, to the default style definitions—every aspect of AxKit processing can be controlled from the methods in the ConfigReader. Given this wide scope, unless you intend to separate AxKit's runtime setup as much as possible from the Apache configuration system, you should consider the wisdom of implementing your custom ConfigReader as a subclass of the default Apache::AxKit::ConfigReader and implementing only those methods required to suit your specific purpose.

8.5.1.1 get_config( )

Called during object initialization and passed a copy of the current Apache object as its sole argument, the *get_config* offers the simplest path to creating a custom Config- Reader class. This method is expected to return a hash reference containing all or some of the data used to control AxKit's configuration. While it is generally better to explicitly override a given option's accessors method, the value returned from the default classes' implementation can be altered by directly setting the appropriate underlying data structure in the reference returned from *get_config*:

```
# Return a simple, mostly hard-coded mapping for certain options

sub get_config {

  my $self = shift;

  my $r = shift;


  my %config = (

      ContentProvider => 'My::Custom::Provider',

      CacheDir        => '/www/mysite/.axkitcache'.

      DebugLevel      => 10,

      );

  return \%config;

}
```

8.5.1.2 StyleMap( )

The *StyleMap* method associates stylesheet MIME types with the Language processor used to apply those stylesheets to the content. It is expected to return a reference to a HASH whose keys are the MIME types and whose values are the package names of the Language modules associated with those types:

```
# a simple, hard-coded mapping
sub StyleMap {
    my %mapping = (
                  'text/xsl' => 'Apache::AxKit::Language::LibXSLT',
                  'application/x-xpathscript' => 'Apache::AxKit::Language::
XPathScript',
                  'application/x-xsp' => 'Apache::AxKit::Language::XSP',
                 );
    return \%mapping;
}
```

8.5.1.3 CacheDir( )

This method defines the directory that AxKit uses to store the cache files for the current request.

8.5.1.4 ContentProviderClass( )

The *ContentProviderClass* is expected to return the package name of the module used to fetch the source XML content for the current request:

```
sub ContentProviderClass {
    my $self = shift;

    # Emulate a FilesMatch block for zip archived content.
    if ( $self->{apache}->uri =~ /\.zip/ ) {
        return 'Provider::Zip';
    }
    else {
        return 'apache::AxKit::Provider::File';
    }
}
```

8.5.1.5 StyleProviderClass( )

Similar to its sister method, *ContentProviderClass*, this method should return the package name of the module responsible for getting the source for any stylesheets to be applied to the content.

8.5.1.6 DependencyChecks( )

By default, AxKit builds a list of file dependencies that it encounters during processing. For example, an XSLT stylesheet that contains an `xsl:import` element can be said to *depend on* the file that contains the stylesheet being imported. To make caching work as expected, the list of dependencies for each resource is also cached. Then during each subsequent request, the files in the dependency list are examined for changes. If a dependency has changed, the cache for the parent resource is considered invalid, and the sources are reprocessed. This design provides AxKit with its fine-tuned caching facilities, where dynamic or changed content is always fresh, but anything that can be cached is saved from further processing. In most cases, this is exactly the behavior that you want and expect.

In cases in which file dependencies are known to be stable and static, however, even greater cache performance can be gained by configuring AxKit to *skip* its dependency checks and serve cached content based solely on the last modification time of the top-level resources. This behavior is achieved by returning 0 or undef from this method.

8.5.1.7 PreferredStyle( )

The value returned by the *PreferredStyle* method declares the preferred style name for the current resource. This name is passed, along with the value returned from the ConfigReader's *PreferredMedia* method, into the ContentProvider's *get_styles* method. There it is used to match against any default configuration-based named styles or any alternate styles declared via xml-stylesheet processing instructions in the content document. See Section 4.3.1 for an overview of how named styles work and are typically used.

ConfigReader classes implementing this method do well to examine the preferred_style entry in the notes table of the Apache::Request instance for the current request. Otherwise, the default StyleChooser and other plug-in modules that use that property do not work as expected. If no style name applies to the current resource, an empty string should be returned, rather than simply undef.

```
sub PreferredStyle {
    my $self = shift;

    # give the plug-ins preferred_style precedence
    my $style = $self->{apache}->notes('preferred_style') ||
              $self->{some_application_object}->preferred_style_name( ) ||
              '';

    return $style;
}
```

8.5.1.8 PreferredMedia( )

Similar to *PreferredStyle*, the *PreferredMedia* method has final say over the media type name used to map the current resource to the appropriate styles. As with *PreferredStyle*, precedence should be given to any value stored in the preferred_media field in the notes table. That allows the MediaChooser or other plug-in modules configured for the current resource to work as expected. On the other hand, if your goal is to intentionally short-circuit the default behavior of the MediaChoosers for a given case, this is the place to do it—just be sure that you know what you are giving up.

```
sub PreferredMedia {
    my $self = shift;

    # Override the plug-in's preferred_media, but fallback to it
    # if no style is returned by the application object

    my $media = $self->{some_application_object}->preferred_media_name( ) ||
              $self->{apache}->notes('preferred_media') ||

              # explicitly set the default value here.
```

```
        'screen';

    return $media;
}
```

If all you need to do is pragmatically control the preferred style or media name for the current request, consider using a custom plug-in instead of a ConfigReader.

8.5.1.9 CacheModule( )

This method is expected to return a single scalar containing the name of the Perl package used as the cache module for the current content and stylesheet documents.

8.5.1.10 DebugLevel( )

Expected to return a value between 0 and 10, *DebugLevel* determines the level of detail and number of errors that will appear in the host's error log. The following shows how a subclass of the default ConfigReader may be used to make developers' lives a little easier by allowing the debugging level to be controlled at request time:

```
sub DebugLevel {
    my $self = shift;
    my %params = $self->{apache}->args( );

    # if you get a query string param called 'noisylog'
    # crank the debugging level to maximum

    if ( defined( $params{noisylog} ) ) {
        return 10;
    }

    # otherwise, fallback to the default
    return $self->SUPER::DebugLevel( );
}
```

8.5.1.11 LogDeclines( )

A defined return value from the *LogDeclines* configures AxKit to provide additional information in cases in which a Declined exception has been thrown.

8.5.1.12 HandleDirs( )

If this method returns a defined value, AxKit will offer up an XML representation of the items in a directory listing (which can then be transformed for delivery) instead of the default Apache directory index.

8.5.1.13 IgnoreStylePI( )

The *IgnoreStylePI* method determines whether or not the ContentProvider for the current XML source takes any xml-stylesheet processing instructions contained in that document into account when mapping the document to the current list of styles. If this method returns a nonzero value, the processing instructions are ignored.

8.5.1.14 AllowOutputCharset( )

Returning a defined value from this method indicates the intention and ability to send the final transformed content for the current resource in a character encoding other than the one that the final Language processor may have used. It is used by some of the more esoteric Language processors to avoid corrupting the (usually nontextual) data on the way to the client. Therefore, any implementation of this method must allow the value to be set by the classes that call it, rather than simply returned.

```
sub AllowOutputCharset L
    my $self = shift;


    # Use the method to set as well as get
    if ( @_ ) {
        $self->{allow_output_charset} = shift;
    }


    return self->{allow_output_charset};
}
```

8.5.1.15 OutputCharset( )

The *OutputCharset* method provides a way to configure AxKit to send the final transformed content to the client translated into a specific character encoding. Any value returned by this method causes AxKit to add a character set OutputTransformer to the current process. That value will be used to define the character encoding. Implementations of this method should always examine the return value from *AllowOutputCharset* to ensure that such translation is possible and allowed for the current request.

8.5.1.16 ErrorStyles( )

The *ErrorStyles* method is expected to return a list of style definitions used to transform a generated XML backtrace for the error encountered. If this method returns no style definitions, the XML backtrace is not generated, and the client receives the disappointing default 500 Server Error page instead. See the Section 8.3.1.4 in Section 8.3.1 earlier in this chapter for the details about the structure of the style definitions that this method is expected to return.

8.5.1.17 GzipOutput( )

Returning a defined value from this method indicates the intention to send gzipped content to the requesting client. The *DoGzip* method determines whether that actually happens.

8.5.1.18 DoGzip( )

Given a defined return value from *GzipOutput*, the *DoGzip* method is responsible for examining the requesting client's ability to handle gzipped content. If the client is found incapable, this method should return 0 or undef in spite of any value *GzipOutput* returns.

8.5.1.19 GetMatchingProcessors( )

Called from the ContentProvider's *get_styles* method, *GetMatchingProcessors* offers the ability to alter or define the list of processors applied to the current resource. Assuming that the default Provider class is used, it is passed several arguments:

- The preferred style name as defined by the *PreferredStyle* method
- The preferred media type name as defined by the *PreferredMedia* method
- The PUBLIC identifier from the DOCTYPE declaration in the source XML content
- The SYSTEM identifier from the DOCTYPE declaration in the source XML
- The top-level element name of the source document
- A reference to a list of any matching style found while examining the xml-stylesheet processing instructions in the source document
- A reference to the ContentProvider instance

If one of the default Providers (or subclasses thereof that do not implement the *get_styles* method) is used, the list of styles returned from the *GetMatchingProcessors* overrides the styles extracted from any xml-stylesheet processing instructions contained in the source document. The default ConfigReader gives explicit precedence to these styles by simply returning the list of styles passed in from the Provider if that list is not empty:

```
sub GetMatchingProcessors {
    my $self = shift;
    my ($media_name, $style_name, $dtd_public, $dtd_system, $root_name,
$styles_list, $provider) = @_;

    # give preference to the Provider is it found any matching style
    # in the processing instructions
    return @$styles_list if @$styles_list;
     . . .
}
```

See the Section 8.3.1.4 for more details about the structure and properties of the style definitions that this method is expected to return.

8.5.1.20 XSPTaglibs( )

Used by the XSP Language module to load the desired tag libraries for the current scope, the *XSPTaglibs* method is expected to return a list of the Perl package names that implement those libraries. If the XSP processor is invoked and the modules returned for this method are not yet loaded into memory, each is registered and cached into memory on the first request.

8.5.1.21 OutputTransformers( )

The *OutputTransformers* method is expected to return a list of Perl package names that define the OutputTransformer classes that will be applied to the content on its way to the client. Each class is called in turn, in the order returned from this method.

8.5.1.22 Plugins( )

The *Plugins* method is expected to return a list of the Perl package names for the plug-in modules to run for the current resource. These packages are loaded and called in the order returned from this method. For example, if you are mostly happy with using the default Apache directive-based configuration but have a plug-in that you want to be run for every request *after* any other plug-ins set up via the configuration directive, you may subclass the default ConfigReader and do the following:

```
sub Plugins {
    my $self = shift;


    # Get the list from the default implementation
    my @plugins = $self->SUPER::Plugins( );


    # And append your global plug-in to the end
    push @plugins, 'Plugin::MyGlobalPlugin';


    return @plugins;
}
```

As an example ConfigReader, let's create a subclass of the default that allows each method to be used to *set* each property explicitly. (See Example 8-4.) The goal is to provide access to the ConfigReader interface from plug-ins and other classes that may want to override certain aspects of the current configuration for a special case, while falling back to the default ConfigReader where a given property is not expressly set in the local instance. Note the addition of a *push_style* method that allows style definitions to be added to the local instance's list of styles and the localized *GetMatchingProcessors* method that returns those styles.

Example 8-4. ConfigReader::OpenAPI

```
package ConfigReader::OpenAPI;
use strict;
use Devel::Symdump;
use Apache::AxKit::ConfigReader;
use vars qw( @method_list @ISA );
@ISA = qw( Apache::AxKit::ConfigReader );


BEGIN {
    @method_list = Devel::Symdump->functions( 'Apache::AxKit::ConfigReader' );


    foreach my $method_name (@method_list) {
        my $full_mname = $method_name;
        $method_name =~ s/.+:://;
        next if grep { $method_name eq $_ } qw/ new GetMatchingProcessors dirname basename
 get_config /;
        my $full_name = _ _PACKAGE_ _ . "::$method_name";


        # Avert your eyes if hacking the symbol table scares you
        {
          no strict 'refs';
          *{$full_name} =
              sub { my $self = shift;
                  if ( @_ ) {
                      $self->{$method_name} = shift;
                  }

                  unless ( defined( $self->{$method_name} ) ) {
                      return &$full_mname($self);
                  }

                  if ( wantarray and ref( $self->{$method_name} ) eq 'ARRAY' ) {
                      return @{$self->{$method_name}};
                  }
                  else {
                      return $self->{$method_name};
                  }
                  };
        }
        }
}


sub GetMatchingProcessors {
    my $self = shift;
    my ( $medianame, $stylename, $dtd_public, $dtd_system, $root, $styles, $provider ) = @_;
    return @$styles if @$styles;

    if ( defined( $self->{style_defs} )) {
         return @$self->{style_defs};
    }
    else {
        return $self->SUPER::GetMatchingProcessors($medianame, $stylename, $dtd_public,
                                                   $dtd_system, $root, $styles, $provider);
    }
```

```
}


sub push_style {
    my $self = shift;
    $self->{style_defs} ||= [  ];
    push @{$self->{style_defs}}, @_;
}


1;
```

After this module is installed and configured, each AxKit runtime option can be set directly from within any other component class—the most likely candidate being a plug-in:

```
# Add your ConfigReader
AxConfigReader ConfigReader::OpenAPI


# Now, add a plug-in to take advantage of the open interface:
AxAddPlugin Plugin::OpenAPI
```

Here's a sample plug-in that uses the more open configuration interface:

```
package Plugin::OpenAPI;


use strict;
use AxKit;
use Apache::Constants qw(OK);
use vars qw( $config );
sub handler {
    my $r = shift;


    local $config = $AxKit::Cfg;


    # $config now contains a reference to the
    # OpenAPI ConfigReader that you can use
    # set the various runtime options.


    $config->AllowOutputCharset(1);
    $config->OutputCharset( 'Shift_JIS' );
    # etc..
    return OK;
}

1;
```

8.6 Getting More Information

Given the many ways that custom implementations of AxKit's components can be combined to meet specific needs, there's no way that the introductory module and method samples in this chapter can do more than hint at what you can do with AxKit, if you are willing to get your hands a little dirty. If you intend to extend AxKit in earnest, other sources of information can prove invaluable.

First, examine the code that is there. When getting a sense of how you can implement a given custom component, nothing is quite so instructive as stepping through one of the default implementations of that component type. Often, you will find that one of the default components does 90% of what you need to accomplish and that simply subclassing that module and covering the variance is all that's required.

Also, be sure to join the axkit-devel and axkit-users mailing lists and share your ideas. Knowing the easiest way from point A to point B is mostly a result of experience—avail yourself of the AxKit communities' wealth of talent, experience, and wild ideas. Topics range from nuts-and-bolts code-level discussions of AxKit itself to general XML web-publishing best practices and nifty AxKit tricks. You can subscribe to these lists by going to http://axkit.org/mailinglist.xml and following the directions there.

Chapter 9. Integrating AxKit with Other Tools

Perl excels at munging textual data. It was specifically designed to simplify the task of extracting and reporting data from text-based formats. As a result, it offers many built-in high-level text-processing facilities. Perl's nature is quite liberal. It presumes (rightly, I believe) that most people who write programs on a daily basis are *not* classically trained computer scientists. Therefore, it strives to be useful at every level of expertise. Given these features and the reality that the vast majority of web-programming tasks involve extracting data and presenting it in a textual format (often HTML markup) under tight deadlines, it's no wonder that Perl enjoys wide success as a web-development technology. And, by extension, you should not be surprised to find CPAN (the Perl community's extension repository) bursting with modules and packages that attempt to streamline the mundane details of web publishing.

Among the most popular types of Perl web modules are templating systems and web-application frameworks. Though the two often overlap in practice, you can distinguish between them. Templating systems are concerned solely with generating consistent document content. Web-application frameworks typically do the same but also provide a development environment that simplifies common back-end programming tasks (database access, etc.), reduces redundancy, and fosters good coding practices. From this perspective, AxKit itself is perhaps best seen as an application framework with built-in support for a variety of XML-centric templating systems.

Choosing the right system is not trivial. At the very least, it takes time to learn a given tool's syntax and common patterns well enough to decide whether they suit your needs and style. If you are like most coders, once you find a tool that flies, you tend to stay with it. If, for example, you learn the ins and outs of a tool such as Mason, and it works well for the project at hand, then you are more likely to use it for your next project and less likely to examine other solutions. On the positive side, this loyalty is the very foundation of healthy and active open source project; like-minded developers congregate around a given system, and their accumulated knowledge, code contributions, and advocacy benefit the larger user community and help bring new developers on board. On the negative side, blinkered devotion to one toolset often means that other, perhaps more suitable, tools and techniques are dismissed in favor of what is familiar.

Fortunately, AxKit does not make you choose. This chapter examines its modular, pluggable design means that many popular frameworks and templating systems can easily be integrated to provide content for, or transform content within, AxKit.

9.1 The Template Toolkit

Full of built-in features and optional extensions, Andy Wardley's Template Toolkit has proven a popular choice for web templating. Although TT2 is useful for generating practically any format, it is especially suited for generating and transforming XML content.

The Template Toolkit implements a modest set of simple directives that provide access to common templating system features: inline variable interpolation, inclusion of component templates and flat files, conditional blocks, loop constructs, exception handling, and more. Directives are combined and mixed with literal output using [% . . . %] as default delimiters. We will not cover TT2's complete syntax and feature set in any detail here (it would likely fill a book), so if you are not familiar with it, visit the project home page at http://template-toolkit.org/. A simple TT2 template looks like this:

```
<html>

<head>

  [% INCLUDE common_meta title="My Latest Rant" %]

</head>

 <body>

  [% INSERT static_header %]



  <p>

    Pleurodonts! Crinoids! Wildebeests! Lend me your, um, ears . . .

  </p>



  [% INSERT static_footer %]

 </body>

</html>
```

This simple HTML template includes and processes a separate common_meta template, setting that template's local title variable. It then inserts static page header and footer components around the document's main content. The directive syntax is not Perl, though it is rather Perl-minded (and you can embed raw Perl into your templates by setting the proper configuration options, if you want), nor is it really like XML.

Part of what makes the Template Toolkit a good fit with an XML environment is precisely that its directive and default delimiter syntax is so markedly different from the syntax of both Perl and XML. For example, XSP and other XML grammars that use Perl as the embedded language often require logic blocks to be wrapped in CDATA sections to avoid potential XML well-formedness errors caused by common Perl operators and "here" document syntax. Similarly, documents created for templating systems that use angle brackets as delimiters often cannot be processed with XML tools. Those that can often require the use of XML namespaces to distinguish between markup that is part of the application grammar and markup meant to be part of the output. The Template Toolkit's syntax deftly avoids these potential annoyances by limiting its operators to a handful of markup-friendly characters and letting different things (embedded directives) be and look substantially different from the literal output.

When considering how the Template Toolkit can be used within the context of AxKit, it is natural to first think of creating a Provider class that simply returns the content generated from the template expansion; it would be easy enough to do so. However, a closer look at the Template Toolkit's features shows that it can be useful for *transforming* XML content, as well. This means that it is probably best implemented as an AxKit Language module, rather than a Provider. In fact, integrating the Template Toolkit via the Language interface creates a unique medium that offers an alternative to both XSP (for generating content), and XSLT and XPathScript (for transforming it).

Based on what you already know from the details covered in Section 8.3, creating an AxKit Language interface for the Template Toolkit is pretty straightforward—you simply store the result of the template process in the appropriate Apache pnotes field. But to pull double duty as both a generative and transformative language, the Template Toolkit Language module needs to be a bit smarter. That is, in cases in which you would use the language to generate content, the source XML itself contains the template to be processed. There is no external stylesheet. When you want to use the Template Toolkit's powers to transform XML, then an external template is required. To be truly useful, the AxKit Template Toolkit Language module needs to be able to distinguish between these two cases. Example 9-1 is an excerpt from a simplified version of the Apache::AxKit::Language::TemplateToolkit2 class.

Example 9-1. Language::TemplateToolkit2

```
package Language::TemplateToolkit2;


use strict;
use vars qw( @ISA );
use Apache::AxKit::Language;
use Template;
@ISA = qw( Apache::AxKit::Language );


sub handler {
    my $class = shift;
    my ($r, $xml, $style, $last_in_chain) = @_;


    AxKit::Debug( 5, 'Language::TT2 called');


    my $input_string = undef;
    my $xml_string = undef;
    my $result_string = '';


    my $params = { Apache => $r };



    if (my $dom = $r->pnotes('dom_tree')) {
        $xml_string = $dom->toString;    }
    else {        $xml_string = $r->pnotes('xml_string');
        delete $r->pnotes( )->{'xml_string'};
    }
    $xml_string ||= ${$xml->get_strref( )};


    # process the source as the template, or a param passed to
    # an external template
    if ( $style->{file} =~ /NULL$/ ) {
        $input_string = $xml_string;
    }
    else {
        $params->{xmlstring} = $xml_string;
        $input_string = ${$style->get_strref( )};
    }

    my $tt = Template->new( get_tt_config($r) );
    $tt->process(\$input_string, $params, \$result_string ) ||
        throw Apache::AxKit::Exception::Error( -text => "Error processing TT2
template: " . $tt->error( ) );

    delete $r->pnotes( )->{'dom_tree'};
    $r->pnotes('xml_string', $result_string);
    return Apache::Constants::OK;
}

# helper functions, etc.
1;
```

Essentially, this module's main *handler( )* method examines the path to the stylesheet for the current transformation. If that option contains the literal value NULL as the final step, then the source content is expected to contain the template document to be processed, much the same way that XSP works. Otherwise, that path is used as the location of the external template to which the source content is passed as a parameter, named xmlstring, for further processing. This allows you to use the Template Toolkit in both generative and transformative contexts while using AxKit's configuration directive syntax and conventions to differentiate between the two cases. The following configuration snippet illustrates both uses:

```
# Add the Language module
AxAddStyleMap application/x-tt2 Apache::AxKit::Language::TemplateToolkit2

# TT2 as a generative Language such as XSP (source defines the template)
AxAddProcessor application/x-tt2 NULL

# TT2 as a transformative Language such as XSLT (external template applied to source)
AxAddProcessor application/x-tt2 /styles/mytemplate.tt2
```

9.1.1 Generating Content

Generating content using the Language::TemplateToolkit2 module follows the most common uses of the Template Toolkit, in general. It combines literal markup with special directives to construct a complete result when processed by the Template engine. Used in this way, the top-level template acts like an XSP page; that is, no external stylesheet is applied. The source document itself is what gets processed. Because the template is the source content, it must be a well-formed XML document, since AxKit will extract the root element name, document-type information, and any xml-stylesheet processing instructions that may be contained there for use with the style processor configuration directives (as it does with all content sources).

```
<?xml version="1.0"?>
<products>
    [% USE DBI( 'dbi:driver:dbname', 'user', 'passwd' ) %]
    [% FOREACH product DBI.query( 'SELECT * FROM products' ) %]
    <product id="[% product.id %]">
      <name>[% product.name %]</name>
      <description>[% product.description %]</description>
      <price>[% product.price %]</price>
      <stock>[% product.stock %]</stock>
    </product>
    [% END %]
</products>
```

This simple template uses the Template Toolkit's DBI plug-in to select product information from a relational database and generate an appropriate product element for each row returned. To configure AxKit to process this template as expected, you only need to add the correct directives to the host's *httpd.conf* or *.htaccess* file:

```
<Location /products.xml>
  AxAddProcessor application/x-tt2 NULL</emphasis>
  AxAddProcessor text/xsl /styles/generic.xsl
</Location>
```

Or, if you do not mind hardcoding styling information into the document itself, you can achieve the same effect by adding xml-stylesheet processing instructions to the template.

```
<?xml version="1.0"?>
<?xml-stylesheet type="application/x-tt2" href="NULL"?>
<?xml-stylesheet type="text/xsl" href="/styles/generic.xsl"?>]]><products>
    [% USE DBI( 'dbi:driver:dbname', 'user', 'passwd' ) %]
    [% FOREACH product DBI.query( 'SELECT * FROM products' ) %]
    <product id="[% product.id %]">
      <name>[% product.name %]</name>
      <description>[% product.description %]</description>
      <price>[% product.price %]</price>
      <stock>[% product.stock %]</stock>
    </product>
    [% END %]
</products>
```

9.1.2 Transforming Content

While XSP is solely concerned with generating content, TT2's filtering capabilities make it useful as a transformative language as well:

```
<html>
 <head><title>Our Products</title></head>
[% USE xpath = XML.XPath(xmlstring) %]
<body>
  <h1>Our Products</h1>
  [% PROCESS productlist %]
</body>
</html>

[% BLOCK productlist %]
  <table>
    <tr>
     <th>ID</th><th>Name</th><th>Description</th><th>Price</th><th>In Stock</th>
    </tr>
    [% FOREACH product = xpath.findnodes('/products/product') %]
       [% PROCESS product item = product %]
    [% END %]
  </table>
[% END %]

[% BLOCK product %]
  <tr>
    <td>[% item.getAttribute('id') %]</td>
    <td>[% item.findvalue('name') %]</td>
    <td>[% item.findvalue('description') %]</td>
    <td>[% item.findvalue('price') %]</td>
    <td>[% item.findvalue('stock') %]</td>
  </tr>
[% END %]
```

This is how the same transformation can be done with XSLT:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
  <html>
  <head><title>Our Products</title></head>
  <body>
     <h1>Our Products</h1>
     <xsl:call-template name="productlist"/>
  </body>
</html>
</xsl:template>

<xsl:template name="productlist">
  <table>
    <tr>
     <th>ID</th><th>Name</th><th>Description</th><th>Price</th><th>In Stock</th>
    </tr>
    <xsl:for-each select="/products/product">
       <xsl:apply-templates select="."/>
    </xsl:for-each>
  </table>
</xsl:template>

<xsl:template match="product">
  <tr>
    <td><xsl:value-of select="@id"/></td>
    <td><xsl:value-of select="name"/></td>
    <td><xsl:value-of select="description"/></td>
    <td><xsl:value-of select="price"/></td>
    <td><xsl:value-of select="stock"/></td>
  </tr>
</xsl:template>

</xsl:stylesheet>
```

If you favor another templating system and want to make it available as an AxKit Language module, please see Section 8.4 for more detail about creating your own custom AxKit Language module.

9.2 Providing Content via Apache::Filter

Filter chains are made available by adding the Apache::Filter extension to *mod_perl* via a PerlModule configuration directive to your *httpd.conf* file. Then, to set the filter chain for a specific resource, you set *mod_perl* as the main Apache handler and pass a whitespace-separated list of Perl classes to the PerlHandler directive. That done, the output of the first Perl content handler is sent as the input to the next, and so on (similar to AxKit's own style transformation chains):

```
PerlModule Apache::Filter


<Location /path/to/my/app>
  SetHandler perl-script
  PerlSetVar Filter On
  PerlHandler Filter1 Filter2 Filter3
</Location>
```

AxKit's standard distribution includes a simple, but powerful, Provider class, Apache::AxKit::Provider::Filter, that uses the Apache::Filter interface to capture the result of one or more *mod_perl* content handlers for further processing inside AxKit. This means that any Apache::Filter-aware handler can be used to provide XML content to AxKit.

AxKit requires that data passed in through the ContentProvider interface be a well-formed XML document. That does not mean that the source of that content must be XML. As long as the *returned result* is well-formed XML, you are free to use whatever means necessary to generate it. AxKit does not care how the XML gets there, only that it does.

9.2.1 CGI

First, let's examine how you can use a basic Perl CGI script to generate XML that is transformed and delivered by AxKit. The script itself is quite straightforward: it uses Perl's built-in *print( )* function to generate a simple XML document with a top-level root element that contains a series of field elements. Each field element contains a name attribute that holds the name of one of the server's environment variables as well as text that contains the value of that variable:

```
#!/usr/bin/perl
use strict;


print qq*<?xml version="1.0"?>
<root>
*;


foreach my $field ( keys(%ENV) ) {
    print qq*<field name="$field">$ENV{$field}</field>*;
}

print qq*</root>
*;

1;
```

Running this script as a CGI inside Apache produces a document like the following:

```
<?xml version="1.0"?>
<root>
    <field name="SCRIPT_NAME">/axkitbook/samples/chapt09/env.cgi</field>
    <field name="HTTP_ACCEPT_ENCODING">gzip,deflate</field>
    <field name="HTTP_CONNECTION">keep-alive</field>
    <field name="REQUEST_METHOD">GET</field>
    <field name="SCRIPT_FILENAME">/www/site/axkitbook/samples/chapt09/env.cgi</field>
    <field name="HTTP_ACCEPT_CHARSET">ISO-8859-1,utf-8;q=0.7,*;q=0.7</field>
    <field name="QUERY_STRING"></field>
    <field name="REQUEST_URI">/axkitbook/samples/chapt09/env.cgi</field>
    <field name="GATEWAY_INTERFACE">CGI-Perl/1.1</field>
    <!-- and so on -->
</root>
```

For this experiment to fly, you need to transform the generated XML into a form easily viewable in all web browsers. Applying the following XSLT stylesheet transforms the generated XML into an HTML document containing a two-column table of key/value pairs. For added visual clarity, the background of every second table row will be light silver.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
  <html>
  <head><title>Server Environement</title></head>
  <body>
     <xsl:apply-templates/>
  </body>
</html>
</xsl:template>

<xsl:template match="root">
  <table>
  <tr><th>Key</th><th>Value</th></tr>

     <xsl:apply-templates/>

  </table>

</xsl:template>


<xsl:template match="field">

<tr>

  <!-- alternate row colors -->

  <xsl:if test="position( ) mod 2 != 1">

    <xsl:attribute name="bgcolor">eeeeee</xsl:attribute>

  </xsl:if>


  <td><xsl:value-of select="@name"/></td>

  <td><xsl:value-of select="."/></td>

</tr>

</xsl:template>


</xsl:stylesheet>
```

You need to place the XML generated from the *env.cgi* script into AxKit so that the XSLT stylesheet can be applied. To do this, add a few key directives to the host's *httpd.conf* or a local *.htaccess* file. The following shows a sample snippet with a <Files> block containing the necessary directives:

```
<Files env.cgi>

 Options ExecCGI

 SetHandler perl-script

 PerlSetVar Filter On

 PerlHandler Apache::RegistryFilter AxKit

 AxContentProvider Apache::AxKit::Provider::Filter

 AxAddProcessor text/xsl styles/env.xsl

</Files>
```

First, set *mod_perl* as the main Apache handler for all requests by passing the value perl_script to the SetHandler directive. Next, enable content filtering by using the PerlSetVar directive to set the Perl variable Filter to On and by passing the two Perl handlers that you want to run to the PerlHandler directive—in this case, Apache::RegistryFilter and AxKit. Apache::RegistryFilter is simply a version of the Apache::Registry handler that allows most Perl CGI scripts to run unaltered inside the stricter *mod_perl* environment and that takes advantage of the Apache::Filter interface. Then, set Apache::AxKit::Provider::Filter as AxKit's ContentProvider, so the data returned from the CGI script will be passed into AxKit via the Apache::Filter. Finally, configure AxKit to apply the *env.xsl* XSLT stylesheet to that generated content by passing the appropriate MIME type and file path to the AxAddProcessor directive.

You can use this same basic configuration pattern to plug AxKit into any Perl handler that implements the appropriate Apache::Filter hooks. Once the Apache::Filter extension is loaded, setting Apache::AxKit::Provider::Filter as the ContentProvider for a given resource invisibly passes on the result of the upstream Perl handlers into AxKit; the only thing that often changes list of Perl handlers themselves.

9.2.2 Apache::ASP

Intended as a direct port of Microsoft's Active Server Pages technology, Apache::ASP offers a complete implementation of that model for use in an Apache and *mod_perl* environment. A predictable, object-oriented API provides easy access to session, application, and server data. Although Apache::ASP offers its own facilities for transforming generated content with XSLT, it assumes that you either need just one predetermined stylesheet per resource or that you will build conditional transformations by calling out to an XSLT processor directly from within the page's code. In any case, Apache::ASP's basic goals differ from AxKit's and therefore lack the extreme flexibility in chaining and choosing styles at runtime. Fortunately, the existence of Apache::Filter and AxKit's Filter Provider class means that developers can take advantage of the strengths of both environments. Here is a tiny page that reimplements the CGI script from the previous example as an ASP page:

```
<?xml version="1.0"?>
<root>
<%
my $env = $Request->ServerVariables;

foreach my $field ( keys( %$env) ) {
    $Response->Write ( qq*<field name="$field">$env->{$field}</field>* );
}
%>
</root>
```

The configuration block needed to enable the ASP-generated XML content to be processed by AxKit looks almost identical to the block used for the CGI example. In this case, a list of handlers configures Apache to process the document using Apache::ASP (instead of Apache::RegistryFilter), before passing the result to AxKit:

```
<Files env.asp>
 SetHandler perl-script
                         PerlHandler Apache::ASP AxKit
 PerlSetVar Filter On
 AxContentProvider Apache::AxKit::Provider::Filter
 AxAddProcessor text/xsl styles/env.xsl
</Files>
```

9.2.3 HTML::Mason

Another popular *mod_perl* web-application framework is HTML::Mason. As with ASP and XPathScript, Mason combines literal markup and inline Perl code, set off by special delimiters, to construct a dynamic result. Syntactically, Mason differs a bit from the others in that it offers more than one type of delimiter. Lines beginning with % are treated as free-form Perl code for creating conditional blocks. Blocks delimited by <% . . . %> are used for interpolating generated data into the result. Expressions contained by <& . . . &> are treated as paths and arguments passed to Mason components. Components are smaller, reusable bits of code and markup combined by higher-level components to create the final result. Much of Mason's popularity is based on the flexibility and modular extensibility that its component-based approach provides.

The following shows a simple Mason component that returns the list of server environment variables in the form from the previous two examples. By default, Mason looks for components, starting at the current host's DocumentRoot. Create a directory called mason_components inside that directory and save the following as a plain-text file named *env*:

```
% while (my ($key,$value) = each(%ENV)) {
  <field name="<% $key %>"><% $value %></field>
% }
```

You need the document that calls the env component. You obtain this by passing the path to components between a pair of special component delimiters. And, since that component returns only a flat list of elements, you need to wrap its interpolated value in a single top-level element to ensure that the resulting document is well-formed XML:

```
<?xml version="1.0"?>

<root>

<& /mason_components/env &>

</root>
```

The configuration block required to hand off your Mason-generated XML to AxKit for further processing is essentially identical to those from the previous two examples: set *mod_perl* as the Apache handler, toggle on the Apache::Filter extension, define the chain of Perl handlers, and set AxKit's ContentProvider to the Apache::AxKit::Provider::Filter class. Again, the only difference is that you pass the document through HTML::Mason::ApacheHandler instead of Apache::ASP or Apache::RegistryFilter, before pulling it into AxKit.

```
<Files env.html>

 SetHandler perl-script

 PerlHandler HTML::Mason::ApacheHandler AxKit

 PerlSetVar Filter On

 AxContentProvider Apache::AxKit::Provider::Filter

 AxAddProcessor text/xsl styles/env.xsl

</Files>
```

The flexibility and power provided by chaining AxKit together with other larger application frameworks, as you have done here, does not come without cost. Depending on the feature set of the framework in question, your web server processes can grow shockingly large, and you probably will not be able to serve lots of dynamic content for a popular site from a single, repurposed desktop PC. On the other hand, developer time is typically more costly than new hardware. If you or members of your team are creating application content that's been proven to be productive with a tool such as Mason, then there's no reason to abandon it. You can use Mason for what it is good for while taking advantage of the features that AxKit offers. At the very least, the Apache::Filter-based solutions discussed here provide a gentle migration path for those who want to experiment with AxKit but don't want to learn XSP or XPathScript, or how to write custom Provider classes right away.

Appendix A. AxKit Configuration Directive Reference

A.1 AxCacheDir

This option takes a single argument and sets the directory in which the cache module stores its files. These files are an MD5 hash of the filename and other information. Make sure the directory you specify is writable by the nobody user or the nobody group (or whatever user your Apache servers run as). It is probably best not to make these directories world-writable!

```
AxCacheDir /tmp/axkit_cache
```

A.2 AxNoCache

This turns caching on and off. This is a FLAG option: On or Off. The default is Off. When this flag is set, AxKit sends out Pragma: no-cache headers.

```
AxNoCache On
```

A.3 AxDebugLevel

If present, this makes AxKit send output to Apache's error log. The valid range is 0-10, with 10 producing more output. I recommend not using this option on a live server.

```
AxDebugLevel 5
```

A.4 AxTraceIntermediate

With this option, you advise AxKit to store the result of each transformation request in a special directory for debugging. This directory must exist and must be writable by the httpd. Files are stored with their full URIs, replacing slashes with | and appending a number indicating the transformation step. ".0" is the XML after the first transformation.

```
AxTraceIntermediate /tmp/axkit-trace
```

A.5 AxDebugTidy

With this option, you advise AxKit to tidy up debug dumps such as XSP scripts or XML files generated by AxTraceIntermediate. Be aware that this can slow down requests considerably, but with this enabled, spotting errors is often much easier.

```
AxDebugTidy On
```

A.6 AxStackTrace

This FLAG option says whether to maintain a stack trace with every exception. This is slightly inefficient, as it must call caller( ) several times for every exception thrown. However, it can give better debugging information.

```
AxStackTrace On
```

A.7 AxLogDeclines

This is a FLAG option: On or Off. The default is Off. When AxKit declines to process a URI, it gives a reason. Normally, this reason is not sent to the log. However, with AxLogDeclines set, the reason is logged. This is useful in figuring out why AxKit is not processing a particular file.

With this option set, the reason is logged regardless of the AxDebugLevel. However, if AxDebugLevel is 4 or higher, the file and line number *where* the DECLINE occurred is logged, but not necessarily the reason.

```
AxLogDeclines On
```

A.8 AxAddPlugin

Setting this to a module loads the module and executes the handler method of the module before any AxKit processing is done.

This allows you to set up such things as sessions, to authenticate, or to perform other actions that require no XML output, before the actual XML processing stage of AxKit.

```
AxAddPlugin MyAuthHandler
```

There is also a companion option, AxResetPlugins. Plug-in lists persist and get merged into directories, so if you want to start completely fresh, use the following:

```
AxResetPlugins AxAddPlugin MyFreshPlugin
```

As with other options that take a module, prefixing with a + sign preloads the module at compile time.

A.9 AxGzipOutput

This allows you to use the Compress::Zlib module to gzip output to browsers that support gzip compressed pages. It uses the Accept-Encoding HTTP header and information about user agents, which can support this option but don't correctly send the Accept-Encoding header. This option allows either On or Off values (default being Off). This is worth using on sites with mostly static pages because it significantly reduces outgoing bandwidth.

```
AxGzipOutput On
```

A.10 AxTranslateOutput

This option enables output character set translation. The default method is to detect the appropriate character set from the user agent's Accept-Charset HTTP header, but you can also hardcode an output character set using AxOutputCharset:

```
AxTranslateOutput On
```

A.11 AxOutputCharset

This fixes the output character set, rather than using either UTF-8 or the user's preference from the Accept-Charset HTTP header. With this option present, all output occurs in the chosen character set. The conversion uses the iconv library, which is part of GNU glibc and/or most modern Unix Systems. Do not use this option if you can avoid it. This option is enabled only if you also enable AxTranslateOutput.

```
AxOutputCharset iso-8859-1
```

A.12 AxExternalEncoding

This directive specifies the character encoding used outside of AxKit. Internally, AxKit strictly uses UTF-8 (remember that when you write taglibs!). Filenames on the filesystem and URIs requested by browsers may use a different encoding, e.g., ISO-8859-15 for most of Europe.

This is a server-global directive, so only use it within `<VirtualHost . . . >` containers or at the root level. As a side effect, this option allows you to work with non-ASCII characters in URLs even outside of AxKit. Some browsers may send URLs in their local charset, although the link that was encoded in UTF-8. Others always send UTF-8 encoded URLs, regardless of link encoding. With this option set, AxKit intercepts each request and checks if the URL came encoded in UTF-8. If so, AxKit transforms it to the character encoding specified here before Apache can resolve the request, so Apache finds the file, even if it is not an AxKit file.

This does not affect the *contents* of documents; they have their own encoding specifier (in the `<?xml . . . ?>` line). For now, it affects only filenames and URL processing.

```
AxExternalEncoding ISO-8859-1
```

A.13 AxAddOutputTransformer

Output transformers are applied just before output is sent to the browser. This directive adds a transformer to the list of transformers to be applied to the output.

```
AxAddOutputTransformer MyModule::Transformer
```

The transformer is a subroutine that accepts a line to process and returns the transformed line:

```
package MyModule; sub Transformer { my $line = shift;  . . .  return $line; }
```

You can use an output transformer to add dynamic output (such as the date and time or a customer name) to a cached page.

A.14 AxResetOutputTransformers

This resets the list of output transformers from the current directory level down:

```
# This directive takes no arguments

AxResetOutputTransformers
```

A.15 AxErrorStylesheet

If an error that occurs during processing throws an exception, the exception handler tries to find an ErrorStylesheet to use to process XML of the following format:

```
<error>

          <file>/usr/htdocs/xml/foo.xml</file>

          <msg>Something bad happened</msg>

          <stack_trace> <bt level="0">

          <file>/usr/lib/perl/site/AxKit.pm</file>

          <line>342</line> </bt>

          </stack_trace> </error>
```

There may be multiple bt tags. If an exception occurs when the error stylesheet is transforming the above XML, then a SERVER ERROR occurs and an error is written in the Apache error log.

```
AxErrorStylesheet text/xsl /stylesheets/error.xsl
```

A.16 AxAddXSPTaglib

XSP supports two types of tag libraries. The simpler is merely an XSLT or XPathScript (or other transformation language) stylesheet that transforms custom tags into the "raw" XSP tag form. The other, however, is faster, and its taglibs transform custom tags into pure code that then gets compiled. These taglibs must be loaded into the server using the AxAddXSPTaglib configuration directive.

```
# load the ESQL taglib and Util taglib

AxAddXSPTaglib AxKit::XSP::ESQL AxAddXSPTaglib AxKit::XSP::Util
```

If you prefix the module name with a + sign, the module is preloaded on server start-up (assuming that the config directive is in a *httpd.conf* file, rather than in a *.htaccess* file).

A.17 AxIgnoreStylePI

This turns off parsing and overrides stylesheet selection for XML files containing an xml-stylesheet processing instruction at the start of the file. This is a FLAG option: On or Off. The default value is Off.

```
AxIgnoreStylePI On
```

A.18 AxHandleDirs

Enable this option to allow AxKit to process directories. It creates an XML document with the contents of the requested directory. Look at sample output to discover that the format is straightforward and easy to understand.

```
AxHandleDirs On
```

A DTD for the output of AxHandleDirs is at http://axkit.org/dtd/axhandledirs.dtd.

A.19 AxStyle

This default stylesheet title is useful when a single XML resource maps to multiple choice stylesheets. One possible way to use this is to symlink the same file in different directories with *.htaccess* files specifying different AxStyle directives.

```
AxStyle "My custom style"
```

A.20 AxMedia

Very similar to the previous directive, this sets the media type. It is most useful in a *.htaccess* file that may have an entire directory for the media "handheld."

```
AxMedia tv
```

A.21 AxAddStyleMap

This is one of the more important directives. It is responsible for mapping module stylesheet MIME types to stylesheet processor modules. (The reason you do this is to make it easy to switch out different modules for the same functionality—for example, different XSLT processors.)

```
AxAddStyleMap text/xsl Apache::AxKit::Language::Sablot AxAddStyleMap

     application/x-xpathscript \ Apache::AxKit::Language::XPathScript AxAddStyleMap

     application/x-xsp \ Apache::AxKit::Language::XSP
```

If you prefix the module name with a + sign, it preloads on server startup (assuming that the config directive is in a *httpd.conf* file, rather than a *.htaccess* file).

A.22 AxResetStyleMap

Since the style map continues deep into your directory tree, it may occasionally be useful to reset the style map—for example, if you want a directory processed by a different XSLT engine:

```
# option takes no arguments.

AxResetStyleMap
```

A.23 AxAddProcessor

This directive maps all XML files to a particular stylesheet for processing. You can do this in a <Files> directive if you need to do it by file extension, or on a file-by-file basis:

```
<Files *.dkb> AxAddProcessor text/xsl /stylesheets/docbook.xsl </Files>
```

Multiple directives for the same set of files make for a chained set of stylesheet processing instructions, in which the output of one processing stage goes into the input of the next. This is especially useful for XSP processing, in which the output of the XSP processor is likely not to be HTML (or WAP or whatever your chosen output format is):

```
<Files *.xsp> # use "." to indicate that XSP gets

    processed by itself. AxAddProcessor application/x-xsp . AxAddProcessor text/xsl

    /stylesheets/to_html.xsl </Files>
```

A.24 AxAddDocTypeProcessor

This allows you to map all XML files conforming to a particular XML public identifier in the document's DOCTYPE declaration to the specified stylesheet(s):

```
AxAddDocTypeProcessor text/xsl /stylesheets/docbook.xsl \

        "-//OASIS//DTD DocBook XML V4.1.2//EN"
```

A.25 AxAddDTDProcessor

This allows you to map all XML files that specify the given DTD file or URI in the SYSTEM identifier to be mapped to the specified stylesheet(s):

```
AxAddDTDProcessor text/xsl /stylesheets/docbook.xsl \ /dtds/docbook.dtd
```

A.26 AxAddRootProcessor

This allows you to map all XML files with the given root element to be mapped to the specified stylesheet(s):

```
AxAddRootProcessor text/xsl /stylesheets/book.xsl book
```

Namespaces are fully supported via the following syntax:

```
AxAddRootProcessor text/xsl /stylesheets/homepage.xsl \ {http://myserver.com/NS/homepage}

homepage
```

This syntax was taken from James Clark's "Introduction to Namespaces" article (http://jclark.com/xml/xmlns.htm).

A.27 AxAddURIProcessor

This allows you to use a Perl regular expression to match the URI of the file in question:

```
AxAddURIProcessor text/xsl /stylesheets/book.xsl \ "book.*\.xml$"
```

A.28 AxResetProcessors

This allows you to reset the processor mappings from the current directory level down.

```
AxResetProcessors
```

From this directory down, you can completely redefine how AxKit processes certain types of files.

A.29 <AxMediaType>

This configuration directive block allows you to more finely control the mappings by specifying that the mappings (which must be specified using the Add*Processor directives above) within the block are only relevant when the requested media type is as specified in the block parameters:

```
<AxMediaType screen> AxAddProcessor text/xsl

        /stylesheets/webpage_screen.xsl </AxMediaType> <AxMediaType

        handheld> AxAddProcessor text/xsl /stylesheets/webpage_wap.xsl

        </AxMediaType> <AxMediaType tv> AxAddProcessor text/xsl

        /stylesheets/webpage_tv.xsl </AxMediaType>
```

A.30 <AxStyleName>

This configuration directive block is very similar to the above, only it specifies alternate stylesheets by name, which can then be requested via a StyleChooser:

```
<AxMediaType screen> <AxStyleName #default>

    AxAddProcessor text/xsl /styles/webpage_screen.xsl </AxStyleName>

    <AxStyleName printable> AxAddProcessor text/xsl

    /styles/webpage_printable.xsl </AxStyleName> </AxMediaType>
```

This and the above directive block can be nested and also contained within <Files> directives to give you even more control over the transformation of your XML.

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 animals on the cover of XML Publishing with AxKit are tarpans (Equus caballus gmelini). Now extinct (the last known true tarpan died in captivity in 1876), the tarpan was the original, prehistoric European wild horse. Tarpans lived in southern France and in Spain and eastward toward central Russia, and they inhabited the ancient forests and wetlands of Poland until the 18th century. Peasants who captured and tamed these wild horses would crossbreed them with the local domestic horse. The tarpans that are around today are genetic recreations of the original wild breed, using several European pony breeds that were descendants of the prehistoric tarpan. The tarpan's body is smoky gray in color, with a darker face and legs. Though it is considered a small horse, it has a large head, massive jaws, and a thick neck.

Tarpans became extinct because of the destruction of the forest, their natural habitat, to make room for villages, cities, and agriculture for the growing European human population. There are about 50 tarpans in North America today and possibly only 100 in the world. Most of the tarpans are in the United States and are owned by private individuals. They are friendly, curious, and affectionate and have a very calm disposition, which makes them suitable for children to ride; they are also presently being used in programs to aid the mentally and physically handicapped. Tarpans are very strong, sturdy, and agile, and can cover long distances without horseshoes.

Matt Hutchinson was the production editor for XML Publishing with AxKit . GEX Publishing Services provided production support. Darren Kelly, Emily Quill, Jamie Peppard, and Claire Cloutier provided quality control.

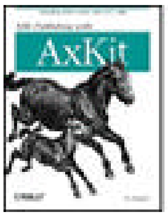
Emma Colby designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Library of Natural History, Volume 2. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

Melanie Wang designed the interior layout, based on a series design by David Futato. 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 Janet Santackas.

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.

g

g

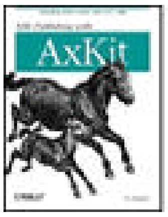


XML Publishing with AxKit

By Kip Hampton

Publisher: O'Reilly
Pub Date: June 2004
ISBN: 0-596-00216-5
Pages: 216

XML Publishing with AxKit presents web programmers the knowledge they need to master AxKit. The book features a thorough introduction to XSP (extensible Server Pages), which applies the concepts of Server Pages technologies (embedded code, tag libraries, etc) to the XML world, and covers integrating AxKit with other tools such as Template Toolkit, Apache:: Mason, Apache::ASP, and plain CGI. It also includes invaluable reference sections on configuration directives, XPathScript, and XSP.



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

XML Publishing with AxKit

By Kip Hampton

Publisher: O'Reilly
Pub Date: June 2004
ISBN: 0-596-00216-5
Pages: 216

A.1 AxCacheDir

This option takes a single argument and sets the directory in which the cache module stores its files. These files are an MD5 hash of the filename and other information. Make sure the directory you specify is writable by the nobody user or the nobody group (or whatever user your Apache servers run as). It is probably best not to make these directories world-writable!

```
AxCacheDir /tmp/axkit_cache
```

A.10 AxTranslateOutput

This option enables output character set translation. The default method is to detect the appropriate character set from the user agent's Accept-Charset HTTP header, but you can also hardcode an output character set using AxOutputCharset:

```
AxTranslateOutput On
```

A.11 AxOutputCharset

This fixes the output character set, rather than using either UTF-8 or the user's preference from the Accept-Charset HTTP header. With this option present, all output occurs in the chosen character set. The conversion uses the iconv library, which is part of GNU glibc and/or most modern Unix Systems. Do not use this option if you can avoid it. This option is enabled only if you also enable AxTranslateOutput.

```
AxOutputCharset iso-8859-1
```

A.12 AxExternalEncoding

This directive specifies the character encoding used outside of AxKit. Internally, AxKit strictly uses UTF-8 (remember that when you write taglibs!). Filenames on the filesystem and URIs requested by browsers may use a different encoding, e.g., ISO-8859-15 for most of Europe.

This is a server-global directive, so only use it within `<VirtualHost . . . >` containers or at the root level. As a side effect, this option allows you to work with non-ASCII characters in URLs even outside of AxKit. Some browsers may send URLs in their local charset, although the link that was encoded in UTF-8. Others always send UTF-8 encoded URLs, regardless of link encoding. With this option set, AxKit intercepts each request and checks if the URL came encoded in UTF-8. If so, AxKit transforms it to the character encoding specified here before Apache can resolve the request, so Apache finds the file, even if it is not an AxKit file.

This does not affect the *contents* of documents; they have their own encoding specifier (in the `<?xml . . . ?>` line). For now, it affects only filenames and URL processing.

```
AxExternalEncoding ISO-8859-1
```

A.13 AxAddOutputTransformer

Output transformers are applied just before output is sent to the browser. This directive adds a transformer to the list of transformers to be applied to the output.

```
AxAddOutputTransformer MyModule::Transformer
```

The transformer is a subroutine that accepts a line to process and returns the transformed line:

```
package MyModule; sub Transformer { my $line = shift;  . . .  return $line; }
```

You can use an output transformer to add dynamic output (such as the date and time or a customer name) to a cached page.

A.14 AxResetOutputTransformers

This resets the list of output transformers from the current directory level down:

```
# This directive takes no arguments

AxResetOutputTransformers
```

A.15 AxErrorStylesheet

If an error that occurs during processing throws an exception, the exception handler tries to find an ErrorStylesheet to use to process XML of the following format:

```
<error>

    <file>/usr/htdocs/xml/foo.xml</file>

    <msg>Something bad happened</msg>

    <stack_trace> <bt level="0">

    <file>/usr/lib/perl/site/AxKit.pm</file>

    <line>342</line> </bt>

    </stack_trace> </error>
```

There may be multiple bt tags. If an exception occurs when the error stylesheet is transforming the above XML, then a SERVER ERROR occurs and an error is written in the Apache error log.

```
AxErrorStylesheet text/xsl /stylesheets/error.xsl
```

A.16 AxAddXSPTaglib

XSP supports two types of tag libraries. The simpler is merely an XSLT or XPathScript (or other transformation language) stylesheet that transforms custom tags into the "raw" XSP tag form. The other, however, is faster, and its taglibs transform custom tags into pure code that then gets compiled. These taglibs must be loaded into the server using the AxAddXSPTaglib configuration directive.

```
# load the ESQL taglib and Util taglib

AxAddXSPTaglib AxKit::XSP::ESQL AxAddXSPTaglib AxKit::XSP::Util
```

If you prefix the module name with a + sign, the module is preloaded on server start-up (assuming that the config directive is in a *httpd.conf* file, rather than in a *.htaccess* file).

A.17 AxIgnoreStylePI

This turns off parsing and overrides stylesheet selection for XML files containing an xml-stylesheet processing instruction at the start of the file. This is a FLAG option: On or Off. The default value is Off.

```
AxIgnoreStylePI On
```

A.18 AxHandleDirs

Enable this option to allow AxKit to process directories. It creates an XML document with the contents of the requested directory. Look at sample output to discover that the format is straightforward and easy to understand.

```
AxHandleDirs On
```

A DTD for the output of AxHandleDirs is at http://axkit.org/dtd/axhandledirs.dtd.

A.19 AxStyle

This default stylesheet title is useful when a single XML resource maps to multiple choice stylesheets. One possible way to use this is to symlink the same file in different directories with *.htaccess* files specifying different AxStyle directives.

```
AxStyle "My custom style"
```

A.2 AxNoCache

This turns caching on and off. This is a FLAG option: On or Off. The default is Off. When this flag is set, AxKit sends out Pragma: no-cache headers.

```
AxNoCache On
```

A.20 AxMedia

Very similar to the previous directive, this sets the media type. It is most useful in a *.htaccess* file that may have an entire directory for the media "handheld."

```
AxMedia tv
```

A.21 AxAddStyleMap

This is one of the more important directives. It is responsible for mapping module stylesheet MIME types to stylesheet processor modules. (The reason you do this is to make it easy to switch out different modules for the same functionality—for example, different XSLT processors.)

```
AxAddStyleMap text/xsl Apache::AxKit::Language::Sablot AxAddStyleMap

        application/x-xpathscript \ Apache::AxKit::Language::XPathScript AxAddStyleMap

        application/x-xsp \ Apache::AxKit::Language::XSP
```

If you prefix the module name with a + sign, it preloads on server startup (assuming that the config directive is in a *httpd.conf* file, rather than a *.htaccess* file).

A.22 AxResetStyleMap

Since the style map continues deep into your directory tree, it may occasionally be useful to reset the style map—for example, if you want a directory processed by a different XSLT engine:

```
# option takes no arguments.

AxResetStyleMap
```

A.23 AxAddProcessor

This directive maps all XML files to a particular stylesheet for processing. You can do this in a <Files> directive if you need to do it by file extension, or on a file-by-file basis:

```
<Files *.dkb> AxAddProcessor text/xsl /stylesheets/docbook.xsl </Files>
```

Multiple directives for the same set of files make for a chained set of stylesheet processing instructions, in which the output of one processing stage goes into the input of the next. This is especially useful for XSP processing, in which the output of the XSP processor is likely not to be HTML (or WAP or whatever your chosen output format is):

```
<Files *.xsp> # use "." to indicate that XSP gets
    processed by itself. AxAddProcessor application/x-xsp . AxAddProcessor text/xsl
    /stylesheets/to_html.xsl </Files>
```

A.24 AxAddDocTypeProcessor

This allows you to map all XML files conforming to a particular XML public identifier in the document's DOCTYPE declaration to the specified stylesheet(s):

```
AxAddDocTypeProcessor text/xsl /stylesheets/docbook.xsl \

        "-//OASIS//DTD DocBook XML V4.1.2//EN"
```

A.25 AxAddDTDProcessor

This allows you to map all XML files that specify the given DTD file or URI in the SYSTEM identifier to be mapped to the specified stylesheet(s):

```
AxAddDTDProcessor text/xsl /stylesheets/docbook.xsl \ /dtds/docbook.dtd
```

A.26 AxAddRootProcessor

This allows you to map all XML files with the given root element to be mapped to the specified stylesheet(s):

```
AxAddRootProcessor text/xsl /stylesheets/book.xsl book
```

Namespaces are fully supported via the following syntax:

```
AxAddRootProcessor text/xsl /stylesheets/homepage.xsl \ {http://myserver.com/NS/homepage}

homepage
```

This syntax was taken from James Clark's "Introduction to Namespaces" article (http://jclark.com/xml/xmlns.htm).

A.27 AxAddURIProcessor

This allows you to use a Perl regular expression to match the URI of the file in question:

```
AxAddURIProcessor text/xsl /stylesheets/book.xsl \ "book.*\.xml$"
```

A.28 AxResetProcessors

This allows you to reset the processor mappings from the current directory level down.

```
AxResetProcessors
```

From this directory down, you can completely redefine how AxKit processes certain types of files.

A.29 <AxMediaType>

This configuration directive block allows you to more finely control the mappings by specifying that the mappings (which must be specified using the Add*Processor directives above) within the block are only relevant when the requested media type is as specified in the block parameters:

```
<AxMediaType screen> AxAddProcessor text/xsl
        /stylesheets/webpage_screen.xsl </AxMediaType> <AxMediaType
        handheld> AxAddProcessor text/xsl /stylesheets/webpage_wap.xsl
        </AxMediaType> <AxMediaType tv> AxAddProcessor text/xsl
        /stylesheets/webpage_tv.xsl </AxMediaType>
```

A.3 AxDebugLevel

If present, this makes AxKit send output to Apache's error log. The valid range is 0-10, with 10 producing more output. I recommend not using this option on a live server.

```
AxDebugLevel 5
```

A.30 <AxStyleName>

This configuration directive block is very similar to the above, only it specifies alternate stylesheets by name, which can then be requested via a StyleChooser:

```
<AxMediaType screen> <AxStyleName #default>

        AxAddProcessor text/xsl /styles/webpage_screen.xsl </AxStyleName>

        <AxStyleName printable> AxAddProcessor text/xsl

        /styles/webpage_printable.xsl </AxStyleName> </AxMediaType>
```

This and the above directive block can be nested and also contained within <Files> directives to give you even more control over the transformation of your XML.

A.4 AxTraceIntermediate

With this option, you advise AxKit to store the result of each transformation request in a special directory for debugging. This directory must exist and must be writable by the httpd. Files are stored with their full URIs, replacing slashes with | and appending a number indicating the transformation step. ".0" is the XML after the first transformation.

```
AxTraceIntermediate /tmp/axkit-trace
```

A.5 AxDebugTidy

With this option, you advise AxKit to tidy up debug dumps such as XSP scripts or XML files generated by AxTraceIntermediate. Be aware that this can slow down requests considerably, but with this enabled, spotting errors is often much easier.

```
AxDebugTidy On
```

A.6 AxStackTrace

This FLAG option says whether to maintain a stack trace with every exception. This is slightly inefficient, as it must call caller( ) several times for every exception thrown. However, it can give better debugging information.

```
AxStackTrace On
```

A.7 AxLogDeclines

This is a FLAG option: On or Off. The default is Off. When AxKit declines to process a URI, it gives a reason. Normally, this reason is not sent to the log. However, with AxLogDeclines set, the reason is logged. This is useful in figuring out why AxKit is not processing a particular file.

With this option set, the reason is logged regardless of the AxDebugLevel. However, if AxDebugLevel is 4 or higher, the file and line number *where* the DECLINE occurred is logged, but not necessarily the reason.

```
AxLogDeclines On
```

A.8 AxAddPlugin

Setting this to a module loads the module and executes the handler method of the module before any AxKit processing is done.

This allows you to set up such things as sessions, to authenticate, or to perform other actions that require no XML output, before the actual XML processing stage of AxKit.

```
AxAddPlugin MyAuthHandler
```

There is also a companion option, AxResetPlugins. Plug-in lists persist and get merged into directories, so if you want to start completely fresh, use the following:

```
AxResetPlugins AxAddPlugin MyFreshPlugin
```

As with other options that take a module, prefixing with a + sign preloads the module at compile time.

A.9 AxGzipOutput

This allows you to use the Compress::Zlib module to gzip output to browsers that support gzip compressed pages. It uses the Accept-Encoding HTTP header and information about user agents, which can support this option but don't correctly send the Accept-Encoding header. This option allows either On or Off values (default being Off). This is worth using on sites with mostly static pages because it significantly reduces outgoing bandwidth.

```
AxGzipOutput On
```

Appendix A. AxKit Configuration Directive Reference

1.1 Exploding a Few Myths About XML Publishing

XML and its associated technologies have generated enormous interest. XML pundits describe in florid terms how moving to XML is the first step toward a Utopian new Web, while well-funded marketing departments churn out page after page of ambiguous doublespeak about how using XML is the cure for everything from low visitor traffic to male-pattern baldness. While you may admire visionary zeal on the one hand and understand the simple desire to generate new business on the other, the unfortunate result is that many web developers are confused about what XML is and what it is good for. Here, I clear up a few of the more common fallacies about XML and its use as a web-publishing technology.

Using XML means having to memorize a pile of complex specifications.

There is certainly no shortage of specifications, recommendations, or white papers that describe or relate to XML technologies. Developing even a cursory familiarity with them all would be a full-time job. The fact is, though, that many of these specifications only describe a single *application* of XML. Unless that tool solves a specific existing need, there's no reason for a developer to try to use it, especially if you come to XML from an HTML background. A general introduction to XML's basic rules, and perhaps a quick tutorial or two that covers XSLT or another transformative tool, are all you need to be productive with XML and a tool such as AxKit. Be sane. Take a pragmatic approach: learn only what you need to deliver on the requirements at hand.

Moving to XML means throwing away all the tools and techniques that I have learned thus far.

XML is simply a way to capture data, nothing more. No tool is appropriate for all cases, and knowing how to use XML effectively simply adds another tool to your bag of tricks. Additionally (as you will see in Chapter 9), many tools you may be using today can be integrated seamlessly into AxKit's framework. You can keep doing what worked well in the past while taking advantage of what AxKit may offer in the way of additional features.

XML is totally revolutionary and will solve all of my publishing problems.

This is the opposite of the previous myth but just as common. Despite considerable propaganda to the contrary, XML offers nothing more than a way to represent information. In itself, XML does not address the issues of archiving, information retrieval, indexing, administration, or any other tasks associated with publishing documents. It may make finding or building tools to perform these tasks simpler, faster, more straightforward, or less ad hoc, but no magic is involved.

XML is useful only for transferring data structures among web services.

Two popular exchange protocols, SOAP and XML-RPC, use XML to capture data, but suggesting that this is the *only* legitimate use for XML is simply wrong. In fact, XML was originally intended primarily as a publishing technology. Tools such as SOAP only emerged later when it was discovered that XML was quite handy for capturing complex data in a way that common programming languages could share. To say that XML is only useful for transferring data between applications is a bit like saying that the ASCII text format is only useful for composing email messages—popular, yes; exclusive, no.

My project only requires documents to be available to web browsers as HTML; using XML would add complexity and overhead without adding value.

It is true—needing to deliver the same content to different target clients is a compelling reason to consider XML publishing, but it is certainly not the only one. Separating the content from its presentation also provides the ability to fundamentally alter the look and feel of an entire site without worrying about the information being communicated getting clobbered in the bargain. Similarly, new site design prototypes can be created using the actual content that will be delivered in production rather than the boilerplate filler that so often only favors the designers' sense of aesthetics.

As for performance, true XML publishing frameworks such as AxKit offer the ability to cache transformed content—even several views of the same document—and will only reprocess when either the source XML or the stylesheets being applied are modified (or when explicitly configured, reprocess for each request). The latest data available shows that AxKit can deliver cached, transformed content at roughly 90% of the speed (requests per second) offered by serving the same content as static HTML.

1.2 XML Basics

Markup technology has a long and rich history. In the 1960s, while developing an integrated document storage, editing, and publishing system at IBM, Charles Goldfarb, Edward Mosher, and Raymond Lorie devised a text-based markup format. It extended the concepts of generic coding (block-level tagging that was both machine-parsable and meaningful to human authors) to include formal, nested elements that defined the type and structure of the document being processed. This format was called the Generalized Markup Language (GML). GML was a success, and as it was more widely deployed, the American National Standards Institute (ANSI) invited Goldfarb to join its Computer Languages for Text Processing committee to help develop a text description standard-based GML. The result was the Standard Generalized Markup Language (SGML). In addition to the flexibility and semantic richness offered by GML, SGML incorporated concepts from other areas of information theory; perhaps most notably, inter-document link processing and a practical means to *programmatically validate* markup documents by ensuring that the content conformed to a specific grammar. These features (and many more) made SGML a natural and capable fit for larger organizations that needed to ensure consistency across vast repositories of documents. By the time the final ISO SGML standard was published in 1986, it was in heavy use by bodies as diverse as the Association of American Publishers, the U.S. Department of Defense, and the European Laboratory for Particle Physics (CERN).

In 1990, while developing a linked information system for CERN, Tim Berners-Lee hit on the notion of creating a small, easy-to-learn subset of SGML. It would allow people who were not markup experts to easily publish interconnected research documents over a network—specifically, the Internet. The Hypertext Markup Language (HTML) and its sibling network technology, the Hypertext Transfer Protocol (HTTP) were born. Four years later, after widespread and enthusiastic adoption of HTML by academic research circles throughout the globe, Berners-Lee and others formed the World Wide Web Consortium (W3C) in an effort to create an open but centralized organization to lead the development of the Web.

Without a doubt, HTML brought markup technology into the mainstream. Its simple grammar, combined with a proliferation of HTML-specific markup presentation applications (web browsers) and public commercial access to the Internet sparked what can only be called a popular electronic markup publishing explosion. No longer was markup solely the domain of information technology specialists working with complex, mainframe-based publishing tools inside the walls of huge organizations. Anyone with a home PC, a dial-up Internet account, and patience to learn HTML's intentionally forgiving syntax and grammar could publish his own rich hypertext documents for the rest of the wired world to see and enjoy.

HTML made markup popular, but it was a single, predefined grammar that only indicated how a document was to be presented visually in a web browser. That meant much of the flexibility offered by markup technology, in general, was simply lost. All the markup reliably communicated was how the document was supposed to *look*, not what it was supposed to *mean*. In the mid-1990s, work began at the W3C to create a new subset of SGML for use on the Web—one that provided the flexibility and best features of its predecessor but could be processed by faster, lighter tools that reflected the needs of the emerging web environment. In 1996, W3C members Tim Bray and C. M. Sperberg-McQueen presented the initial draft for this new "simplified SGML for Web"—the Extensible Markup Language (XML). Two years later in 1998, after much discussion and rigorous review, the W3C published XML 1.0 as an official recommendation.

In the six years since, interest in XML has steadily grown. While not as ubiquitous as some claim, tools to process XML are available for the most popular programming languages, and XML has been used in some fairly novel (though sometimes not always appropriate) ways. Given its generic nature, inherent flexibility, and ways in which it has (or can be) used, XML is hard to pigeonhole. It remains largely an enigma to many developers. At its core, XML is nothing, more or less, than a text-based format for applying structure to documents and other data. Uses for XML are (and will continue to be) many and varied, but looking back at its history helps to provide a reasonable context—a history inextricably bound to automated document publishing.

Many people, especially those coming to XML from a web-development background, seem to expect that it is either intended to replace HTML or that it is somehow HTML: The Next Generation—neither is the case. Although both are markup languages, HTML defines a specific markup grammar (set of elements, allowed structures) intended for consumption by a single type of application: an HTML web browser. XML, on the other hand, does not define a grammar at all. Rather, it is designed to allow developers to use (or create) a grammar that best reflects the structure and meaning of the information being captured. In other words, it gives you a clear way to create the rich, reusable source content crucial to modern adaptive web-publishing systems.

To understand the value of using a more semantically meaningful markup grammar, consider the task of publishing a poetry collection. If you know HTML and want to get the collection onto the Web quickly, you could create a document, such as the one shown in Example 1-1, for each poem.

Example 1-1. poem.html

```
<html>

  <head>

    <title>Post-Geek-chic Folk Poetry Collection</title>

  </head>
  <body>
  <h1>An Ode To Directed Acyclic Graphs</h1>
  <p><i>by: Anonymous</i></p>
  <p>
   I think that I shall never see, <br>
   a document that cannot be represented as a tree.
  </p>
  </body>
</html>
```

If your only goal is to publish your poetic gems on the Web for people to view in a browser, then once you upload the documents to the right location on an appropriate server somewhere, the job is done. What if you want to do more? At the very least, you will probably want an index document containing a list of links to the poems in your collection. If the collection remains small and time is not a consideration, you could create this index by hand. More likely, though, because you are a professional web developer, you would probably create a small script to extract information (title and author) from the poems themselves to create the index document programatically. That's when the weakness in your approach begins to show. Specifically, using HTML to mark up your poetry only gave you a way to present the work visually. In your attempt to extract the title and author's name, you are forced to impose meaning based solely on inference and your knowledge of the conventions used when marking up the poems. You can *infer* that the first h1 element contains the title of the poem, but nothing states this explicitly. You must trust that all poems in the collection will follow the same structure. In the best case, you can only guess and hope that your guess holds up in the long run.

Marking up your poetry collection in XML can help you avoid such ambiguities. It is not the use of XML, *per se*, that helps. Rather, XML gives you a familiar syntax (nested angle-bracketed tags with attributes, such as those in HTML) while offering the flexibility to choose a grammar that more intimately describes the structure and meaning of the content. It would help simplify your indexing script, for example, if something like an author element contained the author's name. You would not have to rely on an unstable heuristic such as "the string that follows the word `by,' optionally contained in an i element, that is in the first p element after the first h1 element in the document" to extract the data. Essentially, you want to use a more exact, domain-specific grammar whose structures and elements convey the meaning of the data. XML provides a means to do that.

Not surprisingly, marking up poetic content is a task that others before you have faced. A quick web search reveals several XML grammars designed for this purpose. A short evaluation of each reveals that the poemsfrag Document Type Definition (DTD) from Project Gutenberg (a volunteer effort led by the HTML Writer's Guild to make the World's great literature available as electronic text) fits your needs nicely. Using the grammar defined by *poemsfrag.dtd*, the sample poem from your collection takes the form shown in Example 1-2.

Example 1-2. poem.xml

```
<?xml version="1.0"?>
<poem>
  <title>An Ode To Directed Acyclic Graphs</title>
  <author>Anonymous</author>
  <verse>
    <line>I think that I shall never see,</line>
    <line>a document that cannot be represented as a tree.</line>
  </verse>
</poem>
```

Using this more specific grammar makes extracting the title and author data for the index document completely unambiguous—you simply grab the contents of the title and author elements, respectively. In addition, you can now easily generate other interesting metadata, such as the number of verses per poem, the average lines per verse, and so on, without dubious guesswork. Moreover, having an explicit, concrete Document Type Definition that describes your chosen grammar provides the chance to programatically validate the structure of each poem you add to the collection. This helps to ensure the integrity of the data from the outset.

Choosing the best grammar (or data model, if you must) for your content is crucial: get it right and the tools to process your documents will grow logically from the structure; get it wrong and you will spend the life of the project working around a weak foundation. Designing useful markup grammars that hold up over time is an art in itself; resist the urge to create your own just because you can. Chances are there is already a grammar available for the class of documents you will mark up. Evaluate what's available. Even if you decide to go your own way, the time spent seeing how others approached the same problem more than pays for itself.

Switching to XML and the poemsfrag grammar arguably adds significant value to your documents—the structure reveals (or imposes) the intended meaning of the content. At the very least, this reduces time wasted on messy guessing both for those marking up the poems and for those writing tools to process those poems. However, you lose something, as well. You can no longer simply upload the documents to a web server and expect browsers to do the right thing when rendering them (as you could when they were marked up as HTML). There is a gap between the grammar that is most useful to us, as authors and tool builders, and the grammar that an HTML web browser expects. Since publishing your poetry online was the goal in the first place, unless you can bridge that gap (and easily too), then really, you take a step backward.

1.3 Publishing XML Content

In the most general sense, delivering XML documents over the Web is much the same as serving any other type of document—a client application makes a request over a network to a server for a given resource, the server then interprets that request (URI, headers, content), returns the appropriate response (headers, content), and closes the connection. However, unlike serving HTML documents or MP3 files, the intended use for an XML document is not apparent from the format (or content type) itself. Further processing is usually required. For example, even though most modern web browsers offer a way to view XML documents, there is no way for the browser to know how to render your custom grammar visually. Simply presenting the literal markup or an expandable tree view of the document's contents usually communicates nothing meaningful to the user. In short, the document must be *transformed* from the markup grammar that best fits your needs into the format that best fits the expectations of the requesting client.

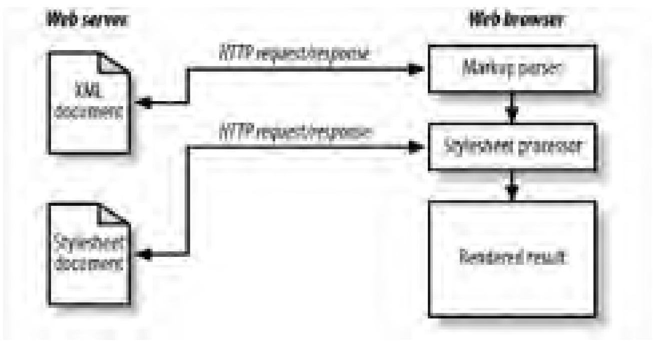
This separation between the source content and the form in which it will be presented (and the need to transform one into the other) is the heart and soul of XML publishing. Not only does making a clear distinction between content and presentation allow you to use the grammar that best captures your content, it provides a clear and logical path toward reusing that content in novel ways without altering the data's source. Suppose you want to publish the poems from the collection mentioned in the previous section as HTML. You simply transform the documents from the poemsfrag grammar into the grammar that an HTML browser expects. Later, if you decide that PDF or PostScript is the best way to deliver the content, you only need to change the way the source is transformed, not the source itself. Similarly, if your XML expresses more record-oriented data—generated from the result of an SQL query, for example—the separation between content and presentation offers a way to provide the data through a variety of interfaces just by changing the way the markup is transformed.

Although there are many ways to transform XML content, the most common is to pass the document—together with a *stylesheet* document—into a specialized processor that transforms or renders the data based on the rules set forth in the stylesheet. Extensible Stylesheet Language Transformations (XSLT) and Cascading Stylesheets (CSS) are two popular variations of this model. Putting aside features offered by various stylesheet-based transformative processors for later chapters, you still need to decide *where* the transformation is to take place.

1.3.1 Client-Side Transformations

In the client-side processing model, the remote application, typically a web browser, is responsible for transforming the requested XML document into the desired format. This is usually achieved by extracting the URL for the appropriate stylesheet from the href attribute of an xml-stylesheet processing instruction or link element contained in the document, followed by a separate request to the remote server to fetch that stylesheet. The stylesheet is then applied to the XML document using the client's internal processor and, assuming no errors occur along the way, the result of the transformation is rendered in the browser. (See Figure 1-1.)

Figure 1-2. The client-side processing model



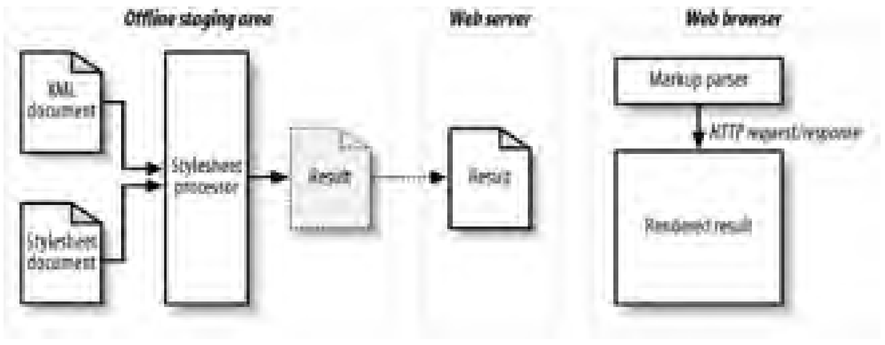
Using the client-side approach has several benefits. First, it is trivial to set up a web server to deliver XML documents in this manner—perhaps adding a few lines to the server's *mime.conf* file to ensure that the proper content type is part of the outgoing response. Also, since the client handles all processing, no additional XML tools need to be installed and configured on the server. There is no additional performance hit over and above serving static HTML pages, since documents are offered up as is, without additional processing by the server.

Client-side processing also has weaknesses. It assumes that the user at the other end of the request has an appropriate browser installed that can process and render the data correctly. Years of working around browser idiosyncrasies have taught web developers not to rely too heavily on client-side processing. The stakes are higher when you expect the browser to be solely responsible for extracting, transforming, and rendering the information for the user. Developers lose one of the important benefits of XML publishing, namely, the ability to repurpose content for different types of client devices such as PDAs, WAP phones, and set-top boxes. Many of these platforms cannot or do not implement the processors required to transform the documents into the proper format.

1.3.2 Preprocessed Transformations

Using preprocessed transformations, the appropriate stylesheets are applied to the source content offline. Only the results of those transformations are published. Typically, a staging area is used, where the source content is transformed into the desired formats. The results are copied from there into the appropriate location on the publicly available server, as shown in Figure 1-2.

Figure 1-3. The preprocessed transformation model



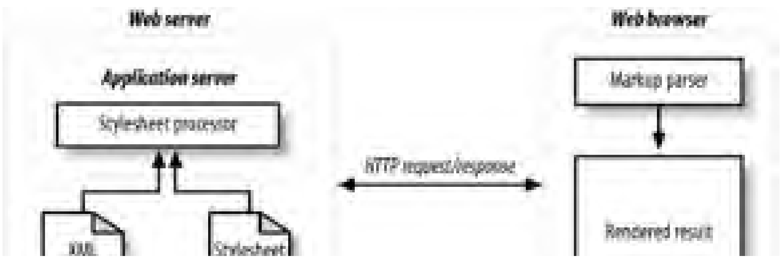
On the plus side, transforming content into the correct format ahead of time solves potential problems that can arise from expecting too much from the requesting client. That is to say, for example, that the browser gets the data that it can cope with best, just as if you authored the content in HTML to begin with, and you did not introduce any additional risk. Also, as with client-side transformations, no additional tools need to be installed on the web-server machine; any vanilla web server can capably deliver the preprocessed documents.

On the down side, offline preprocessing adds at least one additional step to publishing every document. Each time a document changes, it must be retransformed and the new version published. As the site grows or the number of team members increases, the chances of collision and missed or slow updates increase. Also, making the same content available in different formats greatly increases complexity. A simple text change, for example, requires a content transformation for each format, as well as a separate URL for each variation of every document. Scripted automation can help reduce some costs and risks, but someone must write and maintain the code for the automation process. That means more time and money spent. In any case, the static site that results from offline preprocessing lacks the ability to repurpose content on the fly in response to the client's request.

1.3.3 Dynamic Server-Side Transformations

In the server-side runtime processing model, all XML data is parsed and then transformed on the server machine before it is delivered to the client. Typically, when a request is received, the web server calls out via a server extension interface to an external XML parser and stylesheet processor that performs any necessary transformations on the data before handing it back to the web server to deliver to the client. The client application is expected only to be able to render the delivered data, as shown in Figure 1-3.

Figure 1-4. The server-side processing model



Handling all processing dynamically on the server offers several benefits. It is a given that a scripting engine or other application framework will be called on to process the XML data. As a result, the same methods that can be used from within that framework to capture information about a given request (HTTP cookies, URL parameters, POSTed form data, etc.) can be used to determine which transformations occur and on which documents. In the same way, access to the user agent and accept headers gives the developer the opportunity to detect the type of client making the connection and to transform the data into the appropriate format for that device. This ability to transform documents differently, based on context, provides the dynamic server-side processing model a level of flexibility that is simply impossible to achieve when using the client-side or preprocessed approaches.

Server-side XML processing also has its downside. Calling out to a scripting engine, which calls external libraries to process the XML, adds overhead to serving documents. A single transformation from Simplified DocBook to HTML may not require a lot of processing power. However, if that transformation is being performed for each request, then performance may become an issue for high traffic sites. Depending on the XML interface used, the in-memory representation of a given document is 10 times larger than its file size on disk, so parsing large XML documents or using complex stylesheets to transform data can cause a heavy performance hit. In addition, choosing to keep the XML processing on the server may also limit the number of possible hosting options for a given project. Most service providers do not currently offer XML processing facilities as part of their basic hosting packages, so developers must seek a specialty provider or co-locate a server machine if they do not already host their own web servers.

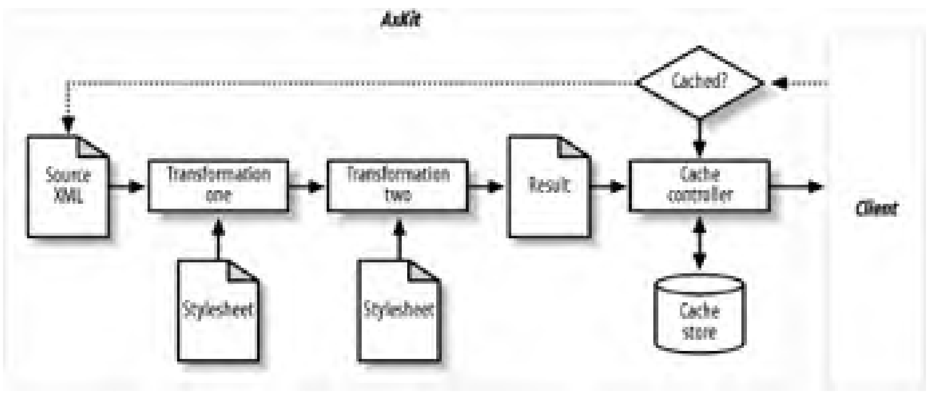
Comparing these three approaches to publishing XML content, you can generally say that dynamic server-side processing offers the greatest flexibility and extensibility for the least risk and effort. The cost of server-side processing lies largely in finding a server that provides the necessary functionality—a far more manageable cost, usually, than that of working around client-side implementations beyond your control or writing custom offline processing tools.

1.4 Introducing AxKit, an XML Application Server for Apache

Originally conceived in 2000 by Matt Sergeant as a Perl-powered alternative to the then Java-centric world of XML application servers, AxKit (short for Apache XML Toolkit) uses the *mod_perl* extension to the Apache HTTP server to turn Apache into an XML publishing and application server. AxKit extends Apache by offering a rich set of server configuration directives designed to simplify and automate common tasks associated with publishing XML content, selecting and applying transformative processes to XML content to deliver the most appropriate result.

Using AxKit's custom directives, content transformations (including *chains* of transformations) can be applied based on a variety of conditions (request URI, aspects of the XML content, and much more) on a resource-by-resource basis. Among other things, this provides the ability to set up multiple, alternate styles for a given resource and then select the most appropriate one at runtime. Also, by default, the result of each processing chain is cached to disk on the first request. Unless the source XML or the stylesheets in the chain change, all subsequent requests are to be served from the cache. Figure 1-4 illustrates the processing flow for a resource with one associated processing chain consisting of two transformations.

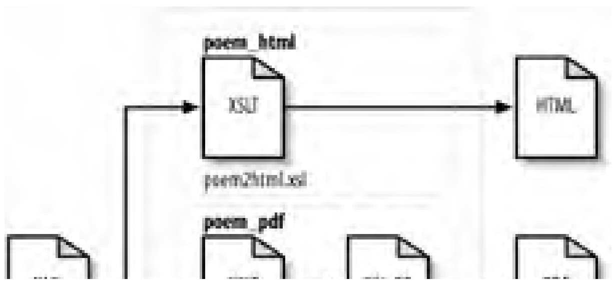
Figure 1-5. Basic two-stage processing chain

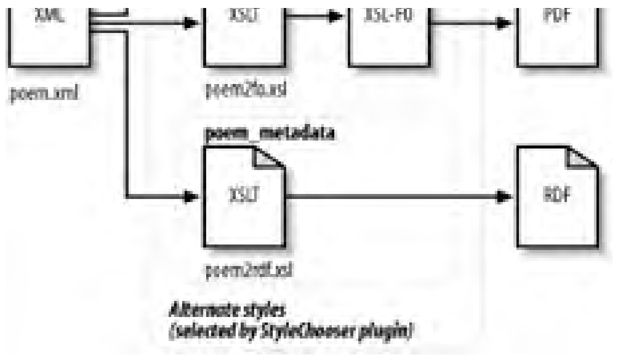


In its design, AxKit implements a modular system that divides the low-level tasks required for serving XML data across a series of swappable component classes. For example, Provider classes are responsible for fetching the sources for the content and stylesheets associated with the current request, while Language modules implement interfaces to the various transformative processors. (You can find details of each type of component class in Chapter 8.) This modular design makes AxKit quite extensible and able to cope with heterogeneous publishing strategies. Suppose that some content you are serving is stored in a relational database. You need only swap in a Provider class that selects the appropriate data for those pages from the database, while still using the default filesystem-based Provider for static documents stored on the disk. Several alternative components of various classes ship with the core AxKit distribution, and many others are available via the Comprehensive Perl Archive Network. Often, little or no custom code needs to be written. You simply drop in the appropriate component and configure its options.

We will look at each AxKit option for creating style processing chains in depth in Chapter 4. But for now, recall the collection of poems that you marked up using the poemsfrag Document Type Definition earlier in this chapter. Also, remember that when you left off, you were a bit stuck: the poems' markup captured the content in a semantically meaningful way, but by abandoning HTML as the source grammar, you lost the ability to just upload the document to a web server and expect that browsers would render it properly. This is precisely the type of task that AxKit was designed to address. Figure 1-5 illustrates a single source document containing a poem and three alternative processing chains implemented as named styles that can be selected at run-time to render that poem in various formats.

Figure 1-6. Alternate style chains





Here is a sample configuration snippet that would implement these styles, making each selectable by adding a style parameter with the appropriate value to the request's query string:

```
<Directory /poems>
    <Files *.xml>
        # choose styles based on the query string
        AxAddPlugin Apache::AxKit::StyleChooser::QueryString

        # renders the poem as HTML
        <AxStyleName poem_html>
            AxAddProcessor text/xsl /styles/poem2html.xsl
        </AxStyleName>

        # generates the poem as PDF

        <AxStyleName poem_pdf>
            AxAddProcessor text/xsl /styles/poem2fo.xsl
            AxAddProcessor application/x-xsl-fo NULL
        </AxStyleName>

        # extracts the metadata from the poem and renders it as RDF
        <AxStyleName poem_rdf>
            AxAddProcessor text/xsl /styles/poem2rdf.xsl
        </AxStyleName>

        # set a default style if none is passed explicitly
        AxStyle poem_html
    </Files>
</Directory>
```

With this in place, you can put your XML documents that use the poemsfrag grammar into the poems directory and render each poem in one of three formats. For example, a request to http://that.host/poems/mypoem.xml?style=poem_pdf returns the selected poem as a PDF document. A request for the same poem with style=poem_rdf in the query string offers the metadata about the selected poem as an RDF document. In each case, the source document does not change. Only the styles *applied* to its contents differ.

Finally, it worth noting here that AxKit is an officially sanctioned Apache Software Foundation (ASF) project. This means that AxKit is not an experimental hobbyware project. Rather it is a battle-tested framework developed and maintained by a community of committed professional developers who need to solve real-world problems. No project of any size is entirely bug-free, but AxKit's role as an ASF-blessed project means, at the very least, that it is held to a high standard of excellence. If something does go wrong, its users can fully expect an active community to be around to address the problem, both now and in the future.

Chapter 1. XML as a Publishing Technology

In the early days of the commercial Web, otherwise reasonable and intelligent people bought into the notion that simply *having* a publicly available web site was enough. Enough to get their company noticed. Enough to become a major player in the global market. Enough to capture that magical and vaguely defined commodity called *market share*. Somehow that would be enough to ensure that consumers and investors would pour out bags of money on the steps of company headquarters. In those heady days, budgets for web-related technologies appeared limitless, and the development practices of the time reflected that—it seemed perfectly reasonable to follow the celebration of a site's rollout with initial discussions about what the *next* version of that site would look like and do. (Sometimes, the next redesign was already in the works before the current redesign was even launched.) It did not matter, technically, that a site was largely hardcoded and inflexible, or that the scripts that implemented the dynamic applications were messy and impossible to maintain over time. What mattered was that the project was done quickly. If a few bad choices were made along the way, it was thought, they could always be addressed during the inevitable redesign.

Those days are gone.

The goldrush mentality has receded and companies and other organizations are looking for more from their investment in the Web. Simply having a site out there is not enough (and truly, it never was). The site must do something that measurably adds value to the organization and that value must exceed the cost of developing the site in the first place. In other words, the New Economy had a rather abrupt introduction to the rules of Business As Usual. This industry-wide belt-tightening means that web developers must adjust their approach to production. Companies can no longer afford to write off the time and energy invested in developing a web site simply to replace it with something largely similar. Developers are expected to provide dynamic, malleable solutions that can evolve over time to include new content, dynamic features, and support for new types of client software. In short, today's developers are being asked to do more with less. They need tools that can cope with major changes to a site or an application without altering the foundation that is already there.

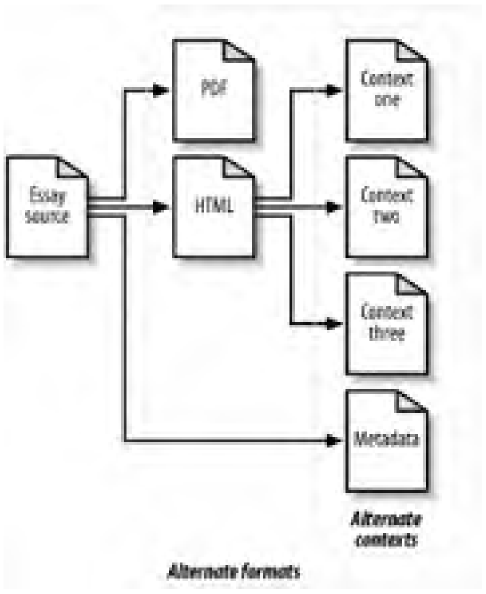
Far from being a story of gloom and doom, the slimming of web budgets has led to a natural and positive reevaluation of the tools and techniques that go into developing and maintaining online media and applications. The need to provide more options with fewer resources is driving the creative development of higher-level application and publishing frameworks that are better able to meet changing requirements over time with a minimum of duplicated effort. Ironically, in many ways, the "dot bomb" was the best thing that could have happened to web software.

One key concept behind today's more adaptive web solutions lies in making sure that the content of the site is *reusable*. By reusable content I mean that the essential information is captured (or available) in a way that lends itself to different uses or views of that data based on the context in which it is being requested. Consider, for example, the task of publishing an informal essay about the life of Jazz great Louis Armstrong. Presuming you will only be publishing this document via the Web, you still have a variety of choices about the form in which the document will be available. You could publish it in HTML for faster downloading or PDF for finer control over the visual layout. If you limit the choice to HTML, you still have many choices to make—what links, ad banners, and other supporting content will you include? Does the data include a generic boilerplate that is the same for every page on the site, or will you attempt to provide a more intimate sense of context by providing links to other related topics? If you want to offer a sense of context, how do you decide what is related? Do you frame the essay in the context of influential Jazz musicians, prominent African Americans, or famous natives of New Orleans? Given that each of these contexts is arguably valid, what if you want to present all three and let the user decide which navigational path suits her interests best? You could also say that the essay's metadata (its title, author's name, abstract summary, etc.) is really just another way of looking at the same document, albeit a highly selective and filtered one. Each of these choices represents nothing more than an alternative *contextual view* of the same content (the Armstrong essay). All that really changes is the way in which that content is presented.

Figure 1-1 shows a simple representation of your essay and some of its possible alternate views. How could you hit all of these targets? Obviously, you could hand-author the document in each of the various formats and contexts, but that would be time-consuming and unrealistic for all but the tiniest of sites. Ideally, what you want is a system that:

- Stores the data in a rich and meaningful way so users can access it easily at various levels of detail
- Provides an easy way to add alternate (expanded or filtered) views of that data without requiring changes to the source document (or, in the case of dynamic content, the code that generates it)

Figure 1-1. Multiple views of a single document



Although many web-development frameworks offer the ability to create sites in a modular fashion through reusable components, most focus largely on automating redundancy through the inclusion of common content blocks and use of code macros. These systems recognize the value of separating content from logic, but they are typically designed to construct documents in only *one* target format. That is, the templates, widgets, and content (or content-generating code) are all focused on constructing a single kind of document (usually HTML). Rendering the same content in multiple formats is cumbersome and often requires so much duplication at the component level that modularity becomes more burden than blessing. One technology, however, is firmly rooted in the ideas of generating context-specific representations of rich content sources through both modular construction and data transformation—that technology is XML.

This is where the subject of this book, AxKit, comes in. As an XML publishing and application server, AxKit begins with XML's high-level notion of reusable content and seeks to simplify the tasks associated with creating dynamic, context-sensitive representations from rich XML sources. That is, the fact that you need to deliver the same content in a variety of ways is a given, and part of what AxKit does is to provide a framework to ensure that the core content is transformed correctly for the given situation.

2.1 Installation Requirements

To get AxKit up and running, you will need:

- The Apache HTTP server (Version 1.3.*x*)
- The mod_perl Apache extension module (Version 1.26 or above)
- An XML parser written in Perl or, more commonly, one written in C that offers a Perl interface module
- The core AxKit distribution

2.1.1 Installing Apache and mod_perl

If you are running an open source or open source-friendly operating system such as GNU/Linux or one of the BSD variants (including Mac OS X), chances are good that you already have Apache and mod_perl installed. If this is the case, then you probably will not have to install them by hand. Simply make sure that you are running the most recent version of each, and skip directly to the next section. However, in some cases, using precompiled binaries of Apache and *mod_perl* proved to be problematic for people who want to use AxKit. In most cases, neither the binary in question, nor AxKit, are really broken. The problem lies in the fact that binaries built for public distribution are usually compiled with a set of general build arguments, not always well suited for specialized environments such as AxKit. If you find that all AxKit's dependencies install cleanly, but AxKit's test suite still fails, you may consider removing the binary versions and installing Apache and *mod_perl* by hand. At the time of this writing, AxKit runs only under Apache versions in the 1.3.*x* branch. Support for Apache 2.*x* is currently in development. Given that Apache 2 is quite different from previous versions, both in style and substance, the AxKit development team decided to take things slowly to ensure that AxKit for Apache 2.*x* offers the best that the new environment has to offer.

To install Apache and mod_perl from the source, you need to download the source distributions for each from http://httpd.apache.org/ and http://perl.apache.org/, respectively. After downloading, unpack both distributions into a temporary directory and *cd* into the new *mod_perl* directory. A complete reference for all options available for building the Apache server and mod_perl is far beyond the scope of this book. The following will get you up and running with a useful set of features:

```
$ perl Makefile.PL \
> EVERYTHING=1 \
> USE_APACI=1 \
> DYNAMIC=1 \
> APACHE_SRC=../apache_1.3.xxx/src \
> DO_HTTPD=1 \
> APACI_ARGS="--enable-module=so --enable-shared=info \
> --enable-shared=proxy --enable-shared=rewrite \
> --enable-shared=log_agent"
$ make
$ make install
```

All lines before the make command are build flags that are being passed to perl Makefile.PL. The \ characters are simply part of the shell syntax that allows you to divide the arguments across multiple lines. The > characters represent the shell's output, and you should not include them. Also, be sure to replace the value of the APACHE_SRC option with the actual name of the directory into which you just unpacked the Apache source.

2.1.2 XML Processing Options

As I mentioned in the introduction to this chapter, AxKit is a publishing and application framework. It is not an XML parser or XSLT processor, but it allows you to choose among these lower-level tools while ensuring that they work together in a predictable way. If you do not already have the appropriate XML processing tools installed on your server, AxKit attempts to install the minimum needed to serve transformed XML content. However, more cautious minds may prefer to install the necessary XML parser and any optional XSLT libraries to make sure they work before installing the AxKit core. Deciding which XML parsers or other libraries to install depends on your application's other XML processing needs, but the following dependency list shows which tools AxKit currently supports and which publishing features require which libraries.

Gnome XML parser (libxml2)

> Requires: XML::LibXML
>
> Required by AxKit for: eXtensible Server Pages
>
> Available from: http://xmlsoft.org/

Expat XML parser

> Requires: XML::Parser
>
> Required by AxKit for: XPathScript
>
> Available from: http://sourceforge.net/projects/expat/

Gnome XSLT processor (libxslt)

> Requires: libxml2, XML::LibXSLT
>
> Required by AxKit for: optional XSLT processing
>
> Available from: http://xmlsoft.org/XSLT/

Sablotron XSLT processor

> Requires: Expat, XML::Sablotron
>
> Required by AxKit for: optional XSLT processing
>
> Available from: http://www.gingerall.com/

You do not need to install all these libraries before installing AxKit. For example, if you plan to do XSLT processing, you need to install *either* libxslt or Sablotron, not both. However, I do strongly recommend installing both supported XML parsers: Gnome Project's libxml2 for its speed and modern features, and Expat for its wide use among many popular Perl XML modules. In any case, remember that you must install the associated Perl interface modules for any of the C libraries mentioned above, or AxKit will have no way to access the functionality that they provide.

Again, some operating system distributions include one or more of the libraries mentioned above as part of their basic packages. Be sure to upgrade these libraries before proceeding with the AxKit installation to ensure that you are building against the most recent stable code.

PREV < Day Day Up > NEXT

2.2 Installing the AxKit Core

Now that you have an environment for AxKit to work in and have some of the required dependencies installed, you are ready to install AxKit itself. For most platforms this is a fairly painless operation.

2.2.1 Using the CPAN Shell

The quickest way to install AxKit is via Perl's Comprehensive Perl Archive Network (CPAN) and the CPAN shell. Log in as root (or become superuser) and enter the following:

```
$ perl -MCPAN -e shell

> install AxKit
```

This downloads, unpacks, compiles, and installs all modules in the AxKit distribution, as well as any prerequisite Perl modules you may need. If AxKit installs without error, you may safely skip to Section 2.4. If it doesn't, see Section 2.6 for more information.

2.2.2 From the Tarball Distribution

The latest AxKit distribution can always be found on the Apache XML site at http://xml.apache.org/dist/axkit/. Just download the latest tarball, unpack it, and *cd* to the newly created directory. As root, enter the following:

```
 $ perl Makefile.PL

 $ make

 $ make test

 $ make install
```

This compiles and installs all modules in the AxKit distribution. Just like the CPAN shell method detailed above, AxKit's installer script automatically attempts to install any module prerequisites it encounters. If *make* stops this process with an error, skip on to Section 2.6 for help. Otherwise, if everything goes smoothly, you can skip ahead to Section 2.4.

In addition to the stable releases available from CPAN and axkit.org, the latest development version is available from the AxKit project's anonymous CVS archive:

```
cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic login
```

Brave souls who like to live on the edge or who may be interested in helping with AxKit development can check it out. When prompted for a password, enter: `anoncvs`. You may now check out a piping hot version of AxKit:

```
<![CDATA[cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic co xml-axkit]]>
```

Installing the CVS version of AxKit is otherwise identical to installing from the tarball.

2.3 Installing AxKit on Win 32 Systems

As of this writing, AxKit's support for the Microsoft Windows environment should be considered experimental. Anyone who decides to put such a server into production does so at her own risk. AxKit *will* run in most cases. (Win9x users are out of luck.) If you are looking for an environment in which to learn XML web-publishing techniques, then AxKit on Win32 is certainly a viable choice.

If you do not already have ActiveState's Windows-friendly version of Perl installed, you must first download and install that before proceeding. It is available from http://www.activestate.com/. I suggest you get the latest version from the 5.8.*x* branch. In addition, you need the Windows port of the Apache web server. You can obtain links to the Windows installer from http://httpd.apache.org/. Be sure to grab the latest in the 1.3.*x* branch. Next, grab the official Win32 binaries for libxml2 and libxslt from http://www.zlatkovic.com/libxml.en.html and follow the installation instructions there.

After you install Apache, Perl libxml2, and libxslt, you can install AxKit using ActiveState's *ppm* utility (which was installed when you installed ActivePerl). Simply open a command prompt, and type the following:

```
C:\> ppm

ppm> repository add theoryx http://theoryx5.uwinnipeg.ca/ppms

ppm> install mod_perl-1

ppm> install libapreq-1

ppm> install XML-LibXML

ppm> install XML-LibXSLT

ppm> install AxKit-1
```

Finally, add the following line to your *httpd.conf* and start Apache:

```
LoadModule perl_module modules/mod_perl.so
```

This combination of commands and packages should give you a workable (albeit experimental) AxKit on your Windows system. If things go wrong, be sure to join the AxKit user's mailing list and provide details about the versions of various packages you tried, your Windows version, and relevant output from your error logs.

2.4 Basic Server Configuration

As you will learn in later chapters, AxKit offers quite a number of runtime configuration options that allow fine-grained control over every phase of the XML processing and delivery cycle. Getting a basic working configuration requires very little effort, however. In fact, AxKit ships with a sample configuration file that can be included into Apache's main server configuration (or used as a road map for adding the configuration directives manually, if you decide to go that way instead).

Copy the *example.conf* file in the AxKit distribution's *examples* directory into Apache's *conf* directory, renaming it *axkit.conf*. Then, add the following to the bottom of your *httpd.conf* file:

```
# AxKit Setup

Include conf/axkit.conf
```

You now need to edit the new *axkit.conf* file to match the XML processing libraries that you installed earlier by uncommenting the AxAddStyleMap directives that correspond to tools you chose. For example, if you installed libxslt and XML::LibXSLT, you would uncomment the AxAddStyleMap directive that loads AxKit's interface to LibXSLT. Example 2-1 helps to clarify this.

Example 2-1. Sample axkit.conf fragment

```
# Load the AxKit core.

PerlModule AxKit



# Associates Axkit with a few common XML file extensions

AddHandler axkit .xml .xsp .dkb .rdf



# Uncomment to add XSLT support via XML::LibXSLT

# AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT



# Uncomment to add XSLT support via Sablotron

# AxAddStyleMap text/xsl Apache::AxKit::Language::Sablot



# Uncomment to add XPathScript Support

# AxAddStyleMap application/x-xpathscript Apache::AxKit::Language::XPathScript



# Uncomment to add XSP (eXtensible Sever Pages) support

# AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP
```

The one hard-and-fast rule about configuring AxKit is that the PerlModule directive that loads the AxKit core into Apache via *mod_perl must* appear at the top lexical level of your *httpd.conf* file, or one of the files that it includes. All other AxKit configuration directives may appear as children of other configuration directive blocks in whatever way best suits your server policy and application needs, but the PerlModule AxKit line *must* appear only at the top level.

2.5 Testing the Installation

AxKit's distribution comes with a fairly complete test suite that typically runs as part of the installation process. Running the *make test* command in the root of the AxKit source directory fires up a new instance of the Apache server on an alternate port with AxKit enabled. It then examines the output of a series of test requests made to that instance that exercise various aspects of AxKit's functionality. *make test* runs automatically by default if you are installing AxKit via the CPAN shell. If all test scripts pass during the *make test* process, you can be sure that you have a working AxKit installation and are ready to proceed.

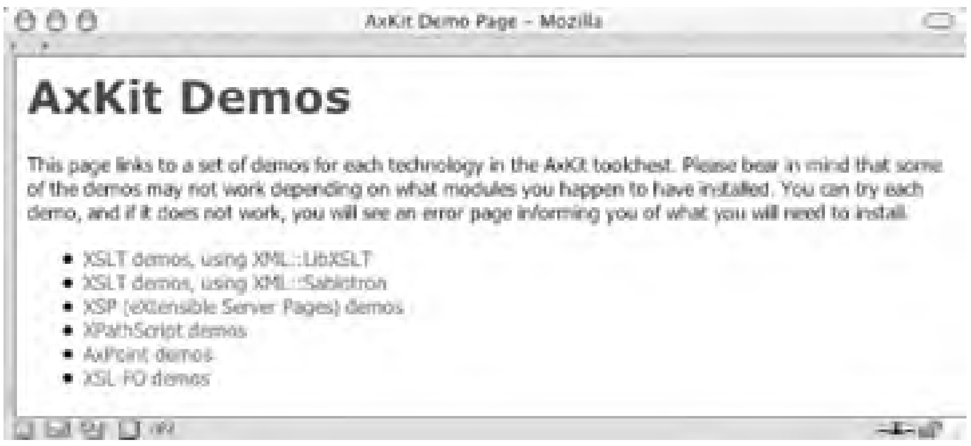
In addition to the automated test suite, AxKit comes with a set of demonstration files that you can also use to test your new installation. To install the demo, copy the *demo* directory and its contents from the root of the AxKit distribution into an appropriate directory to which you have write access. The configuration file in the *demo* directory presumes that you will copy the *demo* directory into */opt/axkit*. So if you choose another location, be sure to edit *all* paths in the demo's *axkit.conf* file to reflect your choice.

Before the demo will work, you need to include the *axkit.conf* contained in the new demo directory into your server's *httpd.conf* file. For example, if you installed the demo in */opt/axkit* (again, the default), you would add the following:

```
# AxKit Demo

Include /opt/axkit/demo/axkit.conf
```

Start (or stop and restart) the Apache server and point a browser to *http://localhost/axkit/*. You should see a page congratulating you on your new AxKit installation. This page also presents a number of links that allow you to test AxKit's various moving parts. For example, if you chose to install libxslt and its Perl interface XML::LibXSLT to use as an XSLT processor, you would click on the XSLT demos, using the XML::LibXSLT link to verify that AxKit works and is configured properly to those libraries to transform XML documents, as shown in Figure 2-1.

Figure 2-1. Proof of a successful demo AxKit installation



AxKit Demos

This page links to a set of demos for each technology in the AxKit toolchest. Please bear in mind that some of the demos may not work depending on what modules you happen to have installed. You can try each demo, and if it does not work, you will see an error page informing you of what you will need to install.

- XSLT demos, using XML::LibXSLT
- XSLT demos, using XML::Sablotron
- XSP (eXtensible Server Pages) demos
- XPathScript demos
- AxPoint demos
- XSL-FO demos

If you receive an error when you click on one of the demo links, verify that you have the associated libraries for that demo installed. If you go back and install any processor that AxKit supports, there is no need to reinstall the demo. Just reload the demo index and click on the appropriate link to verify that the new libraries work. You must, however, stop and start Apache (not just restart it) for AxKit to pick up the new interfaces.

If all goes as expected, congratulations. You have installed a working version of AxKit and are ready to get down to business.

2.6 Installation Troubleshooting

As I mentioned in this chapter's introduction, AxKit's core consists largely of code that glues other things together. In practice, this means that most errors encountered while installing AxKit are due to external dependencies that are missing, broken, out of date, or invisible to AxKit's Makefile. Including a complete list of various errors that may be encountered among AxKit's many external dependencies is not realistic here. It would likely be outdated before this book is on the shelves. In general, though, you can use a number of compile-time options when building AxKit. They will help you diagnose (and in many cases, fix) the cause of the trouble. AxKit's *Makefile.PL* recognizes the following options:

`DEBUG=1`

This option causes the Makefile to produce copious amounts of information about each step of the build process. Although wading through the sheer amount of data this option produces can be tedious, you can diagnose most installation problems (missing or unseen libraries, etc.) by setting this flag.

`NO_DIRECTIVES=1`

This option turns off AxKit's apache configuration directives, which means you must set these via Apache's `PerlSetVar` directives instead. Use this option only in extreme cases in which AxKit's custom configuration directives conflict with those of another Apache extension module. (These cases are very rare, but they do happen.)

`EXPAT_OPTS=" . . . "`

This option is relevant only if you do not have the Expat XML parser installed and decide to install it when installing AxKit. This argument takes a list of options to be passed to libexpat's *./configure* command. For example, `EXPAT_OPTS="--prefix=/usr"` installs libexpat in */usr/lib*, rather than the default location.

`LIBS="-L/path/to/somelib -lsomelib"`

This option allows you to set your library search path. It is primarily useful for pointing the Makefile to external libraries that you are sure are installed but, for some reason, are being missed during the build process.

`INC="-I/path/to/somelib/include"`

This option is like `LIBS`, but it sets the include search path.

2.6.1 Where to Go for Help

If you get stuck at any point during the installation process, do not despair. There are still other resources available to help you get up and running. In addition to this book, there are other sources of AxKit documentation, as well as a strong AxKit user community that is willing and able to help.

2.6.1.1 Installed AxKit documentation

Most Perl modules that comprise the AxKit distribution include a level of documentation. In many cases, these documents are quite detailed. You can access this information using the standard *perldoc* utility typically installed with Perl itself. Just type *perldoc <modulename>*, in which `<modulename>` is the package name of the module that you want to read the docs from. The following list provides a general overview of the information you can find in the various modules.

AxKit

The documentation in *AxKit.pm* provides a brief overview of each AxKit configuration directive, including simple examples.

Example: *perldoc AxKit*

*Apache::AxKit::Language::**

The modules in this package namespace provide support for the various XML processing and transformation languages such as XSLT, XSP, and XPathScript.

Example: *perldoc Apache::AxKit::Language::XSP* provides an XSP language reference.

*Apache::AxKit::Provider::**

The modules in this namespace provide AxKit with the ability to fetch and read the sources for the XML content and stylesheets that it will use when serving the current request.

Example: *perldoc Apache::AxKit::Provider::Filter* shows the documentation for a module that allows an upstream PerlHandler (such as Apache::ASP or Mason) to generate content.

*Apache::AxKit::Plugin::**

Modules in this namespace provide extensions to the basic AxKit functionality.

Example: *perldoc Apache::AxKit::Plugin::Passthru* offers documentation for the Passthru plug-in, which allows a "source view" of the XML document being processed based on the presence or absence of a specific query string parameter.

*Apache::AxKit::StyleChooser::**

The modules in this namespace offer the ability to set the name of a preferred transformation style in environments that provide more than one way to transform documents for a given media type.

Example: *perldoc Apache::AxKit::StyleChooser::Cookie* shows the documentation for a module that allows stylesheet transformation chains to be selected based on the value of an HTTP cookie sent from the requesting client.

Additional user-contributed documentation is also available from the AxKit project's web site at http://axkit.org/. Not only does the project site offer several useful tutorials, it also provides a user-editable Wiki that often contains the latest platform-specific installation instructions, as well as many other AxKit tips, tricks, and ideas.

2.6.1.2 Mailing lists

The AxKit project sports a lively and committed user base with lots of friendly folks who are willing to help. Even if you are not having trouble, I highly recommend joining the axkit-users mailing list. The amount of traffic is modest, the signal-to-noise ratio is high, and topics range from specific AxKit installation questions to general discussions of XML publishing best practices. You can subscribe online by visiting http://axkit.org/mailinglist.xml or by sending an empty email message to mailto:axkit-users-subscribe@axkit.org.

You can find browsable archives of axkit-users at:

- http://axkit.org/cgi-bin/ezmlm-cgi/3
- http://www.mail-archive.com/axkit-users@axkit.org/index.html

Topics relating specifically to AxKit development are discussed on the axkit-devel list. Generally, you should post most questions, bug reports, patches, etc., to axkit-users. If you want to contribute to the AxKit codebase, then axkit-devel is the place for you. You can subscribe to the development list by sending an empty message to mailto: axkit-dev-subscribe@xml.apache.org.

In addition to the mailing lists, the AxKit community also maintains an #axkit IRC channel for discussing general AxKit topics. The IRC server hosting the channel changes periodically, so check the AxKit web site for details.

Chapter 2. Installing AxKit

AxKit combines the power of Perl's rich and varied XML processing facilities with the flexibility of the Apache web server. Rather than implementing such an environment in a monolithic package, as some application servers do, it takes a more modular approach. It allows developers to choose the lower-level tools such as XML parsers and XSLT processors for themselves. This neutrality with respect to lower-level tools gives AxKit the ability to adapt and incorporate new, better performing, or more feature-rich tools as quickly as they appear. That flexibility costs, however. You will probably have to install more than just the AxKit distribution to get a working system.

3.1 Preparation

By design, XML processing tools are less forgiving about what they accept than the HTML browsers that you may be used to working with. Omitting a closing tag when creating an element in an HTML page, for example, may cause an undesirable result when the page is rendered, but the browser usually tries to recover gracefully and render *something* for you to see. In contrast, omitting an end tag when creating an element in a document that an XML parser will consume results in a fatal well-formedness error, and no such recovery is possible. In the context of AxKit (in which all XML processing happens on the server), this means that if you pass in a bad document, AxKit sends no content to the client. At best, you see an error message that indicates where things went wrong. To avoid frustration, take a little time to familiarize yourself with the XML processing tools available to you. At the very least, investigate how the XML parser you installed can be used from the command line to verify a document's well-formedness and validity. Being able to catch bad documents going in reduces the overall number of potentially user-visible errors. The ability to verify that your content and stylesheets are at least syntactically correct can make finding the cause of an error easier.

Even more than with a static HTML-based site, starting with a good directory structure is key to creating an easy-to-maintain XML-based site. The time and labor-saving benefits of having predictable paths for images, CSS stylesheets, etc. also apply to the files associated with XML processing. It's a good idea to get in the habit of creating a *stylesheets* (or similarly named) directory at the base of the host's DocumentRoot when you start a new project.

If you installed the AxKit demonstration site or included the sample *axkit.conf* in your main Apache configuration (covered in Section 2.4), you do not need to alter the web server's configuration at all. If not, follow the directions there, or add the following lines to the web server's *httpd.conf,* and stop and restart the server before proceeding:

```
PerlModule AxKit

AddHandler axkit .xml .xsp .dkb .rdf



AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT

AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP

AxAddStyleMap application/x-xpathscript Apache::AxKit::Language::XPathScript
```

These directives load the AxKit core into Apache, set up the required Language transformation processors, and configure Apache to process all files that end in *.xml*, *.xsp*, *.dkb*, and *.rdf* with AxKit. With an appropriate directory structure and configuration in place, you can move on to creating the XML documents that you want to publish.

3.2 Creating the Source XML Documents

Often, many benefits of using an XML publishing framework such as AxKit become obvious only later in a project's life (e.g., the ability to easily add new heavy-duty features to an existing site, or the power to completely change the look and feel of an entire site without touching its content). Given this, any examples you may choose for this introduction will surely fall short of illustrating AxKit's real power. Accepting the notion that the task at hand is a bit absurd frees you to have a little fun with it while still learning the basics. Let's run with the absurdity, and imagine that you are charged with the task of publishing a small site on the very silly subject of cryptozoology.

Cryptozoology (literally, *the study of hidden animals*) is concerned with the gathering and analysis of data related to animals that are frequently reported by local residents or found in popular folklore, but whose existence the scientific community has not formally recognized. Familiar examples include the Yeti, Loch Ness Monster, and Mokele-Mbembe.

The first document for your site, *cryptozoo.xml*, contains a list of cryptozoological species (called *cryptids* by insiders). (See Example 3-1.)

Example 3-1. cryptozoo.xml

```
<?xml version="1.0"?>
<cryptids>
  <species>
    <name>Jackalope</name>
    <habitat>Western North America</habitat>
    <description>
      <para>
        Similar to the Bavarian raurackl (stag-hare), the
        North American Jackalope resembles a large jackrabbit
        with small, deer-like antlers. This vicious
        carnivore is frequently mistaken for common rabbits or hares
        suffering from <italic>papillomatosis</italic> (a condition
        that produces horn-like growths on the head in those species).
      </para>
    </description>
  </species>
  <species>
    <name>Dahut</name>
    <habitat>French Alps</habitat>
    <description>
      <para>
        A shy relative of the Alpine deer, the Dahut has
        adapted to the challenges of its mountainous habitat by
        growing legs that are considerably longer on one side
        of its body. While this asymmetrical limb configuration allows
        for level grazing on steep grades, it leaves the unfortunate
        creature unable to reverse its course. Local hunters exploit
        this weakness by sneaking up behind the Dahut and either
        whistling softly or crying "Dahut!"; when the startled
        creature turns to face its assailant, it finds its
        longer legs on the wrong side and it tumbles to it doom.
      </para>
    </description>
  </species>
  <!--  . . . more species here -->
</cryptids>
```

No cryptozoology site worth its salt is complete without a list of cryptid sightings. Your second XML document, creatively named *cryptid_sightings.xml*, contains just that. (See Example 3-2.)

Example 3-2. cryptid_sightings.xml

```
<?xml version="1.0"?>
<sightings>
  <sighting>
    <species>Jersey Devil</species>
    <location>Bordentown, NJ</location>
    <date>Autumn, 1816</date>
    <description>
      <p>
        A Jersey Devil was reportedly seen
        by Joseph Bonaparte, former King of Spain
        and brother of Napoleon, while hunting in
        the woods near Bordentown, New Jersey.
      </p>
    </description>
    <witnesses>
      <name>Joseph Bonaparte</name>
    </witnesses>
  </sighting>
  <sighting>
    <species>Snipe</species>
    <location>Phelan, CA</location>
    <date>June 2002</date>
    <description>
      <p>
        The Phelan Phine Snipe Hunters Association
        celebrated the opening of this year's Snipe
        season. Unfortunately, the entire
        photographic record of the event was ruined
        during a nasty "keg stand" incident in
        the beer tent after the hunt.
      </p>
    </description>
    <witnesses>
      <name>Jason Nugall</name>
      <name>William Q. Rozborne</name>
    </witnesses>
  </sighting>
</sightings>
```

You need one last XML document: a small file that captures the filenames of the two other documents in the site. You will use this bit of metadata to create the navigation in the final result delivered to the requesting HTML browser, so the name *nav.xml* seems appropriate. (See Example 3-3.)

Example 3-3. nav.xml

```
<?xml version="1.0"?>
<links>
  <a href="cryptozoo.xml">Species</a>
  <a href="cryptid_sightings.xml">Sightings</a>
</links>
```

3.3 Writing the Stylesheet

So far, you have three XML documents that contain three very different, but randomly overlapping, grammars. (The species and name elements appear in different roles in the two main content documents.) Your goal is to make this information available on the Web to HTML browsers. You want to reach the widest possible audience, and that means maintaining the lowest possible expectations of the requesting client's capabilities. That is, you cannot rely on everyone who wants to read your pages having a thoroughly modern browser capable of doing appropriate client-side transformations to your XML documents via CSS or XSLT. You must deliver basic HTML if you expect your data to be widely accessible.

With this in mind, you need a way to transform the disparate data structures contained in each of your XML documents into the unified grammar of simple HTML.That's where AxKit's transformational languages and stylesheets enter the picture. AxKit offers many ways to transform XML data. (We will examine the merits of many of these in later chapters.) In this example, we examine how you can transform your cryptozoology documents into HTML using two of the more popular transformation languages: XSLT and XPathScript.

I will save the examination of the lower-level details of these languages for later. At this point, it suffices to understand that both XSLT and XPathScript offer a declarative syntax that provides a way to create new documents by applying transformations to all or some of the elements, attributes, and other content that an existing XML document contains.

3.3.1 Using XSLT

Rather than taking small steps through the XSLT stylesheet, I present it here in one block to give you an idea of what a full, working stylesheet looks like. (See Example 3-4.) Do not worry if much of it seems foreign; we will look at the syntactic elements of XSLT in more detail in Chapter 5.

As you read through the stylesheet, keep in mind that:

- An XSLT stylesheet itself is an XML document.
- Transformation rules are applied based on the properties of the various parts of the source XML document (element and names, relationships between elements, etc.).
- Template rules are created to match all elements of the grammars found in your XML documents, so the same stylesheet can be used to transform both the list of species and the list of sightings.

Example 3-4. cryptozoo.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  >
<xsl:output
     method="html" encoding="ISO-8859-1"
/>

<xsl:template match="/">
<html>
  <head><title>Cryptozoology Pages</title>
  <link rel="stylesheet" type="text/css" href="crypto.css"/>
  </head>
  <body>
  <div class="header">
  <h1>My Cryptozoology Pages</h1>
  </div>
  <div class="nav">
    <xsl:apply-templates select="document('nav.xml', /)/*"/>
  </div>
  <div class="content">
    <xsl:apply-templates/>
  </div>
  </body>
</html>
</xsl:template>

<!-- top-level templates -->
<xsl:template match="animals">
  <p>
    Today's rural legend is tomorrow's newly
    discovered species.
  </p>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="sightings">
  <p>
    Here is a list of sightings.
  </p>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="animals/species">
<div class="species">
  <xsl:apply-templates/>
</div>
</xsl:template>

<xsl:template match="sighting">
```

```
<div class="sighting">
  <xsl:apply-templates/>
</div>
</xsl:template>

<xsl:template match="species/name">
<h2>
  <xsl:apply-templates/>
</h2>
</xsl:template>

<xsl:template match="species/habitat">
  <p>
    <i>Habitat:</i>
    <xsl:text> </xsl:text>
    <xsl:value-of select="."/>
  </p>
  <xsl:apply-templates select="*"/>
</xsl:template>

<xsl:template match="sighting/location|sighting/species|sighting/date">
  <div class="subheading">
    <i><xsl:value-of select="name( )"/>:</i>
    <xsl:text> </xsl:text>
    <xsl:value-of select="."/>
  </div>
  <xsl:apply-templates select="*"/>
</xsl:template>

 <xsl:template match="witnesses">
   <div>
     <i>witnesses:</i>
     <xsl:text> </xsl:text>
     <xsl:for-each select="name">
       <xsl;value-of select="." />
       <xsl:if test="position( ) != last( )">, </xsl:if>
     </xsl:for-each>
   </div>
```

```
</xsl:template>


<xsl:template match="witnesses/name">

  <xsl:text> </xsl:text>

  <xsl:value-of select="."/>

</xsl:template>


<xsl:template match="para|p">

<p>

  <xsl:apply-templates/>

</p>

</xsl:template>


<xsl:template match="italic">

<i>

  <xsl:apply-templates/>

</i>

</xsl:template>


<xsl:template match="a">

  <xsl:copy-of select="."/><br/>

</xsl:template>


</xsl:stylesheet>
```

3.3.2 Using XPathScript

Compare the previous XSLT stylesheet, which transforms your cryptozoology documents into HTML, with the one below, written in XPathScript. I will leave the syntactic details for Chapter 6; however, as you read through the stylesheet, note that most moving parts are written in Perl and separated from the larger template document using `<%` and `%>` as delimiters.

```
<%

# declarative templates

$t->{'p'}{pre} = '<p>';

$t->{'p'}{post} = '</p>';


$t->{'para'}{pre} = '<p>';

$t->{'para'}{post} = '</p>';


$t->{'animals'}{pre} = qq|
  <p>
    Today's rural legend is tomorrow's newly
    discovered species.
  </p>
|;

$t->{'sightings'}{pre} = qq|
  <p>
    Here is a list of sightings.
  </p>
|;

$t->{'sighting'}{pre} = '<div class="sighting">';
$t->{'sighting'}{post} = '</div>';


$t->{'habitat'}{pre} =  '<p><i>Habitat:</i> ';
$t->{'habitat'}{post} =  '</p>';



$t->{'location'}{pre} = '<div class="subheading"><i>location: </i>';

$t->{'location'}{post} = '</div>';

$t->{'date'}{pre} = '<div class="subheading"><i>date: </i>';
$t->{'date'}{post} = '</div>';

$t->{'witnesses'}{pre} = '<div><i>witnesses: </i>';
$t->{'witnesses'}{post} = '</div>';

$t->{'a'}{showtag} = 1;

# testcode templates for like-named elements
$t->{'name'}{testcode} = sub {
  my ($node, $t) = @_;
  if (findnodes('parent::species', $node)) {
      $t->{pre} = '<h2>';
      $t->{post} = '</h2>';
  }
  return 1;
};

$t->{'species'}{testcode} = sub {
  my ($node, $t) = @_;
  if (findnodes('parent::animals', $node)) {
     $t->{pre} = '<div class="species">';
     $t->{post} = '</div>';
  }
  else {
    $t->{pre} = '<div class="subheading"><i>species: </i>';
    $t->{post} = '</div>';
  }
  return 1;
};
%>

<html>
  <head><title>Cryptozoology Pages</title></head>
  <link rel="stylesheet" type="text/css" media="screen" href="crypto.css"/>
  <body>
  <div class="header">
  <h1>My Cryptozoology Pages</h1>
  </div>
  <div class="nav">
    <%= apply_templates( "document('nav.xml')" ) %>
  </div>
  <div class="content">
    <%= apply_templates( ) %>
  </div>
  </body>
</html>
```

Do not be overwhelmed. Remember that most sites typically use *either* XSLT or XPathScript and only rarely both, so you need not try to take in both at once. These duplicated examples only intend to show that with AxKit, as with Perl, there is always more than one way to do it. You are free to choose the tools and techniques that suit you best.

3.4 Associating the Documents with the Stylesheet

AxKit offers a variety of configuration options for associating documents with its various language processors. Chapter 4 covers each in detail. In Example 3-5, you create an *.htaccess* file in the same directory as your XML documents. It defines a default style for AxKit to use when processing documents in this directory.

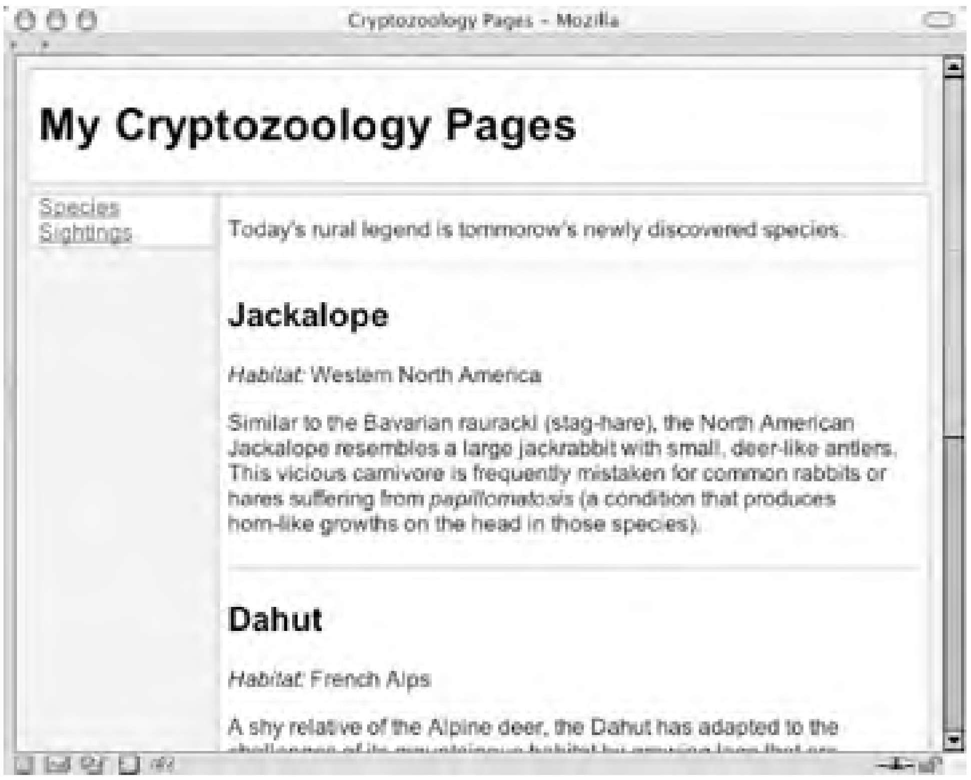
Example 3-5. A simple .htaccess file

```
<AxStyleName "#default">

  AxAddProcessor text/xsl stylesheets/cryptozoo.xsl

</AxStyleName>
```

Pay attention to the arguments passed to the AxAddProcessor directive. The first is the MIME type that AxKit examines to decide which language processor modules to use, and the second is the DocumentRoot-relative path to the stylesheet that will be passed to that language processor to transform your XML documents. If you want to use your XPathScript stylesheet rather than the XSLT, you would use AxAddProcessor application/x-xpathscript stylesheets/cryptozoo.xps instead. This processor definition is wrapped in an AxStyleName block. This directive block, in turn, combines the processor definitions it contains into a single "named style" that a StyleChooser or other plug-in can select at runtime. By giving this style the special name #default, you are configuring AxKit to use this style as a fallback if no other style is explicitly selected.

It's time to fire up a web browser and check the results of your work. A request to http://myhost.tld/cryptozoo.xml yields what is shown in Figure 3-1.

Figure 3-1. cryptozoo.xml rendered as HTML



My Cryptozoology Pages

Today's rural legend is tommorow's newly discovered species.

Jackalope

Habitat: Western North America

Similar to the Bavarian rauracki (stag-hare), the North American Jackalope resembles a large jackrabbit with small, deer-like antlers. This vicious carnivore is frequently mistaken for common rabbits or hares suffering from *papillomatosis* (a condition that produces horn-like growths on the head in those species).

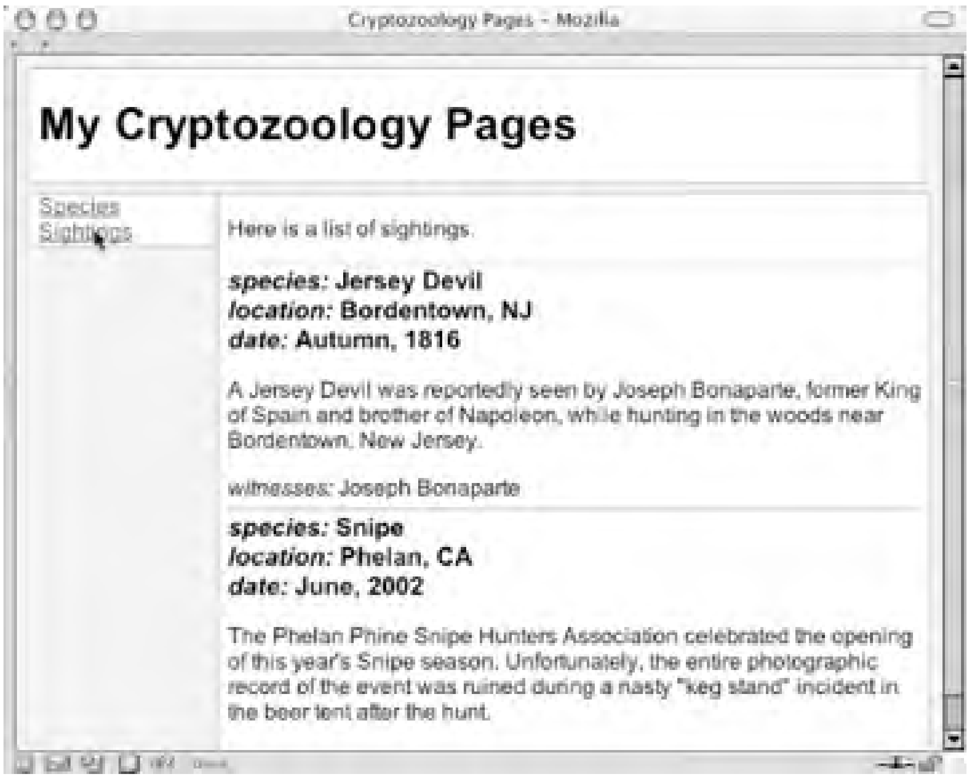
Dahut

Habitat: French Alps

A shy relative of the Alpine deer, the Dahut has adapted to the

Clicking on the Sightings link reveals what is shown in Figure 3-2.

Figure 3-2. cryptid_sightings.xml rendered as HTML



My Cryptozoology Pages

Here is a list of sightings.

***species:* Jersey Devil**
***location:* Bordentown, NJ**
***date:* Autumn, 1816**

A Jersey Devil was reportedly seen by Joseph Bonaparte, former King of Spain and brother of Napoleon, while hunting in the woods near Bordentown, New Jersey.

witnesses: Joseph Bonaparte

***species:* Snipe**
***location:* Phelan, CA**
***date:* June, 2002**

The Phelan Phine Snipe Hunters Association celebrated the opening of this year's Snipe season. Unfortunately, the entire photographic record of the event was ruined during a nasty "keg stand" incident in the beer tent after the hunt.

3.5 A Step Further: Syndicating Content

You have reached your initial goal of publishing your XML documents for consumption by HTML browsers on the Web using AxKit. Even if that were all you ever wanted to do, you still made a clear division between the content you will maintain and the way in which it is presented. Among other benefits, you can now redesign the look and feel of pages sent to the client without touching content documents. Don't worry about clobbering or obscuring essential information just to change the way it renders in a browser. Similarly, using a custom XML grammar for your content means that the documents themselves can unambiguously define the intended roles of the data they contain, rather than the way that data may be represented on the visual medium of an HTML browser. This makes reusing the data for other purposes a lot easier.

To understand the practical benefits of separating content from presentation, suppose that your list of cryptid sightings becomes wildly popular on the Web. People start asking for a way to put links to the newly reported sighting on their own cryptozoology sites. You could tell them to screen-scrape the HTML list. Instead, you decide to be a good information-sharing citizen and make the list available as an RSS syndication feed. To achieve this, the first thing you need is a stylesheet that transforms the list of cryptid sightings to RSS, in addition to the one you already have that transforms the data into HTML.

For those who may not be familiar with it, RSS (RDF Site Summary, Rich Site Summary, or Really Simple Syndication, depending on whom you ask) is a popular XML grammar used for syndicating online content, especially news headlines. Most weblogs use RSS as the means to both publish content and share links with other bloggers, and many weblog tools store their data natively as RSS. (See Example 3-6.) For more information about RSS and some of its more creative uses, see Ben Hammersley's *Content Syndication with RSS* (O'Reilly).

Example 3-6. cryptidsightings_rss.xsl

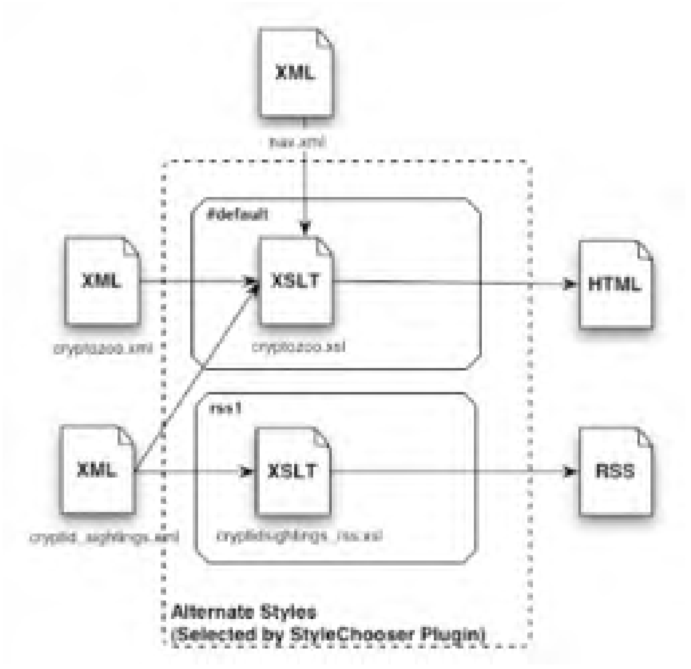
```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://purl.org/rss/1.0/"
  version="1.0"
>

<xsl:variable
     name="this_url"
     select="'http://myhost.tld/cryptosightings.xml'"
/>

<xsl:template match="/">
<rdf:RDF>
  <channel rdf:about="{$this_url}">
    <title>My Cryptozoology Pages- Sightings</title>
    <link>
      <xsl:value-of select="$this_url"/>
    </link>
    <description>
      The latest sightings of animals that do not officially exist.
    </description>

    <items>

      <rdf:Seq>

        <xsl:for-each select="/sightings/sighting">

          <rdf:li rdf:resource="{$this_url}#{position( )}"/>

        </xsl:for-each>

      </rdf:Seq>


    </items>

  </channel>

  <xsl:apply-templates select="/sightings/sighting"/>

</rdf:RDF>

</xsl:template>


<xsl:template match="sighting">

<item rdf:about="{$this_url}#{position( )}">

  <link>

    <xsl:value-of select="concat($this_url,'#', position( ))"/>

  </link>


  <title>

    <xsl:value-of select="species"/> Sighting - <xsl:value-of select="date"/>

  </title>

  <description>

    <xsl:apply-templates select="description"/>

  </description>

</item>

</xsl:template>


<xsl:template match="description|p">

  <xsl:value-of select="normalize-space(.)" />

</xsl:template>


</xsl:stylesheet>
```

Before you can see this stylesheet in action, you need to add a couple of configuration directives to the *.htaccess* file you created earlier:

```
<Files cryptid_sightings.xml>

  <AxStyleName rss1>

    AxAddProcessor text/xsl stylesheets/cryptidsightings_rss.xsl

  </AxStyleName>

  AxAddPlugin Apache::AxKit::StyleChooser::QueryString

</Files>
```

The AxStyleName block creates a named style called rss1. The AxAddProcessor it contains associates that named style with the RSS stylesheet you just created. The AxAddPlugin directive, in this case, tells AxKit to use additional logic to examine the query string sent from the requesting client for a parameter named style. If it finds one, and the value of that parameter matches one of the named styles configured for that URI, it uses the stylesheets contained in that named style to transform the XML source document. Here, this means that a request to http://myhost.tld/cryptid_sightings.xml?style=rss1 returns the list of species sightings processed by your RSS output stylesheet, not the default style you created earlier. (See Figure 3-3.)

Figure 3-3. Cryptozoology site-processing diagram



Example 3-7 shows the result for a request for the *cryptid_sightings.xml* file as an RSS document.

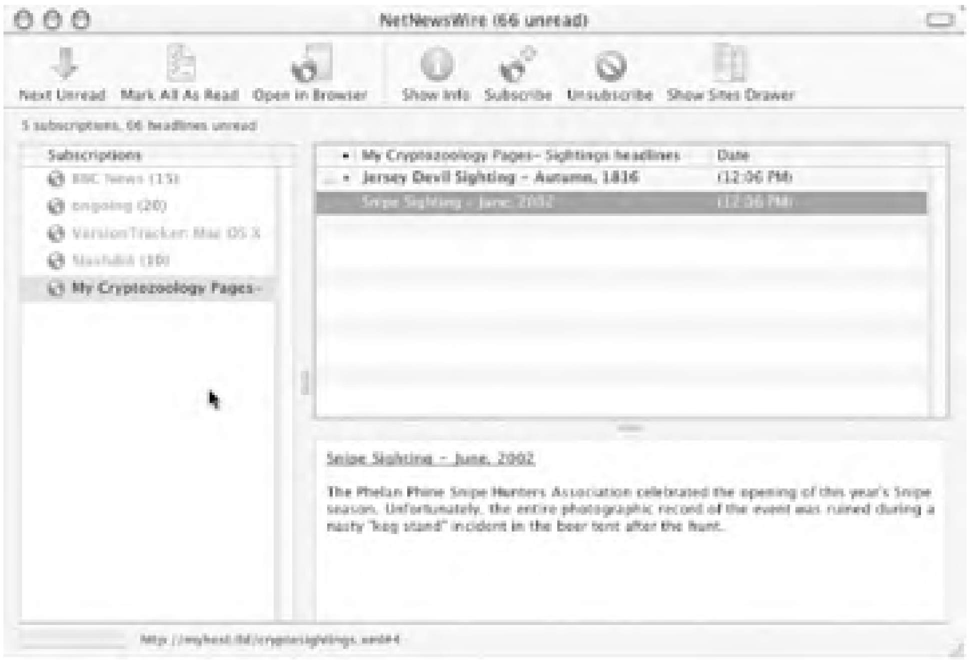
Example 3-7. Cryptid sightings delivered as an RSS feed

```
<?xml version="1.0"?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

xmlns="http://purl.org/rss/1.0/">

  <channel rdf:about="http://myhost.tld/cryptosightings.xml">

    <title>My Cryptozoology Pages- Sightings</title>

    <link>http://myhost.tld/cryptosightings.xml</link>

    <description>
```

```
      The latest sightings of animals that do not officially exist.
    </description>
    <items>
      <rdf:Seq>
        <rdf:li rdf:resource="http://myhost.tld/cryptosightings.xml#1"/>
        <rdf:li rdf:resource="http://myhost.tld/cryptosightings.xml#2"/>
      </rdf:Seq>
    </items>
  </channel>
  <item rdf:about="http://myhost.tld/cryptosightings.xml#1">
    <link>http://myhost.tld/cryptosightings.xml#1</link>
    <title>Jersey Devil Sighting - Autumn, 1816</title>
    <description>
        A Jersey Devil was reportedly seen
        by Joseph Bonaparte, former King of Spain
        and brother of Napoleon, while hunting in
        the woods near Bordentown, New Jersey.
      </description>
  </item>
  <item rdf:about="http://myhost.tld/cryptosightings.xml#2">
    <link>http://myhost.tld/cryptosightings.xml#2</link>
    <title>Snipe Sighting - June, 2002</title>
    <description>
        The Phelan Phine Snipe Hunters Association
        celebrated the opening of this year's Snipe
        season. Unfortunately, the entire
        photographic record of the event was ruined
        during a nasty "keg stand" incident in
        the beer tent after the hunt.
      </description>
  </item>
</rdf:RDF>
```

Figure 3-4 shows a request to that same URL rendered in the NetNewsWire RSS client.

Figure 3-4. Information rendered as RSS in NetNewsWire



Snipe Sighting - June, 2002

The Phelan Phine Snipe Hunters Association celebrated the opening of this year's Snipe season. Unfortunately, the entire photographic record of the event was ruined during a nasty "keg stand" incident in the beer tent after the hunt.

Now your friends in the cryptozoology community can add your list of sightings to their list of daily news feeds by pointing their RSS viewer or other client to http://myhost.tld/cryptid_sightings.xml?style=rss1, while casual web surfers get the default HTML version. Keep in mind, too, that the query string StyleChooser is only one way to dynamically select the preferred style. Using other plug-ins, you could just as easily examine the connecting client's User-Agent header, or another aspect of the request, to get the same effect. Remember, too, that XSLT and XPathScript are only two transformative Language modules that AxKit supports; there are several others, each with its own unique strengths and weaknesses.

As promised, your first example site is a bit silly (Dahuts, indeed), but let's examine exactly what you have done:

- You used stylesheets to transform data marked up in custom XML grammars into a commonly used delivery format (HTML).
- You combined XML from different resources (the content documents and the navigation metadata) to meet your application's needs.
- You mixed standard Apache configuration blocks with AxKit's custom directives to select an alternative style transformation that delivered the same XML content in a different format in response to data received from the requesting client.

Taken together, these points represent the basic foundation of publishing XML documents with AxKit. You can apply the patterns you learned here to most, if not all, of your future XML publishing projects. However, your little cryptids' site offers only a glimpse of the flexibility and power that AxKit offers. The following chapters build on these basic concepts to show how you can use AxKit to create sophisticated, dynamic, and feature-rich sites.

Chapter 3. Your First XML Web Site

With AxKit installed, you can begin putting it though its paces. In this chapter, we create a simple XML-based web site. Along the way, I will introduce AxKit's facilities for how to apply stylesheets to transform data marked up in XML into a commonly used delivery format, how to combine XML from different sources, and how to configure an alternate style transformation to deliver the same XML content in a different format in response to data received from the requesting client.

4.1 Adding Transformation Language Modules

Before you can begin associating documents with the stylesheets that will be used to transform them, you must tell AxKit which lower-level processors to use to perform those transformations. In AxKit, access to various transformative processors is provided by its Language modules. Many of these modules create a bridge between AxKit and an existing XML processing tool. For example, the Apache::AxKit::Language::LibXSLT module allows AxKit access to Perl's XML::LibXSLT interface and, hence, to the Gnome project's XSLT processing library, `libxslt`. Other Language modules, such as AxKit's implementation of eXtensible Server Pages, ApacheAxKit::Language::XSP, are unique to AxKit and implement both the interface that allows it to be added to the AxKit processing chain and the code that actually processes the XML content. The core AxKit distribution contains several such Language modules:

Apache::AxKit::Language::LibXSLT

Adds support for the Gnome Project's `libxslt` processor; used to transform documents with XSLT

Apache::AxKit::Language::Sablot

An alternate XSLT transformer using the Sablotron XSLT processor from The Ginger Alliance

Apache::AxKit::Language::XPathScript

Adds support for a more Perlish alternative to XSLT, XPathScript

Apache::AxKit::Language::XSP

Provides an interface to AxKit's implementation of the eXtensible Server Pages (XSP)

Apache::AxKit::Language::SAXMachines

Provides an AxKit interface to Barrie Slaymaker's popular XML::SAX::Machines Perl module, which offers an easy way to set up chains of SAX Filters to transform XML content

Apache::AxKit::Language::PassiveTeX

Offers XSL-FO (XSL Formatting Objects) support for generating PDF documents from FO sources via PassiveTeX suite's `pdfxmltex` utility

Apache::AxKitLanguage::HtmlDoc

Converts XHTML documents to PDF, also using PassiveTeX

Several other Language modules come with AxKit, and still others are available via the Comprehensive Perl Archive Network (CPAN). For example, many of Perl's more popular templating packages (the Template Toolkit, Petal, etc.) are also available as AxKit Language processors. However, while AxKit or an individual CPAN distribution may provide the Language modules needed to allow AxKit to access a given type of processor, they do not usually install any lower-level tools with which the modules may interface. For example, if you intend to use the Apache::AxKit::LibXSLT module for XSLT processing, you must be sure to first install the XML::LibXML and XML::LibXSLT Perl modules as well as the `libxml2` and `libxslt` C libraries that it requires.

Once you decide which kinds of transformations to use for your site, the Language module is added to AxKit via the `AxAddStyleMap` directive. This directive expects two arguments: a MIME type that uniquely identifies the language, and the Perl package name of the module that implements that language's processor:

```
<AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT
```

This associates the MIME type `text/xsl` with the Perl package name of the LibXSLT Language module, Apache::AxKit::Language::LibXSLT. This mapping is important since the MIME type assigned here is used as the identifier to associate the site's stylesheets with the appropriate language processor that will be used to apply those stylesheets to the source content. So, given the above example, all style definitions that declare the type `text/xsl` will be applied to the source XML using the Apache::AxKit::Language::LibXSLT language module.

Although AxAddStyleMap directives may appear anywhere in a site's configuration files, these type-to-module mappings are typically meant to apply to a whole site and, therefore, usually appear together at the top level of the host's AxKit configuration:

```
<Directory "/www/sites/myaxkitsite">

  AddHandler axkit .xml .xsp. .xsp .dbk]]>

  <emphasis role="bold">AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT

  AxAddStyleMap application/x-xpathscript Apache::AxKit::Language::XPathScript

  AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP</emphasis>

</Directory>
```

Once the language processors are set up, the style definitions that will control the transformations that AxKit will apply to the content can be added.

4.2 Defining Style Processors

The most basic component used to define which transformations will be applied to a given resource within AxKit is perhaps best termed a *style processor definition*. These definitions indicate a single transformational step (applying an XSLT stylesheet or passing the content through a SAX Filter, for example) in what may be a chain of transformations. In their most basic and typical form, these style definitions declare two bits of crucial information: a MIME type that will be used by AxKit to determine which Language module will be used to transform the content, and a file path to a stylesheet (or other Language-specific argument) that will be used by that Language module to determine how to transform the content. Individual processor definitions may optionally be combined into named style and media groups that can then be selected conditionally, based on a number of factors.

By default, style processors are defined within AxKit in one of two ways: by using AxKit's processor configuration directives or by special stylesheet processing instructions contained in the source documents themselves. (In Chapter 8, you'll learn to create your own way to configure AxKit's styling rules by rolling a custom ConfigReader module, but that's another story.) The following illustrates how to create a simple named style containing a single style definition using AxKit's server configuration directives. The lone processor definition contains the required MIME type and path to the stylesheet that will be applied.

```
<AxStyleName "#default">

  AxAddProcessor text/xsl /path/to/style1.xsl

</AxStyleName>
```

The MIME type used in the processor definition corresponds to the one you associated earlier with the Language::LibXSLT module. The effect of this is that, starting with the source, XML will be transformed by the LibXSLT processor by applying the stylesheet at the location defined by the second argument.

4.2.1 AxKit's Runtime Styling Directives

AxKit offers a small host of runtime configuration directives that can be used to define the styles for a given site. These directives are actually extensions of Apache's configuration syntax and, as such, are added to the usual server configuration files, such as *httpd.conf* or *.htaccess* files. They can be mixed and matched with other Apache directives. A basic familiarity with Apache's runtime configuration syntax is presumed in this section.

Using AxKit's configuration directives, style definitions can be added and applied in all cases or applied conditionally, based on an aspect of the XML document being served (the document's root element name, URI on the server, etc.). A style definition's first argument is the MIME type associated with the language module that will apply the transformation, and the second is the path to the stylesheet that will be applied. Conditional processor definitions add a third, directive-specific argument that defines the rules that govern when and if that processor will be adding to the current processing chain.

Relative stylesheet paths are resolved in the context of the directory that contains the resource being requested, while qualified paths are resolved relative to the host's DocumentRoot. However, not all language modules employ external stylesheets. For example, the SAXMachines module expects a quoted, whitespace-separated list of Perl package names that implement SAX Filters instead of a stylesheet path. Others, such as XSP, are self-contained (the source document itself defines the transformations), and the literal string NULL is used in place of the stylesheet path.

We will now take a look at each of AxKit's processor definition configuration directives. Keep in mind as you read the details of each directive that these are just the lowest-level building blocks from which you can create sophisticated application- and media-specific processing chains.

4.2.1.1 AxAddProcessor

AxAddProcessor, the most basic of the styling configuration directives, unconditionally adds that processor definition to AxKit's processing chain. The processor definition requires two arguments: the MIME type associated with the language processor that will be used to do the transformation, and the path to the stylesheet that will be used by that processor to transform the XML source.

```
AxAddProcessor text/xsl /styles/global.xsl
```

This unconditionally adds the stylesheet */styles/global.xsl* to the processing chain and tells AxKit to use the language module associated with the MIME type text/xsl to perform the transformation.

The AxAddProcessor directive is commonly used in conjunction with Apache's standard <Files>, <Directory>, <Location>, and similar directives. For example, adding the following to a *.htaccess* at the root level of a site would configure AxKit to apply the stylesheet */styles/docbook_simple.xsl* to all documents in that site that have the file extension *.dkb,* while transforming all documents with *.xml* extension with the global stylesheet from the above example:

```
<Files *.dkb>

  AxAddProcessor text/xsl /styles/docbook_simple.xsl

</Files>



<Files *.xml>

  AxAddProcessor text/xsl /styles/global.xsl

</Files>
```

Transformation chains can be created by adding more than one style processor definition to a given context. The following would configure AxKit to apply the stylesheet *pre_process.xsl* to all documents in the /docs directory and then to apply the stylesheet *main.xsl* to the result of the first transformation:

```
<Directory /docs>

  AxAddProcessor text/xsl /styles/pre_process.xsl

  AxAddProcessor text/xsl /styles/main.xsl

</Directory>
```

AxKit processing definitions, like many Apache directives, are inherited recursively down directory branches. So, a style definition configured for the root of a site will be applied to all documents in that site, unless explicitly overridden.

4.2.1.2 AxAddRootProcessor

The AxAddRootProcessor directive sets up document-to-stylesheet mappings based on the name of the root element in the source XML document. It requires three arguments: the MIME type associated with the language processor that will perform the transformation, the path to the stylesheet that will be applied, and the name of the root element to match against.

The following configures AxKit to apply the stylesheet */styles/docbook_simple.xsl* to all documents in the current scope that have a root element named article:

```
AxAddRootProcessor text/xsl /styles/docbook_simple.xsl article
```

Matching top-level element names in documents employing XML namespaces is supported using the Clarkian Notation, in which the element's namespace URI is wrapped in curly braces { } and prepended to the element's local name.

```
AxAddRootProcessor text/xsl /styles/mydoc.xsl {http://localhost/NS/mydoc}document
```

This would apply the stylesheet */styles/mydoc.xsl* to both of the following documents:

```
<?xml version="1.0"?>

<document xmlns="http://localhost/NS/mydoc">

  <element/>

</document>



<?xml version="1.0"?>

<doc:document xmlns:doc="http://localhost/NS/mydoc">

  <doc:element/>
```

```
</doc:document>
```

The element's namespace prefix is not considered, since it is really only a syntactic shortcut used to bind the given element to the namespace URI to which the prefix is bound. Hence, the following would match *neither* of the above documents:

```
AxAddRootProcessor text/xsl /styles/mydoc.xsl {http://localhost/NS/mydoc}

doc:document
```

4.2.1.3 AxAddDocTypeProcessor and AxAddDTDProcessor

These directives allow for stylesheet selection based on aspects of the source content's Document Type Definition (DTD).

The AxAddDocTypeProcessor directive conditionally adds processors based on the value assigned to the PUBLIC identifier contained in the source document's DTD. So, to add a style processor to match the following Simplified DocBook document, do the following:

```
<?xml version="1.0"?>

<!DOCTYPE article

      PUBLIC "-//OASIS//DTD Simplified DocBook XML V4.1.2.5//EN"

      "http://www.oasis-open.org/docbook/xml/simple/4.1.2.5/sdocbook.dtd" [  ]>

<article>

 . . .

</article>
```

You can use the following AxAddDocTypeProcessor directive:

```
AxAddDocTypeProcessor text/xsl /styles/sdocbook.xsl "-//OASIS//DTD Simplified

DocBook XML V4.1.2.5//EN"
```

Similarly, the AxAddDTDProcessor directive conditionally adds processors based on the path contained in the SYSTEM identifier in the source document's DTD:

```
<?xml version="1.0"?>

<!DOCTYPE mydoc

      SYSTEM "/path/to/my.dtd" [  ]>

<article>

 . . .

</article>
```

and the matching AxAddDTDProcessor directive:

```
AxAddDTDProcessor text/xsl /styles/sdocbook.xsl /path/to/my.dtd
```

It is important to remember that *only* the original source document is examined for matches against the conditional AxAddRootProcessor, AxAddDTDProcessor, and AxAddDocTypeProcessor directives. So, for example, if you alter the document's root element name during a transformation by one stylesheet, that new root element will *not* be evaluated against any AxAddRootProcessor directives that may exist in the current context.

4.2.1.4 AxAddURIProcessor

The AxAddURIProcessor provides a way to apply styles by matching a Perl regular expression against the current request URI. Mostly, this directive is a useful way to emulate LocationMatch blocks in contexts in which those blocks are not allowed.

```
# In httpd.conf, fine here

<LocationMatch "/my/virtual/uri">

  AxAddProcessor text/xsl /styles/application.xsl

</LocationMatch>



# But I like .htaccess files and can't use LocationMatch!!

AxAddURIProcessor text/xsl /styles/application.xsl "/my/virtual/uri"
```

4.2.1.5 AxAddDynamicProcessor

AxAddDynamicProcessor, the syntactic oddball among the processor directives, accepts the name of a Perl package as its sole argument. When this directive is found in a given context, AxKit calls the *handler( )* subroutine in the package specified. That function is expected to return a list of processor definitions that will be added to the current processing chain.

The *handler( )* subroutine is passed, in order: an instance of the current ContentProvider class, the preferred media name, the preferred style name, the source content's DTD PUBLIC identifier, its SYSTEM identifier, and root element name. If the source content does not contain a DTD, or a preferred media and style that have not been set by an upstream plug-in for the current request, those corresponding arguments will not be defined. The styles returned take the form of a list of HASH references, each containing two required key/value pairs: href, whose value should contain the path to the stylesheet being applied; and type, whose value should contain the MIME associated with the Language module that will do the transformation.

```
package My::Processors;

sub handler {

    my ($provider, $preferred_media_name, $preferred_style_name, $doctype,

$dtd, $root) = @_;



    # normally, @styles will be generated dynamically rather than hardcoded, as

    # it is here.



    my @styles = (

        { type => 'application/x-xsp', href => 'NULL' },

        { type => 'text/xsl', href => '/styles/mystyle.xsl' },

    );



    return @styles;

}
```

4.2.1.6 AxResetProcessors

The AxResetProcessors directive clears the list of processor mapping within the scope of its surrounding block. This is especially useful for handling special cases in otherwise homogeneous configurations.

```
# Set the default style for the entire site

<Directory "/www/sites/mysite">

  AxAddProcessor text/xsl /styles/global.xsl

</Directory>



# But you need to transform the content in the 'products'

# directory with a different style and you don't want to

# inherit the global style.

<Directory "/www/sites/mysite/products">

  AxResetProcessors

  AxAddProcessor text/xsl /styles/products.xsl

</Directory>
```

4.2.1.7 Style definition directive inheritance

Like most Apache configuration directives, AxKit style processor definitions are inherited recursively down directories. A style defined at the root level of the site, for example, will be applied to all documents in or below that directory in the hierarchy, and, hence, the entire site. Styles added to locations deeper in the hierarchy do not override the styles defined by their parents but rather are *added* to the processing chain. So, for instance, if you have a global style defined for the root level of the site, and one defined for a child directory named contact, all documents in the contact directory will have both styles applied during processing. What surprises many new AxKit users, however, is the order in which the styles are applied in these cases—styles defined in child directories are *prepended* to the list of processors defined by their parents, not appended, as you may initially expect. Consider the following style configuration:

```
<Directory "/www/sites/mysite">

  AxAddProcessor text/xsl /styles/global.xsl

</Directory>



<Directory "/www/sites/mysite/contact">

  AxAddProcessor application/x-xsp NULL

</Directory>
```

The *mysite* directory has a style definition that applies the *global.xsl* XSLT stylesheet to all of its contents, and that the *.contact* directory (a child of the mysite directory) adds a style definition that configures AxKit to process contents of that directory with the XSP processor. With this configuration, a document requested from the *contact* directory will have both styles applied, but the local XSP process will be applied *first* and the result of that will be processed using the *global.xsl* stylesheet. While this behavior seems a bit strange at first glance, in practice it helps simplify setting up the Style processing for many sites where a common look and feel is applied to all content and various source-specific transformations are configured for lower levels in the document hierarchy. (The results are often copied through *verbatim* by the higher level transformations.)

Taken together, these configuration directives represent a rich set of options that, when combined with Apache's standard configuration blocks, can meet the styling requirements of a great many sites. However, in Section 4.3, you will see how these components can be combined with AxKit's StyleChooser and media chooser modules to make your sites even more dynamic and responsive.

4.2.2 Stylesheet Processing Instructions

In addition to defining style processor mappings through the configuration directives, stylesheets can also be applied based on one or more xml-stylesheet processing instructions in the source document itself. Stylesheet processing instructions must appear in the prolog of the document—that is, between the XML declaration and the top-level document element:

```
<?xml version="1.0"?>]]><emphasis role="bold"><![CDATA[

<?xml-stylesheet type="text/xsl" href="/styles/docbook_simple.xsl" ?>]]></emphasis><![CDATA[

<article>

  . . .

</article>
```

Similar in behavior to the AddAddProcessor configuration directive, the xml-stylesheet processing instruction's type attribute should contain the MIME type associated with the language module that will perform the transformation. The href attribute should contain the path to the stylesheet that will be applied.

Styles added via xml-stylesheet processing instructions come in three flavors: persistent styles that are applied in all cases, preferred styles that define a default style if none is specifically selected, and alternate styles that are only applied under specific conditions.

Persistent styles are defined as those whose processing instruction has neither a title nor an alternate attribute. The following adds two persistent styles to the document that contains them:

```
<?xml-stylesheet type="text/xsl" href="/styles/nav_includes.xsl"?>

<?xml-stylesheet type="text/xsl" href="/styles/html.xsl"?>
```

All relevant styles in a given document are applied in the order in which their xml-stylesheet processing instructions appear. So, the above would tell AxKit to apply the */styles/nav_includes.xsl* stylesheet to the source XML and then apply the */styles/html.xsl* stylesheet to the result of the first transformation.

A preferred style is one whose processing instruction contains a title but not an alternate attribute. It is used to define the nonpersistent default style among a set of alternate styles:

```
<?xml-stylesheet type="text/xsl" href="/styles/html.xsl"

title="html"?>
```

An alternate style is one whose processing instruction contains *both* a title and an alternate attribute:

```
<![CDATA[

<?xml-stylesheet type="text/xsl" href="/styles/html.xsl" title="html"

alternate="yes"

]]>
```

In AxKit, alternate styles are used together with a StyleChooser module to select the appropriate styles to apply for a given circumstance. (See Section 4.3.1 later in this chapter.)

Finally, styles set via xml-stylesheet processing instruction can associate themselves with a specific media type by adding the media attribute. This allows alternate styles to be selected based on the type of device requesting the resource. (See Section 4.3.1.)

```
<?xml-stylesheet type="text/xsl" href="/styles/wap.xsl"

alternate="yes" media="handheld"?>
```

Stylesheet processing instructions are extracted *only* by the ContentProvider module that fetches the original source XML document. This means that you cannot add stylesheets to the processing chain by including xml-stylesheet instructions in the output of a stylesheet transformation. Once the document is sent to AxKit's language modules to process, the stylesheet PIs (if any) have already been extracted, and the results of the transformations are not reexamined.

4.2.2.1 AxIgnoreStylesheetPI

Any styles defined by xml-stylesheet processing instructions in the source document *override* all those defined via configuration directive. This is often desirable. For example, it provides applications that return dynamic documents through a custom ContentProvider, an easy way to set the styles for that application state while still having defaults set via configuration directive to cover common cases. In many situations, however, this is not the behavior that you want. To disregard all styles defined by xml-stylesheet processing instructions in a particular context, use the AxIgnoreStylesheetPI directive:

```
AxIgnoreStylesheetPI On
```

So, which is better, styles defined by configuration directive or by xml-stylesheet processing instruction? The answer depends largely on the site's requirements, but, in general, sites that define style processors via configuration directive tend to be more flexible and easier to maintain in the long run. For example, altering the stylesheet path for a given style is a one-line change for a site that uses configuration directive to define its styles, while the path would have to be altered in *each document* that references that style in the same site that relies on xml-stylesheet processing instructions.

PREV < Day Day Up > NEXT

4.3 Dynamically Choosing Style Transformations

The options that you've looked at so far for associating documents with transformative processes are quite capable. Often, the use of stylesheet processing instructions, or simple AxKit processor definitions, combined with constraints imposed by Apache built-in <Files>, <Directory>, <Location>, and similar block-level directives, are all you need to meet the needs of many sites. However, AxKit offers even more flexibility by providing additional mechanisms that allow you to combine these low-level style processing options into logical groups that can be selected at runtime. In this section, I introduce the concepts and syntax for creating named styles and media types and explain how these can be used in conjunction with the StyleChooser and MediaChooser modules to apply just the right content transformations under the right circumstances.

The reasons for using named style and media blocks are quite varied. Generally, they are best suited for cases when you need to select a transformation (or a chain of transformations) based on a condition *external* to the properties of the source XML content itself. Some reasons to use named styles and media blocks include:

Vendor branding

Your site offers a service, and each customer wants the content presented in a way that matches his unique look and feel.

User-selected skinning

One size never fits all. You want to offer your visitors the ability to select the style that suits them best.

Automated metadata extraction

You want to extract important metadata, like abstract summaries, author, title, copyright information, etc., by simply applying an alternative set of styles to your documents.

Role-specific data transformations

Your customers, vendors, shipping department, and company president all have different needs when they look at your product list. You want to serve all of them from the same XML data source.

Application state-specific transformations

You want to apply different transformations to your XML data to reflect the current state (or screen) of your online applications.

Any or all of the above

You want to incorporate any or all of the features listed, *plus* the ability to provide each option across a series of different client devices (phone, desktop browser, TV, etc.).

AxKit's named styles and media blocks, in conjunction with StyleChooser and MediaChooser plug-in modules, provide absolute control over when and how your XML content gets transformed. Literally, any condition that can be determined via the Perl programming language can be used to regulate which style processors will be applied to your content. The next two sections examine both named styles and named media, how they can be used together, and how AxKit's plug-ins can be used to select just the right combination.

4.3.1 Named Styles and StyleChoosers

A *named style* consists of one or more style processor definitions grouped together and given a unique name. The name given is used in conjunction with a StyleChooser or other AxKit plug-in to select the processors to apply in response to a given request. These modules set AxKit's internal preferred_style property based on a condition; then, if the name contained in that property matches the name given a named style block within the scope of the current request, all styles associated with that name are applied to the source document.

Named styles are created in one of two ways: by using the <AxStyleName> configuration directive or by adding the optional title and alternate attributes to an xml-stylesheet processing instruction in the source XML document. When using directive-based named styles, the <AxStyleName> configuration block acts as a container of one or more styling directives and uniquely associates that set of processors with the given name.

Suppose that you have an XML document that contains detailed data about the list of products offered by an online shop. You want to present a page typical for this kind of application: a list of all products, and links to more detailed information for each. Now, you could use an offline transformation to carve up the larger document into a set of smaller ones, then associate different style processors based on the files' location on the disk—one for the list view, and one for the detail views. The weakness of that approach, however, is that you must remember to rerun the process every time the product list is updated. AxKit's named styles provide another option: you can create two named styles that can be applied to the same global list of products—one presents the full list with minimal detail, and one reveals additional details for a specific product. The directives used to set up these named alternate views may look like this:

```
<Files products.xml>
  <AxStyleName
              list_view
    AxAddProcessor text/xsl /styles/list.xsl
  </AxStyleName>

  <AxStyleName detail_view
    AxAddProcessor text/xsl /styles/detail.xsl
  </AxStyleName>
</Files>
```

This creates two named styles that AxKit can apply conditionally when serving the *products.xml* document: the list_view style, which transforms the content into the complete list of products, and the detail_view style, which selects a single record from the global list and shows more detailed information about that specific item. The same can be achieved when using xml-stylesheet processing instructions by including title and alternate attributes with the appropriate values:

```
<?xml-stylesheet type="text/xsl" href="/styles/list.xsl"
              title="list_view"  alternate="yes"?>
<?xml-stylesheet type="text/xsl" href="/styles/detail.xsl"
title="detail_view" alternate="yes"?>
```

You can now use a StyleChooser plug-in to select between the two styles. AxKit ships with a number of default StyleChooser modules that set the preferred style based on commonly used conditions such as the name of the requesting user agent, a specific query string parameter, or the value of an HTTP cookie. These modules are added to AxKit via the AxAddPlugin configuration directive. So, for example, if you want to choose between the list view and detail view of your products document, based on the query string sent by the browser, you add the following to your configuration file:

```
 AxAddPlugin Apache::AxKit::StyleChooser::QueryString
```

With this in place, you can now easily select the different views of your products document by adding a style parameter whose value matches one of your named styles to the query string of a request to *products.xml*:

```
http://myhost.tld/products.xml?style=detail_view
```

When using named styles, it is important to set up a default style to cover cases in which the StyleChoosers may not set a preferred style or in which the style name returned does not match any style configured for the current document. This can be done by adding a style with the name #default or by using the AxStyle directive to set an existing named style as the default style:

```
# Fall back to a default style that does not match any
# other named style
<AxStyleName
              "#default"
```

```
   AxAddProcessor test/xsl /styles/list.xsl

</AxStyleName>



# Does the same, but uses the existing 'list_view' style

# as the default style.

<AxStyleName

                 list_view

  AxAddProcessor test/xsl /styles/list.xsl

 </AxStyleName>



AxStyle list_view
```

You can achieve the same using xml-stylesheet processing instructions by defining a preferred style (one that has a title attribute but not an alternate attribute with the value of yes):

```
<?xml-stylesheet type="text/xsl" href="/styles/product_list.xsl"

title="product_list"?>

<?xml-stylesheet type="text/xsl" href="/styles/product_detail.xsl"

title="product_detail" alternate="yes"?>
```

Another common use for named styles is user-defined "skinning" of a site's content. Recall your cryptozoology site from Chapter 3. Imagine that this site has become the primary resource on the Web for that topic. Both serious researchers in the area, as well as the hyper-ironic, slumming for a kitschy thrill, frequent it. Both groups share an interest in the site's content, but their expectations and preferences about *how* that content is best presented are likely to be quite different. To meet this need, you can create a new, flashier stylesheet to render the content for your hipster visitors and then add a named style to your configuration file that reflects that preference:

```
# the existing, plain style

<AxStyleName plain>

  AxAddProcessor text/xsl stylesheets/cryptozoo.xsl

</AxStyleName>



# provides the RSS 1.0 news feed

# view of the sightings

<AxStyleName rss1>

  AxAddProcessor text/xsl stylesheets/cryptidsightings_rss.xsl

</AxStyleName>



# the new, flashy style

<AxStyleName flashy>

  AxAddProcessor text/xsl stylesheets/cryptozoo_flashy.xsl
```

```
</AxStyleName>

# set the plain style as default
AxStyle plain

# lets the ?style=rss1 interface work for the news feed
AxAddPlugin Apache::AxKit::StyleChooser::QueryString
```

Here, the choice between styles is not so much a short-term, functional one (as in the list/detail view for your products document) but rather a persistent preference that should be reflected each time the reader visits the site. In this case, you are better off giving the user an HTTP cookie and using AxKit's Cookie StyleChooser.

But wait, you are already using the query string StyleChooser to provide the interface to the RSS feed. You don't want that link to break. No matter. Since AxKit plug-ins are executed in the order that they appear in the configuration, you only need to add the Cookie StyleChooser *before* the one that examines the query string. That way, even if the user has a cookie that sets AxKit's preferred_style, its value will be overwritten by the query string StyleChooser. Therefore, the functional transformation that provides the RSS interface will continue to work:

```
# lets the ?style=rss1 interface work for the news feed
# and allows skinning based on the user's cookie
AxAddPlugin Apache::AxKit::StyleChooser::Cookie
AxAddPlugin Apache::AxKit::StyleChooser::QueryString
```

What if you decide not to rely solely on AxKit's order of execution to handle the cases in which the styles may overlap? You can choose instead to write a little plug-in module that combines the two StyleChoosers while giving RSS query string interface top priority. (See Example 4-1.)

Example 4-1. CryptidStyleChooser.pm

```
package CryptidStyleChooser;

# Combine the Cookie and QueryString StyleChoosers, giving
# preference to a style named 'rss1' in the query string.

use strict;
use Apache::Constants qw(OK);
use Apache::Cookie;

sub handler {
    my $r = shift;

    my $preferred_style = undef;

    # Borrow the key names for the existing StyleChoosers
    my $query_key  = $r->dir_config('AxStyleChooserQueryStringKey') || 'style';
    my $cookie_key = $r->dir_config('AxStyleChooserCookieKey') ||
                     'axkit_preferred_style';

    my %query_params = $r->args( );

    if ( defined ($query_params{$query_key} ) ) {
        $style = $query_params{$query_key});

        # give preference to the rss1 query param by returning if you find it set.
        if ( $style eq 'rss1' ) {
            $r->notes('preferred_style', $style);
            return OK;
        }
    }

    # let the users' cookie override the query string otherwise
    my $cookies = Apache::Cookie->fetch; . .
    if ( defined $cookies->{$cookie_key} ) {
        $style = $cookies->{$cookie_key}->value);
    }

    # finally, set AxKit's internal 'preferred_style' if you found a style
    if ( defined( $style ) ) {
        $r->notes('preferred_style', $style;
    }
    return OK;
}

1;
```

To use your custom StyleChooser plug-in, you need to install it on your server in a location that Perl knows about and to replace the former AxAddPlugin directives with:

```
# lets the ?style=rss1 interface work for the news feed
# and allows skinning based on the user's cookie
AxAddPlugin CryptidStyleChooser
```

You can now rest assured that the correct stylesheets will be applied to the documents in your cryptid site for each case that arises. In reality, custom StyleChooser plug-ins are rarely required. (In fact, you didn't really *need* one here.) By creating a StyleChooser that covers both the presentational and functional transformations for your site, you peeked at just how easy it can be to use named styles to get AxKit to apply exactly the styles that you want, no matter how tricky the requirements first appear.

4.3.2 AxMediaType and MediaChoosers

In addition to named styles, AxKit offers a way to further define the styles that are conditionally applied to a site's resources by associating sets of styling directives with the media type the requesting client expects. This is achieved using the <AxMediaType> container directive.

```
<AxMediaType screen>
  AxAddProcessor text/xsl /styles/screen.xsl
</AxMediaType>
```

Similar to the <AxStyleName> block, the <AxMediaType> tells AxKit to examine an internal property, preferred_media, set by a MediaChooser, or other plug-in. If the value of that property matches the name given to an <AxMediaType> block, the directives within that block are used to process the current request:

```
# Set a global style for most client devices
<AxMediaType screen>
  AxAddProcessor text/xsl /styles/screen.xsl
</AxMediaType>

# But give smaller devices a different look
<AxMediaType handheld>
  AxAddProcessor text/xsl /styles/handheld.xsl
</AxMediaType>
```

Unlike named styles, AxKit sets a default value of screen for the preferred media type, if no other is found. This behavior can be altered by setting the AxMedia directive to the name of the <AxMediaType> block that you want to use as the default. For example, the following configures AxKit to use the handheld media type as the default within the current context:

```
AxMedia handheld
```

Pretend that you want to further enhance your silly cryptozoology site. You want to support visitors using web phones. In this case, you simply create another set of stylesheets that renders the content appropriately for that platform and alter your configuration to reflect the change:

```
# the previous, screen-only config, now wrapped in an <AxMediaType> block for clarity
<AxMediaType screen>
  <AxStyleName plain>
    AxAddProcessor text/xsl stylesheets/cryptozoo.xsl
  </AxStyleName>

  <AxStyleName rss1>
    AxAddProcessor text/xsl stylesheets/cryptidsightings_rss.xsl
  </AxStyleName>

  <AxStyleName flashy>
```

```
    AxAddProcessor text/xsl stylesheets/cryptozoo_flashy.xsl

  </AxStyleName>

</AxMediaType>



# the new <AxMediaType> for handheld support

<AxMediaType handheld>

  <AxStyleName plain>

    AxAddProcessor text/xsl stylesheets/cryptozoo_handheld.xsl

  </AxStyleName>



  <AxStyleName rss1>

    AxAddProcessor text/xsl stylesheets/cryptidsightings_rss.xsl

  </AxStyleName>

</AxMediaType>



AxStyle plain
```

You can give the same name to two different-named style blocks in the same configuration context without collision, so long as they appear as the children of different media type blocks. Here, you give the default style contained by the new media type the same name as the default screen style. This allows the AxStyle directive to work as expected for both media types:

```
# lets the ?style=rss1 interface work for the news feed

AxAddPlugin CryptidStyleChooser

AxAddPlugin Apache::AxKit::MediaChooser::WAPCheck
```

With this addition, your absurd little mystery animals' site becomes quite respectable, technologically speaking. You offer users a choice of styles through which to view the content, an RSS 1.0 news feed of recent cryptid sightings, and full support for surfers using web phones—all this from two XML documents, a few stylesheets, and, most importantly, knowing how to configure AxKit to apply the right style at the right time.

4.4 Style Processor Configuration Cheatsheet

So far, we have looked at the various individual elements that can be used to control how AxKit applies style transformations. So many, in fact, that seeing how these parts all fit together may be a bit tough. For example, a named style block (selected by a StyleChooser based on an environmental condition) may contain one or more AxAddDTDProcessor or similar conditional processing directives that are only applied if an additional condition is met. True AxKit mastery comes from knowing how to combine all its various configuration options to create elegant styling rules that meet the need of your specific application.

To help examine the processing order for various configuration combinations, we will create a series of very simple XSLT stylesheets whose sole purpose is to show the order in which AxKit applies a given style. The stylesheet in Example 4-2, *alpha.xsl*, simply appends the string . . . Alpha processed to the text of the top-level root element of the document being processed.

Example 4-2. alpha.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template match="/">
  <root><xsl:value-of select="/*"/> . . . Alpha processed</root>
</xsl:template>

</xsl:stylesheet>
```

The tiny sample XML document used for the transformations is shown in Example 4-3.

Example 4-3. minimal.xml

```
<?xml version="1.0"?>
<root>Base content</root>
```

Add more stylesheets to these—*beta.xsl*, *gamma.xsl*, and so on—that do more or less the same thing—that is, adding . . . Beta processed, etc., to the text of the root element. Wherever a simple description does not suffice, use these stylesheets to examine the precise processing order based on the returned result.

4.4.1 Rule 1: Style Processors at the Same Lexical Level Are Evaluated in Configuration-File Order

Wherever more than one style processing directive exists within a given context, each will be added (or, in the case of conditional processors, evaluated) in the order in which they appear in the configuration files. In cases in which directives are added to the same context by both the *httpd.conf* file and an *.htaccess* file, those from the *httpd.conf* are added or evaluated first:

```
<Directory /www/mysite.com/>
   AxAddProcessor text/xsl /styles/alpha.xsl
   AxAddProcessor text/xsl /styles/beta.xsl
</Directory>
```

As you may expect with this configuration, a document requested from root directory *mysite.com* will have the *alpha.xsl* stylesheet applied to the source content, then the *beta.xsl* stylesheet applied to the result of the first transformation:

```
<?xml version="1.0"?>

<root>Base content . . . Alpha processed . . . Beta processed</root>
```

Similarly, the rules governing conditional processing directives are tested in the order in which they appear in the configuration files:

```
<Directory /www/mysite.com/>

  AxAddProcessor text/xsl /styles/alpha.xsl

  AxAddRootProcessor text/xsl /styles/beta.xsl root

  AxAddProcessor text/xsl /styles/gamma.xsl

</Directory>
```

Here, the *alpha.xsl* stylesheet is added unconditionally. Then, if the source document's top-level element is named <root>, the *beta.xsl* stylesheet is added. Finally, the *gamma.xsl* stylesheet is unconditionally added to the processing chain. The result looks like this:

```
<?xml version="1.0"?>

<root>Base content . . . Alpha processed . . . Beta processed . . . Gamma processed</root>
```

If the root element of the source document were something other than <root>, or if that element were bound to a namespace that did not match the one used in the AxAddRootProcessor directive, only the *alpha.xsl* and *gamma.xsl* processors would be added to the processing chain. Compare this configuration snippet to the previous one:

```
<Directory /www/mysite.com/>

  AxAddProcessor text/xsl /styles/alpha.xsl

  AxAddRootProcessor text/xsl /styles/beta.xsl {http://myhost.tld/namspaces/root}root

  AxAddProcessor text/xsl /styles/gamma.xsl

</Directory>
```

Here, the *beta.xsl* would *not* be added to the processing chain because, although the name of the top-level element matches the one in your AxAddRootProcessor directive, the directive specifies that the <root> element must be bound to the http://myhost.tld/namespaces/root namespace URI. This is not the case in our sample XML document. Therefore, only the *alpha.xsl* and *gamma.xsl* stylesheets would be applied.

Mixing various conditional processing directives at the same level can be quite powerful. Many simple sites really only use a handful of source XML grammars, and if only one result format is expected (i.e., HTML), setting up a block of carefully chosen conditional processors for the host's top-level directory can often cover most, if not all, the processing rules for the entire site. Here is a more practical example:

```
# The root directory for our AxKit-enabled site

<Directory /www/mysite.com/>

  AxAddDocTypeProcessor text/xsl /styles/docbook/html/docbookx.xsl "-//OASIS//DTD DocBook

 XML V4.2//EN"

  AxAddRootProcessor text/xsl  /styles/rss2html.xsl {http://www.w3.org/1999/02/22-rdf-
syntax-ns#}RDF

   AxAddRootProcessor application/x-xsp NULL {http://apache.org/xsp/core/v1}page

   AxAddRootProcessor text/xsl /styles/appgrammar2html.xsl

{http://apache.org/xsp/core/v1} page

</Directory>
```

Here, you have a small group of conditional processing directives—one that will be applied to documents containing a Document Type Definition with the SYSTEM ID indicating the latest version of DocBook, one that will be applied if the content's top-level element is named RDF and that element is bound to the http://www.w3.org/1999/02/22-rdf-syntax-ns# namespace, and two that will match any document that contains a top-level page element bound to the http://apache.org/xsp/core/v1 namespace URI.

Just by setting up this simple block of conditional processing directives at the top level of the site, you can now publish static RSS 1.0 news feeds and DocBook documents (in their myriad forms from books to FAQs), as well as generate dynamic XML content via eXtensible Server Pages from any directory at or below this level in the site. AxKit will simply work as expected. Admittedly, this setup presumes that you will publish documents only as HTML, but that's still a lot of power for minimal effort.

By setting two conditional AxAddRootProcessor directives to match the same root element, you create a two-step processing chain for XSP documents in which AxKit's XSP engine processes the source, and the */styles/appgrammar2html.xsl* XSLT stylesheet processes the resulting markup.

4.4.2 Rule 2: Conditional Processing Directives Are Evaluated Only Against the Original XML Source

The rules for style processors that are applied conditionally based on a feature contained in the XML source (AxAddDTDProcessor, AxAddRootProcessor, etc.) are only evaluated against the original source content, not subsequent transformations in the processing chain. That is, the rules are evaluated *once*, using the source document returned from the ContentProvider. The rules are *not* reevaluated. Therefore, if you have a stylesheet that changes the root element name during transformation, and that new root element coincides with the name passed to an AxAddRootProcessor directive that is in scope for that request, the AxAddRootProcessor rule does *not* match, since the transformed result is not examined. Similarly, only those xml-stylesheet processing instructions contained in the original source document will be considered. You cannot add to or otherwise alter the processing chain by adding a stylesheet PI to the result of a transformation.

4.4.3 Rule 3: Style Processors Are Prepended to the Processing Chain as You Descend into the Directory Hierarchy

Although we touched on this earlier, it bears repeating: style definitions are *prepended* to the processing chain as one descends into the site's directory tree. That is, processing definitions at deeper levels in the site are added to the front of the processing chain, not to the end. This is most easily explained with a simple example using the stylesheets that illustrate AxKit processing order:

```
# the site's DocumentRoot directory

<Directory /www/mysite.com/>

   AxAddProcessor text/xsl /styles/alpha.xsl

</Directory>


# a child directory of the DocumentRoot

<Directory /www/mysite,com/levelone/>

   AxAddProcessor text/xsl  /styles/beta.xsl
```

```
</Directory>


# a child directory of the 'levelone' directory

<Directory /www/mysite.com/levelone/leveltwo/>

   AxAddProcessor text/xsl  /styles/gamma.xsl

</Directory>
```

Given this configuration, a request for http::/myhost.tld/levelone/leveltwo/minimal.xml yields the following result:

```
<?xml version="1.0"?>

<root>Base content . . . Gamma processed . . . Beta processed . . . Alpha processed</root>
```

The styles are added to the processing chain from the bottom up, so the *alpha.xsl* stylesheet is applied last since it is at the top level of the site's hierarchy. Again, the reason is that, in general, most sites deploy more generic styles (those that will be applied to all or most of the documents in the site) at higher levels in the directory tree. This behavior allows that to work as expected, while handling special cases in lower-level directories by prepending special processors to the chain.

To see the utility of this behavior, imagine that you are publishing a site that is divided into several sections. In addition to the content being rendered, these sites usually have a certain amount of markup that appears on every page (a common graphical header, a legal/copyright footer, a common navigation bar, etc.). They also have a certain amount of markup that is common, but unique, to each individual section (section headers, contextual navigation, etc.). By building the processing chain from the bottom up, AxKit allows you to put the site-wide boilerplate markup in a top-level stylesheet, while handling the section-specific markup in each section's unique directory.

```
# the site's DocumentRoot

<Directory /www/mysite.com/>

   # the global stylesheet that contains the site-wide headers/footers, etc.

   AxAddProcessor text/xsl  /styles/global.xsl

</Directory>


<Directory /www/mysite.com/products/>

   # adds the markup common to all pages in the 'products' section (contextual navigation,

 section headers, etc)..

   AxAddProcessor text/xsl  /styles/products.xsl

</Directory>


<Directory /www/mysite.com/locations/>

   # adds the markup common to all pages in the 'locations' section . . .

   AxAddProcessor text/xsl  /styles/locations.xsl

</Directory>
```

Here, a request for a document from the products directory would be transformed by the */styles/locations.xsl* stylesheet to add the section-specific content, then passed to the */styles/global.xsl* stylesheet to add the site-wide boilerplate. This strategy presumes that higher-level stylesheets add only what is needed at that level and pass the rest of the markup through as-is. For example, with the above configuration, the stylesheet applied to the source content may return a result that has a div root element containing the page's main text; that result may then be wrapped in another div by the */styles/location.xsl* stylesheet to add the section-specific markup (while copying the result of the original transformation through, untouched). Finally, *that* result is wrapped by the */styles/global.xsl* stylesheet to create the completed document. (Think: the component-based approach to constructing pages but built from the bottom up.)

Also remember that Rule 1 still applies: style definitions in the same lexical scope are evaluated in the order found in the configuration files. This means that if you have more than one processing directive that matches at a given level in the directory hierarchy, the processors for that level are still added in the order they appear, and that *group* of processors will be prepended to the list of processors from a higher level. An example may help clarify:

```
 # the site's DocumentRoot directory

<Directory /www/mysite.com/>

  AxAddProcessor text/xsl /styles/alpha.xsl

  AxAddProcessor text/xsl  /styles/beta.xsl

</Directory>


# a child directory of the DocumentRoot

<Directory /www/mysite.com/levelone/>

  AxAddProcessor text/xsl  /styles/gamma.xsl

  AxAddProcessor text/xsl  /styles/delta.xsl

</Directory>


# a child directory of the 'levelone' directory

<Directory /www/mysite.com/levelone/leveltwo/>

  AxAddProcessor text/xsl  /styles/epsilon.xsl

</Directory>
```

Here, using your processing order stylesheets, a request for http::/myhost.tld/levelone/leveltwo/minimal.xml returns:

```
<?xml version="1.0"?>

<root>Base content . . . Epsilon processed . . . Gamma processed . . .

   Delta processed . . . Alpha processed . . . Beta processed</root>
```

At each level, the processors are added in the order found in the configuration file for that scope, but each of those groups of processors is added to the chain from the bottom up. Now, apply this idea to the previous sample. You can see how the page-level content can be generated for the location and products sections of the site:

```
# the site's DocumentRoot

<Directory /www/mysite.com/>

  # the global stylesheet that contains the site-wide headers/footers, etc.

  AxAddProcessor text/xsl  /styles/global.xsl

</Directory>


<Directory /www/mysite,com/products/>

  # generate XML from database select for 'products'

  AxAddProcessor application/x-xsp NULL


  # Transform 'product' application grammar to HTML
  AxAddProcessor text/xsl /styles/product2html.xsl


  # adds the markup common to all pages in the 'products'

  # section (contextual navigation, section headers, etc)..

  AxAddProcessor text/xsl  /styles/products.xsl

</Directory>


<Directory /www/mysite,com/locations/>

  # generate XML from database select for 'locations'

  AxAddProcessor application/x-xsp NULL


  # Transform 'product' application grammar to HTML

  AxAddProcessor text/xsl /styles/locations2html.xsl


  # adds the markup common to all pages in the 'locations' section . . .

  AxAddProcessor text/xsl  /styles/locations.xsl

</Directory>
```

Now, each subdirectory has two processing definitions: one that processes the XSP source content to generate markup for the location and products data from a database query, respectively, and one that transforms each domain-specific grammar into HTML. The results of those transformations are then passed to the *global.xsl* stylesheet, which passes that markup through, adding things like the common header, site-wide navigation, copyright information, etc. Again, in each lexical scope (the *location/* and *products*/ subdirectories), the processor definitions are added to the processing chain in the same order they are found in the configuration file. However, since they are at a lower level in the directory structure than the global stylesheet, each ordered group is processed *before* the global one. If you want to keep this same layout but need to allow more than a single type of document in each of the subdirectories, you could use groups of conditional processing directives instead of the unconditional AxAddProcessor directives you have here to match the properties of each supported document type. As long as the result passes up to the global stylesheet in the grammar expected, you can mix and match as needed. Everything still works.

4.4.4 Rule 4: Processors in the Named BlocksAre Always Evaluated Last

Essentially, named style and media blocks create their own lexical scope. When a named style or media is selected by a StyleChooser, MediaChooser, or other plug-in and there are other processing directives that match for the same request, the processors from the named blocks are always added last. Let's break out the processing order stylesheets again and examine the details:

```
# the site's DocumentRoot

<Directory /www/mysite.com/>


 # First, add the QueryString StyleChooser so you can easily select named styles

 AxAddPlugin Apache::AxKit::StyleChooser::QueryString


 # Add a simple named style

 <AxStyleName styleone>

    AxAddProcessor text/xsl /styles/beta.xsl

    AxAddProcessor text/xsl /styles/gamma.xsl
```

```
  </AxStyleName>


  # Add a 'global' style

  AxAddProcessor text/xsl /styles/alpha.xsl

</Directory>
```

A request for the *minimal.xml* file in the directory that does not specify a preferred style yields the expected result—only the *alpha.xsl* stylesheet is applied:

```
<?xml version="1.0"?>

<root>Base content . . . Alpha processed</root>
```

But what happens if you select the named style for a request to the same document? Given Rule 1, you may expect the processors in the named style block to be applied first, since the named block appears before the global style. This is not the case. A request for *minimal.xml* with style=styleone in the query string gives the following:

```
<?xml version="1.0"?>

<root>Base content . . . Alpha processed . . . Beta processed . . . Gamma processed</root>
```

The styles contained in the <AxStyleName> block are added according to Rule 1. (They are at the same lexical level and are, therefore, added in the order they appear in the configuration file.) Because they are contained in that block, they are added last to the processing chain, *after* the unnamed, global style (*alpha.xsl*).

The "named processors are always last" rule also applies in cases in which a named block appears at a lower level in the document hierarchy than any matching unnamed directives. This creates a partial exception to Rule 3.

```
# the site's DocumentRoot

<Directory /www/mysite.com/>


  # Add the QueryString StyleChooser so you can easily select named styles

  AxAddPlugin Apache::AxKit::StyleChooser::QueryString


  # Add a 'global' style

  AxAddProcessor text/xsl /styles/alpha.xsl

</Directory>


# A first-level subdirectory

<Directory /www/mysite.com/levelone/>


  # Add a simple named style

  <AxStyleName styleone>

      AxAddProcessor text/xsl /styles/beta.xsl

      AxAddProcessor text/xsl /styles/gamma.xsl

  </AxStyleName>

</Directory>
```

Here, a request for *http://mysite.com/levelone/minimal.xml?style=styleone* gives the same result as before, despite the fact that the matching named style block is defined at a lower level in the directory tree than the global stylesheet configured for the site's DocumentRoot:

```
<?xml version="1.0"?>

<root>Base content . . . Alpha processed . . . Beta processed . . . Gamma processed</root>
```

One final, admittedly pernicious, example reveals precisely what's going on:

```
# the site's DocumentRoot
<Directory /www/mysite.com/>

  # Add the QueryString StyleChooser so you can easily select named styles
  AxAddPlugin Apache::AxKit::StyleChooser::QueryString

  # Add a 'global' style
  AxAddProcessor text/xsl /styles/alpha.xsl

# Add a simple named style
  <AxStyleName styleone>
      AxAddProcessor text/xsl /styles/beta.xsl
      AxAddProcessor text/xsl /styles/gamma.xsl
  </AxStyleName>
</Directory>

# A first-level subdirectory
<Directory /www/mysite.com/levelone/>

  # another  'global' style
  AxAddProcessor text/xsl /styles/delta.xsl

  # The style name here is the same as the one in the parent directory!
  <AxStyleName styleone>
      AxAddProcessor text/xsl /styles/eplison.xsl
      AxAddProcessor text/xsl /styles/zeta.xsl
  </AxStyleName>
</Directory>
```

Here, a request to *http://mysite.com/levelone/minimal.xml?style=styleone* results in the following:

```
<?xml version="1.0"?>

<root>Base content . . . Delta processed . . . Alpha processed . . . Epsilon processed . . .

    Zeta processed . . . Beta processed . . . Gamma processed</root>
```

Surprised? Let's take things step by step:

1. The global *alpha.xsl* stylesheet is added to the processing chain.
2. The global processor *delta.xsl* is configured for a directory at a lower level and is, therefore, prepended to the list of processors, according to Rule 3.
3. The style named styleone is selected, so the two processors contained in the named block at the root level of the site (*beta.xsl* and *gamma.xsl*) are added in configuration order, according to Rule 1.
4. The two processors (*epsilon.xsl* and *zeta.xsl*) in the named block in the levelone directory are also added in configuration order, according to Rule 1. Since they appear at a lower level in the directory tree, they are added, as a group, to the list of named styles *before* the two in the parent directory, according to Rule 3.
5. *All* processors associated with a named style or media block are added to the processing chain *after* all those that are not, according to Rule 4.

The simplest way to conceptualize the processing order for rare cases such as this is that all processing directives not contained in an <AxStyleName> or <AxMediaType> blocks are combined into one list, according to Rules 1-3; a second list of those contained in a named style or media block is created, also using Rules 1-3; then the two lists are combined, with the "named styles list" always appearing last.

If all this seems a bit much, rest assured. Cases such as this last example are extremely rare in the real world. The example here is particularly nasty only because it illustrates every detail of how AxKit creates its processing chain.

Now that you have had a close look at AxKit's many options for applying different transformative styles to XML documents. It is time to look at *how* those transformations happen by examining two of the more popular languages that it supports for transforming content: XSLT and XPathScript.

Chapter 4. Points of Style

Style is an important concept in the AxKit world. Much of the value that AxKit adds as an XML publishing and application server lies in the flexibility and ease with which documents can be associated with one or more sets of transformations that can be selected dynamically in response to a condition. From more common web-publishing tasks, such as vendor co-branding and user customization, to more advanced dynamic data-oriented applications, the key to developing clean, manageable sites with AxKit lies in learning how to apply the appropriate styles to the content to meet the specific need. In this chapter, I introduce AxKit's basic styling configuration options and show how these options can be combined to create sophisticated, responsive sites.

5.1 XSLT Basics

An XSLT stylesheet is made up of a single top-level xsl:stylesheet element that contains one or more xsl:template elements. These templates can contain literal elements that become part of the generated result, functional elements from the XSLT grammar that control such things as which parts of the source document to process, and often a combination of the two. The contents of the source XML document are accessed and evaluated from within the stylesheet's templates and other function elements using the XPath language. The following shows a few XSLT elements (the associated XPath expression is highlighted):

```
<xsl:value-of select="price"/>

<xsl:apply-templates select=" /article/section"/>

<xsl:copy-of select="order/items"/>
```

5.1.1 The XPath Language

XPath is a language used to select and evaluate various properties of the elements, attributes, and other types of nodes found in an XML document. In the context of XSLT, XPath is used to provide access to the various nodes contained by the source XML document being transformed. The XPath expressions used to access those nodes is based on the relationships between the nodes themselves. Nodes are selected using either the shortcut syntax that is somewhat reminiscent of the file and directory paths used to describe the structure of a Unix filesystem, or one that describes the abstract relationship axes between nodes (parent, child, sibling, ancestor, descendant, etc.). In addition, XPath provides a number of useful built-in functions that allow you to evaluate certain properties of the nodes selected from a given document tree. The most common components of an XPath expression are location paths and relationship axes, function calls, and predicate expressions.

5.1.1.1 Location paths and relationship axes

Borrowing from the Document Object Model, XPath visualizes the contents of an XML document as an abstract tree of nodes. At the top level of that tree is the root node represented by the string /. The root node is *not* the same as the top-level element (often called the document element) in the XML document, but is rather an abstract node above that level, which contains the document element and any special nodes, such as processing instructions:

```
<?xml version="1.0"?>

<?xml-stylesheet href="mystyle.xsl" type="text/xsl"?>

<page>

  <para>I &#2665; the XML Infoset</para>

</page>
```

In this document, the xml-stylesheet processing instruction is a meaningful part of the document as a whole, but it is not contained by the page document element. Were it not for the abstract root node floating above the document element, you would have no way to access the processing instruction from within your stylesheets.

The practical result of having a root node above the top-level document element is that all XPath expressions that attempt to select nodes using an absolute path from root node to a node contained in the document must include both an / for the root node and the name of the document element.

Here are a few examples of absolute location paths:

```
/

/html/body

/book/chapter/sect1/title

/recordset/row/order-quantity
```

Relative location paths are resolved within the context of the current node (element, attribute, etc.) being processed. The following are functionally identical:

```
chapter/sect1

child::chapter/sect1
```

Attribute nodes are accessed by using the attribute:: axis (or the shortcut, @), followed by the name of the attribute:

```
attribute::class

@class

sect1/attribute::id

sect1/@id

/html/body/@bgcolor
```

Relationship axes provide a way to access nodes in the document based on relative relationships to the context node that often cannot be captured by a simple location path. For example, you can look back up the node tree from the current node using the ancestor or parent axis:

```
ancestor::chapter/title

parent::chapter/title
```

or across the tree at the same level using the preceding-sibling or following-sibling axes:

```
preceding-sibling::product/@id

following-sibling::div/@class
```

XPath also provides several useful shortcuts to help make things easier. The . (dot) is an alias for the self axis, and .. is an alias for the parent axis. The // path abbreviation is an alias for the descendant-or-self axis. While using // can be expensive to process, it is hard to fault the simplicity it offers. For example, collecting all the hyperlinks in an XHTML document into a single nodeset, regardless of the current context or where the links may appear in the document, is as simple as:

```
//a
```

Similarly, you can select all the para descendants of the context node (the node currently being processed) using:

```
.//para
```

If all these relationship axes are confusing, recall that XPath visualizes the document as a hierarchical tree of nodes in much the same way that the filesystem is presented in most Unix-like operating systems. (Parents and ancestors are "up" towards the "root," siblings are at the same level on a given branch, and children and descendants are contents of the current node.)

5.1.1.2 Functions

XPath provides a nice list of built-in functions to help with node selection, evaluation, and processing. This list includes functions for accessing the properties of nodesets (e.g., position( ) and count( )), functions for accessing the abstract components of a given node (e.g., namespace-uri( ) and name( )), string processing functions (e.g., substring-before( ), concat( ), and translate( )), number processing (e.g., round( ), sum( ), and ceiling( )) and Boolean functions (e.g., true( ), false( ), and not( )). A detailed reference covering each function is not appropriate here, but a few useful examples are.

Get the number of para elements in the document:

```
count(//para)
```

Create a fully qualified URL based on the relative_url attribute of the context node:

```
concat('http://mysite.com/', @relative_url )
```

Replace dashes with underscores in the text of the context node:

```
translate(., '-', '_' )
```

Get the scheme name from a fully qualified URL:

```
substring-before(@url, '://' )
```

Quickly, get the total number of items ordered:

```
sum(/order/products/item/quantity)
```

In addition to the core functions provided by XPath, XSLT adds several additional functions to make the task of transforming documents easier, or more robust. Two are especially useful; the document( ) (which provides a way to include all or part of separate XML documents into the current result), and the key( ) function (which offers an easy way to select specific nodes from larger, regularly-shaped sets by using part of the individual nodes as a lookup). You can find a complete listing of XSLT's functions on the Web at http://www.w3.org/TR/xslt#add-func.

Many functions provided by XPath and XSLT can be useful for transforming the source document's content; others are only useful when combined with XPath predicate expressions.

5.1.1.3 Predicates

Predicate expressions provide the ability to examine the properties of a nodeset in a way similar to the WHERE clause in SQL. When a predicate expression is used, only those nodes that meet the criteria established by the predicate are selected. Predicates are set off from the rest of the expression using square brackets and can appear after any node identifier in the larger XPath expression. When the predicate expression evaluates to an integer, the node at that position is returned. The following all select the second div child of the context node:

```
div[2]

div[1 + 1]

div[position( ) = 2]

div[position( ) > 1 and position < 3]
```

More than one predicate can be used in a given expression. The following returns a nodeset containing the title of the first section of the fifth chapter of a book in DocBook XML format.

```
/book/chapter[5]/sect1[1]/title
```

Predicate expressions may also contain function calls. The following selects all descendants of the current node that contain significant text data but no child elements:

```
.//*[string-length(normalize-space(text( ))) > 0 and count(child::*) = 0]
```

5.1.2 Stylesheet Templates

The basic building block of an XSLT stylesheet is the xsl:template element. The xsl:template element is used to create the output of the transformation. A template is invoked either by matching nodes in the source document against a pattern contained by the template element's match attribute, or by giving the template a name via the name attribute and calling it explicitly.

The xsl:template element's match attribute takes a pattern expression that determines to which nodes in the source tree the template will be applied. These match rules can be evaluated and invoked from within other templates using the xsl:apply-templates element. The pattern expressions take the form of an XPath location expression that may also include predicate expressions. For example, the following tells the XSLT processor to apply that template to all div elements that have body parents:

```
<xsl:template match="body/div">

 . . .

</xsl:template>
```

You can also add predicates to your patterns for more fine-tuned matching:

```
<xsl:template match="body/div[@class='special']">

  . . .

</xsl:template>
```

By adding the @class='special' predicate, this template would only be applied to the subset of those elements matched by the previous example that have a class attribute with the value special.

Templates that contain match rules are invoked using the xsl:apply-templates element. If the xsl:apply-templates element's optionalselect attribute is present, the nodes returned by its pattern expression are evaluated one at a time against each pattern expression in the stylesheet's templates. When a match is found, the nodes are processed by the matching template. (If no match is found, a built-in template is used, and the children of the selected nodes are processed, and so on, recursively.) If the select attribute is not present, all child nodes of the current node are evaluated. This allows straightforward recursive processing of tree-shaped data.

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:template match="/">

   <!-- root template, always matches -->

   <html>

      <xsl:apply-templates/>

   </html>

  </xsl:template>


  <xsl:template match="article">

    <!-- matches element nodes named 'article' -->

    <body>

      <xsl:apply-templates/>

    </body>

  </xsl:template>


  <xsl:template match="para">

    <!-- matches element nodes named 'para' -->

    <p>

      <xsl:apply-templates/>
    </p>

  </xsl:template>


  <xsl:template match="emphasis">

    <!-- matches element nodes named 'emphasis' -->

    <em>

      <xsl:apply-templates/>

    </em>

  </xsl:template>


</xsl:stylesheet>
```

Applying this stylesheet to the following XML document:

```
<?xml version="1.0"?>

<article>

 <para>

 I was <emphasis>not</emphasis> pleased with the result.

 </para>

</article>
```

gives the following result:

```
<?xml version="1.0"?>

<html>

 <body>

 <p>

  I was <em>not</em> pleased with the result.

 </p>

 </body>

</html>
```

The name attribute provides a way to explicitly invoke a given template from within other templates using the xsl:call-template element. For example, the following inserts the standard disclaimer contained in the template named document_footer at the bottom of an HTML page:

```
<xsl:template match="/">

   . . .  more processing here  . . .

    <xsl:call-template name="document_footer"/>
    </body>

  </html>

</xsl:template>



<xsl:template name="document_footer">

  <div class="footer">

    <p>copyright © 2001 Initech LLC. All rights reserved.</p>

  </div>

</xsl:template>
```

The xsl:template element's mode attribute provides a way to have template rules that have the same match expression (match the same nodeset) but process the matched nodes in a very different way. For example, the following snippet shows two templates whose expressions match the element nodes named section. One displays the main view of the document (the default), and the other builds a table of contents:

```
<xsl:template match="article">

  <!-- create the table of contents first -->

  <div class="toc">

    <xsl:apply-templates select="section" mode="toc"/>

  </div>

  <!-- then process the document as usual -->

  <xsl:apply-templates select="section"/>

</xsl:template>



<xsl:template match="section">

  <div class="section">

    <h2>

      <a name="{generate-id(title)}">

        <xsl:value-of select="title"/>

      </a>

    </h2>

    <xsl:apply-templates/>

  </div>

</xsl:template>



<xsl:template match="section" mode="toc">

  <a href="#{generate-id(title)}">

    <xsl:value-of select="title"/>

  </a>

  <br />

  <xsl:apply-templates select="section" mode="toc"/>

</xsl:template>
```

5.1.3 Loop Constructs

Adding to its flexibility, XSLT borrows the concepts of iterative loops and conditional processing from traditional programming languages. The xsl:for-each element is used for iterating over the nodes in the nodeset returned by the expression contained in its select attribute. In spirit, this corresponds to Perl's foreach (@some_list) loop.

```
<xsl:template match="para">

  <xsl:for-each select="xlink">

    . . .  do something with each xlink child of the current para element

  </xsl:foreach>

</xsl:template>
```

In the earlier example showing how template modes are used, you created two templates for the section elements: one for processing those nodes in the context of the main body of the document, and one for building the table of contents. You could just as easily have used an xsl:for-each element to create the table of contents:

```
<xsl:template match="article">

  <!-- create the table of contents first -->

  <div class="toc">

    <xsl:for-each select=".//section">

      <a href="#{generate-id(title)}">

        <xsl:value-of select="title"/>

      </a>

      <br />

    </xsl:for-each>

  </div>

  <!-- then process the document as usual -->

  <xsl:apply-templates select="section"/>

</xsl:template>
```

So, which type of processing is better: iteration or recursion? There is no hard-and-fast rule. The answer depends largely on the shape of the node trees being processed. Generally, an iterative approach using xsl:for-each is appropriate for nodesets that contain regularly shaped data structures (such as line items in a product list), while recursion tends to work better for irregular trees containing mixed content (elements that can have both text data and child elements, such as articles or books). XSLT's processing model is founded on the notion of recursion (process the current node, apply templates to all or some of the current node's children, and so on). The point is that one size does not fit all. Having a working understanding of both styles of processing is key to the efficient and professional use of XSLT.

5.1.4 Conditional Blocks

Conditional "if/then" processing is available in XSLT using xsl:if, xsl:choose, and their associated child elements.

The xsl:if element offers an all-or-nothing approach to conditional processing. If the expression passed to the processor through thetest attribute evaluates to *true*, the block is processed; otherwise, it is skipped altogether.

Consider the following template. It prints a list of an employee's coworkers, adding the appropriate commas in between the coworker's names (plus an and just before the final name) by testing the position( ) of each coworker child:

```
<xsl:template match="employee">
  <p>
    Our employee <xsl:value-of select="first-name"/> works with:
    <xsl:for-each select="coworker">
      <xsl:value-of select="."/>
      <xsl:if test="position( ) != last( )">, </xsl:if>
      <xsl:if test="position( ) = last( )-1"> and </xsl:if>
    </xsl:for-each>
    on a daily basis.
  </p>
</xsl:template>
```

In cases in which you need to emulate the if-then-else block or switch statement found in most programming languages, use the xsl:choose element. An xsl:choose must contain one or more xsl:when elements and may contain an optional xsl:otherwise element. The test attribute of each xsl:when is evaluated in turn and contents processed for the first expression that evaluates to true. If none of the test conditions return a true value and an xsl:otherwise element is included, the contents of that element are processed:

```
<xsl:template match="article">
  <xsl:choose>
    <xsl:when test="$page-view='toc'">
      <xsl:apply-templates select="section" mode="toc"/>
    </xsl:when>
    <xsl:when test="$page-view='summary'">
      <xsl:apply-templates select="abstract" mode="summary"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-templates select="section"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

5.1.5 Parameters and Variables

XSLT offers a way to capture and reuse arbitrary chunks of data during processing via the xsl:param and xsl:variable elements. In both cases, a unique name is given to the parameter or variable using a name attribute, and the contents can be accessed elsewhere in the stylesheet by prepending a $ (dollar sign) to that name. Therefore, the value of a variable declaration whose name attribute is myVar will be accessible later as $myVar.

The xsl:param element serves two purposes: it provides a mechanism for passing simple key/value data to the stylesheet from the outside, and it offers a way to pass information between templates within the stylesheet. One benefit of using XSLT in an environment such as AxKit is that all HTTP parameters are available from within your stylesheets via xsl:param elements:

```
<?xml version="1.0"?>

<xsl:stylesheet

    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

    version="1.0">

<xsl:param name="user"/>


<xsl:template match="/">

  . . .

  <p>Greetings, <xsl:value-of select="$user"/>.

  Welcome to our site.</p>

  . . .

</xsl:template>

 . . .
```

The result of a request to a document that is transformed by this stylesheet, such as http://myhost.tld/mypage.xml?user=ahclem (or a POST request that contains a defined "user" parameter), contains the following:

```
  <p>Greetings, ahclem. Welcome to our site.</p>
```

Default values can be set by adding text content to the xsl:param element:

```
<xsl:param name="user">Mystery Guest</xsl:param>
```

or by passing a valid XPath expression to its select attribute:

```
<xsl:param name="user" select="'Mystery Guest'"/>

<xsl:param name="my-value" select="/path/to/default"/>
```

Parameters can also be used to pass data to other templates during the transformation process. In this second form, a template rule is invoked using the xsl:call-template element, whose required name attribute must correspond to the name attribute of an xsl:template, or by using an xsl:apply-templates that is expected to match one of the template's match attributes. In both cases, data can be passed to the template via one or more xsl:with-param elements, which are then available inside its invoked template using xsl:param elements. The result returned by the called template is inserted into the transformation's output at the point that the template is called.

Calling named templates and passing in parameters is the closest thing that XSLT has to the user-defined functions or subroutines provided in traditional programming languages. It can be employed to create reusable pseudofunctions:

```
<xsl:template match="guestlist">

  <xsl:for-each select="visitor">

    <xsl:call-template name="lastname-first">

      <xsl:with-param name="fullname" select="name"/>

    </xsl:call-template>

  </xsl:for-each>

  . . .

</xsl:template>

 . . .
```

```
<xsl:template name="lastname-first">
  <xsl:param name="fullname"/>
  <xsl:value-of select="$fullname/lastname"/>
  <xsl:text>, </xsl:text>
  <xsl:value-of select="$fullname/firstname"/>
  <xsl:if test="$fullname/middle-initial">
    <xsl:text> </xsl:text>
    <xsl:value-of select="$fullname/middle-initial"/>
    <xsl:text>.</xsl:text>
  </xsl:if>
</xsl:template>
```

The xsl:variable element is similar to xsl:param in that it can be assigned an arbitrary value such as a nodeset or string. Unlike parameters, though, variables only provide a way to store chunks of data; they are not used to pass information in from the environment or between templates.

They can be useful for such things as creating shortcuts to complex nodesets:

```
<xsl:varable name="super-stars"
             select="/roster/players[points-per-game > 25]"
/>
```

or setting default values for data that may be missing from a given part of the document:

```
<xsl:varable name="username">
 <xsl:choose>
  <xsl:when test="/application/session/username">
    <xsl:value-of select="/application/session/username"/>
  </xsl:when>
  <xsl:otherwise>Unknown User</xsl:otherwise>
 </xsl:choose>
</xsl:variable>
```

Once a variable or parameter has been assigned a value, it becomes read-only. This behavior trips most web developers who are used to doing such things as:

```
my $grand_total = 0;
foreach my $row (@order_data) {
        $grand_total += $row->{quantity} * $row->{price};
}
print "Order Total: $grand_total";
```

There is a way around this limitation, but it requires creating a template that recursively consumes a nodeset while passing the sum of the previous value and current value back to itself through a parameter as each node is processed:

```
<xsl:template match="/">
<root>
   . . .
  Order total:
  <xsl:call-template name="price-total">
    <xsl:with-param name="items" select="order/item"/>
  </xsl:call-template>
</root>
</xsl:template>

<xsl:template name="price-total">
  <xsl:param name="items"/>
  <xsl:param name="total">0</xsl:param>
  <xsl:choose>
    <xsl:when test="$items">
      <xsl:call-template name="price-total">
        <xsl:with-param name="items"
                              select="$items[position( ) != 1]"/>
        <xsl:with-param name="total"
                         select="$total + $items[position( )=1]/quantity/text( ) *
$items[position( )=1]/price/text( )"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="format-number($total, '#,##0.00')"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

If you are more used to thinking in Perl, the following snippet illustrates the same principle:

```
my $order_total = &price_total('0', @order_items);

sub price_total {
    my ($total, @items) = @_;
        if (@items) {
                my $data = shift (@items);
                &price_total($total + $data->{price} * $data->{quantity}, @items);
        }
        else {
```

```
        return $total;

    }

}
```

This short introduction to XSLT's syntax and features really only touches the surface of what it can achieve. If what you read here intrigues you, I strongly recommend picking up one of the many fine books that cover the language in much greater depth.

5.2 A Brief XSLT Cookbook

As I mentioned in the introduction, the goal of this chapter is to provide just enough information to allow you to be productive as quickly as possible with XSLT and AxKit. To that end, we will complete our whirlwind tour by looking at how XSLT can be used for several tasks that web developers are commonly asked to perform.

5.2.1 Delivering Browser-Friendly HTML

5.2.1.1 Problem

Your stylesheets work fine, but older HTML browsers are choking on tags such as <br/> and <img/>.

5.2.1.2 Solution

Use the xsl:output element and set its method attribute to "html":

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output method="html" />

  . . .
```

5.2.1.3 Discussion

The xsl:output element offers an easy way to control the formatting of the result of a given transformation. Other valid values for the method attribute include "text", and "xml" (the default). This element offers several other useful options, including the ability to set the encoding of the result document (via the encoding attribute) and the ability to add a document type declaration to the output (using the doctype-system and doctype-public attributes).

5.2.2 Alternating Colors in HTML Table Rows

5.2.2.1 Problem

You are transforming a document that consists of a long list of line items into HTML. You want to make the background of every other row a different color so that the page is more readable.

5.2.2.2 Solution

Use an xsl:if element that tests the value of the current node's position and conditionally adds the proper attribute to the output:

```
<xsl:template match="recordset/item">

  <tr>

    <xsl:if test="position( ) mod 2 != 1">

      <xsl:attribute name="bgcolor">#eeeeee</xsl:attribute>

    </xsl:if>

    <xsl:apply-templates/>

  </tr>

</xsl:template>
```

5.2.2.3 Discussion

Here, a combination of the position( ) function and the mod operator are used to test whether to add the bgcolor attribute to the surrounding table row. The expression position( ) mod 2 != 1 evaluates to true only for nodes in odd-numbered positions in the larger nodeset, so the bgcolor attribute is added only to every other row (starting with the first row).

5.2.3 Using XSLT Parameters with POST and GET Data

5.2.3.1 Problem

You want to make your stylesheets more dynamic by being able to access the POST and GET parameters that are part of a given HTTP request.

5.2.3.2 Solution

Use an xsl:param element and use the name of the desired field as the name of the parameter:

```
<xsl:param name="my-url-param"/>
```

5.2.3.3 Discussion

Unless explicitly configured to behave otherwise, all HTTP POST and GET parameters are available from within your stylesheets via top-level xsl:param elements. You only need to declare a top-level xsl:param whose name attribute corresponds to the name of the request field.

If you have caching turned on for documents being transformed by a stylesheet that uses a parameter extracted from the query string, be sure to add the following line to your *.htaccess* or other configuration file:

```
AxAddPlugin Apache::AxKit::Plugin::QueryStringCache
```

This plug-in makes AxKit's internal caching mechanism smarter with respect to query string parameters. More cache files are created on the disk (and therefore take more space), but it allows you to safely use the values of query string parameters inside your stylesheet without serving stale data or having to turn caching off altogether.

5.2.4 Creating a "Breadcrumb" Navigation Bar

5.2.4.1 Problem

You want to add context-sensitive navigation that allows visitors to easily climb the document hierarchy from the current folder (often called a *breadcrumb bar*).

5.2.4.2 Solution

Use AxKit's AddXSLParams::Request plug-in to pass the URL path of the source XML document to your XSLT stylesheet as an xsl:param and process that string into the desired HTML hyperlinks:

```
<?xml version="1.0"?>

<xsl:stylesheet

    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

    version="1.0">
```

```
<xsl:param name="request.uri"/>

<xsl:variable name="breadcrumb-delimiter">
  <xsl:text> :: </xsl:text>
</xsl:variable>

<xsl:template name="breadcrumb-nav">
<xsl:param name="url"/>
<xsl:param name="step-url"/>
<xsl:param name="links"/>

<xsl:choose>
  <xsl:when test="contains($url, '/')">

    <xsl:variable name="step">
      <xsl:choose>
        <xsl:when test="starts-with($url, '/')">
          <xsl:value-of select="'home'"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="substring-before( $url, '/')"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:variable>

    <xsl:variable name="local-url">
      <xsl:choose>
        <xsl:when test="$step='home'">
          <xsl:value-of select="'/'"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="concat( $step-url, $step, '/')"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:variable>

    <!-- call this template again to process the rest of the url -->
    <xsl:call-template name="breadcrumb-nav">
      <xsl:with-param name="url">
        <xsl:value-of select="substring-after( $url, '/')"/>
      </xsl:with-param>
      <xsl:with-param name="step-url">
        <xsl:value-of select="$local-url"/>
      </xsl:with-param>
      <xsl:with-param name="links">
        <xsl:copy-of select="$links"/>
        <xsl:value-of select="$breadcrumb-delimiter"/>
        <a href="{$local-url}"><xsl:value-of select="$step"/></a>
      </xsl:with-param>
    </xsl:call-template>

  </xsl:when>

  <!-- if you make it to here, all the url steps
       have been extracted. cover the last case
       and return the result -->

  <xsl:otherwise>
    <xsl:copy-of select="$links"/>
    <xsl:value-of select="$breadcrumb-delimiter"/>
    <a href="{concat($step-url, $url)}" >
      <xsl:value-of select="$url"/>
    </a>
  </xsl:otherwise>

</xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

5.2.4.3 Discussion

AxKit makes all form and query parameters available to the XSLT stylesheet via top-level xsl:param elements by default. This is not the only data that can be accessed in this way. AxKit's plug-in modules can pass *any* simple key/value pair as a stylesheet parameter. You learn more about this in Custom Plug-ins in Chapter 8, but in this case, you do not need a no custom plug-in; you can simply use the existing Apache::AxKit::Plugin::AddXSLParams::Request plug-in to pass the URI of the content source into your stylesheet. When added to the processing chain, this verbosely named class provides the ability to select sets of commonly used request and server information (HTTP headers, cookie data server hostnames, etc.) and makes them available as XSLT parameters. We will not dig into details of this plug-in's parameter groups or their naming conventions (see the installed documentation for more information), but suffice it to say, as with form and query string data, if the name of one of this plug-in's selected fields matches the name attribute of a top-level xsl:para element in your stylesheet, then the data is made available via that parameter. To access the request URI as an XSLT parameter, you need to first add and configure the plug-in via one of the web server's configuration files:

```
# Add the plug-in to the processing chain

AxAddPlugin Apache::AxKit::Plugin::AddXSLParams::Request



# Tell the plugin which group of data you are interested in

PerlSetVar AxAddXSLParamGroups "Request-Common"
```

With this in place, you can access the URI fron the current request using the following top-level parameter in your stylesheet:

```
<xsl:param name="request.uri"/>
```

Now that you can access the URL, you still need to process that string to generate the breadcrumb bar for the top of your pages. To do this, create a named template that recursively consumes the URL string and produces a series of HTML links separated by the text string defined by the $breadcrumb-delimiter variable.

A stylesheet such as the one in this example is a good candidate for inclusion in a reusable library of common templates. With that stylesheet imported, you can simply call the breadcrumb-nav template (passing the URI parameter as an argument) at the location in the page that you want the breadcrumb bar to appear. For example, the following inserts the link bar into its own div element at the top of the result's body element:

```
<?xml version="1.0"?>

<xsl:stylesheet

    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

    version="1.0">



<xsl:import href="/style/common/breadcrumb.xsl"/>



<xsl:template match="article">

<html>

  <body>

    <div class="breadcrumb">

       <xsl:call-template name="breadcrumb-nav">

         <xsl:with-param name="url">

         <xsl:value-of select="$request.uri"/>

       </xsl:with-param>

      </xsl:call-template>

    </div>

    <!-- main content continues.. -->

  </body>

</html>

</xsl:template>
```

With the template above as a part of your stylesheet, the resulting HTML generated from this snippet may look something like this:

```
<html>
  <body>
  <div class="breadcrumb">
    <a href="/">home</a> :: <a href="/samples/">samples</a> ::
    <a href="/samples/mypage.xml">mypage.xml</a>
  </div>
  <!-- main content continues.. -->
  </body>
</html>
```

The breadcrumb-nav template that does the real work is fairly verbose, and truly, you may have slimmed it down a bit by using an extension function. It does, however, illustrate the functional recursive template technique introduced earlier in this chapter in "Parameters and Variables." It underscores the fact that some tasks, most notably up-translating text strings into markup, are tricky but possible.

5.2.5 Passing Markup Through Untransformed

5.2.5.1 Problem

You have certain elements in your source documents that you want to appear untransformed in the result of a stylesheet transformation.

5.2.5.2 Solution

Create a template to match the nodes you want to copy, and use xsl:copy and xsl:copy-of.

Pass through a copy of the selected nodes, including their attributes and text children without including any descendant elements:

```
<xsl:template match="para">
  <xsl:copy>
    <xsl:copy-of select="@*|text( )"/>
  </xsl:copy>
</xsl:template>
```

Copy the selected nodes and all of their descendants as is:

```
<xsl:template match="article/articleinfo">
  <xsl:copy-of select="."/>
</xsl:template>
```

Copy the selected nodes, and pass their descendants on for further processing:

```
<xsl:template match="article/articleinfo">
   <xsl:copy>
     <xsl:copy-of select="@*"/>
     <xsl:apply-templates />
   </xsl:copy>
</xsl:template>
```

5.2.5.3 Discussion

The significant difference between xsl:copy and xsl:copy-of is that the former creates a shallow copy of the node being processed, while the latter creates a deep copy that includes all attributes, text nodes, and descendant elements. The xsl:copy element offers finer control over exactly which parts of a node are copied, but puts the onus on the stylesheet author to explicitly descend into the nodeset's structure to extract the parts that are to be copied into the result. The xsl:copy-of makes it easy to include complex nodesets in the result but relies on the fact you do not want to process any descendant nodes contained by that nodeset.

As a general rule, use xsl:copy-of when you want to include the entire nodeset without examining its contents, and a combination of xsl:copy and xsl:copy-of with xsl:apply-templates when the nodeset may contain other elements that you want to transform.

The following passes a defined subset of HTML elements through the stylesheet untransformed, while processing any non-HTML descendants according to the other templates contained in the stylesheet:

```
<xsl:template match="p|i|b|font|em|blockquote|span|pre|br|div|ul|ol|li|
                     img|script|html|head|meta|a|
                     table|tr|td|th|
                     form|input|select|option|textarea|
                     embed">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates />
  </xsl:copy>
</xsl:template>
```

Rather than listing all the elements by name as you do here, it's far more common to use what is popularly called an identity transformation template that copies through all nodes that do not have a more specific template that matches. This offers a way to operate on only the nodes that you need to, while passing the rest through undisturbed:

```
  <xsl:template match="node( )|@*">
    <xsl:copy>
      <xsl:apply-templates select="@*|node( )" />
    </xsl:copy>
  </xsl:template>
```

The template is rather common in processing chains in which an upstream stylesheet may have generated HTML (or whatever the client expects), and the last stylesheet in the chain wants to add a common header and footer to the body element while passing the rest of the content through, as is.

5.2.6 Including External Documents

5.2.6.1 Problem

You want to include all or part of another XML document in the result of an XSLT transformation.

5.2.6.2 Solution

Use the XSLT document( ) function.

Include the entire document as is:

```
<xsl:copy-of select="document('include.xml')"/>
```

Process the contents of the bookinfo element in the included document with the template rules in the current stylesheet:

```
<xsl:apply-templates select="document('book.xml')/bookinfo/*"/>
```

5.2.6.3 Discussion

XSLT's document( ) function offers a flexible mechanism for including all or some nodes in a separate XML document in the result of a transformation. It is especially useful for including common site elements such as headers, footers, and global navigation. The result of a call to document( ) is a nodeset containing all or some content of the external file. The resulting nodeset can be assigned to an xsl:variable and reused within the stylesheet:

```
<xsl:variable name="common_nav" select="document('sitenav.xml')"/>

<xsl:apply-templates select="$common_nav/*" mode="rightnav"/>

  . . .

<xsl:apply-templates select="$common_nav/*" mode="footerenav"/>
```

You can use the document( ) function for much more than adding chunks of common boilerplate. For example, you can include details about the products in a customer's order by looking up the product details in a separate file:

```
<xsl:template match="/">

  <xsl:variable name="item_details"

                select="document('productdetail.xml')"/>


  <xsl:for-each select="order/item">

    <xsl:variable name="lookup">

      <xsl:value-of select="@product_id"/>

    </xsl:variable>

    <item product_id="{$lookup}">

      <xsl:copy-of select="$item_details/productlist/item[@product_id=$lookup]/*"/>

    </item>

  </xsl:for-each>

  . . .

</xsl:template>
```

The way the document( ) function treats relative URIs often trips the unwary. Stylesheet authors typically expect the file path to be evaluated in the context of the document being transformed—it is not. It is processed in the context of the stylesheet, not the source document. When you type `select="document('header.xml')"`, remember that the XSLT processor looks in the same directory as the stylesheet for the file *header.xml*.

You can alter this default behavior, however, by using the two-argument form of `document( )`. In the two-argument form, the second argument is a nodeset. The URI of the source document that originally contained that nodeset is used to set context in which any relative path contained by the first argument is evaluated.

To resolve a relative file path in the context of the source document, rather than the stylesheet, use:

```
<xsl:copy-of select="document('widgets/footer.xml', /)"/>
```

This includes the contents of the file *footer.xml* from the directory *widgets* in the same directory as the source document. It works because the second argument (/) represents the root of the document currently being transformed. Therefore, the relative path contained in the first argument is resolved in the context of that URI.

5.2.7 Dividing Large Datasets into Pages

5.2.7.1 Problem

You have a large XML source document containing many records. You want to display only a few at a time.

5.2.7.2 Solution

Use the XPath `position( )` function to select a smaller subset of the records stored in the document.

Use `position( )` in the select expression of an `xsl:for-each` loop:

```
<xsl:template match="/">

<recordset>

  <xsl:for-each select="recordset/record[position ( ) >= $start and position( )

<= $end]">

    <xsl:copy-of select="."/>

  </xsl:for-each>

</recordset>
```

5.2.7.3 Discussion

Displaying a limited set of all records contained in a large document (*pagination*) is one of the most common XML publishing tasks. Selecting a given set of records is as easy as selecting the proper element nodes whose position falls within the desired range.

Using top-level `xsl:param` elements that allow the range and context to be passed dynamically to the stylesheet via the query string makes navigating large documents fairly straightforward. Example 5-1 demonstrates one way to navigate a large set of records using the type of interface popularized by major web search engines.

Example 5-1. An XSLT stylesheet for paginating large records

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

                version="1.0">

<xsl:output method="html" />



<xsl:param name="start">1</xsl:param>
```

```
<xsl:param name="perpage">10</xsl:param>

<xsl:variable name="totalitems" select="count(//item)"/>

<xsl:variable name="end">

  <xsl:choose>

    <xsl:when test="($start + $perpage) > $totalitems">

      <xsl:value-of select="$totalitems"/>

    </xsl:when>

    <xsl:otherwise>

      <xsl:value-of select="$start + $perpage - 1"/>

    </xsl:otherwise>

  </xsl:choose>

</xsl:variable>


<!-- begin root template -->

<xsl:template match="/">


  <h3>

    Showing records

    <xsl:value-of select="$start"/> - <xsl:value-of select="$end"/> of

    <xsl:value-of select="$totalitems"/>

  </h3>


  <table border="0" cellpadding="0" cellspacing="0">

    <tr><th>Product ID</th><th>Description</th><th>Price</th></tr>

    <xsl:for-each select="/order/item[position( ) >= $start and position( ) <= $end]">

      <tr>

        <!-- make every other row gray -->

        <xsl:if test="position( ) mod 2 != 1">

          <xsl:attribute name="bgcolor">eeeeee</xsl:attribute>

        </xsl:if>

        <td><xsl:value-of select="@product-id"/></td>

        <td><xsl:value-of select="name"/></td>

        <td><xsl:value-of select="price"/></td>

      </tr>

    </xsl:for-each>

  </table>


  <!-- if there are records before the block you are viewing, provide a 'prev.' link -->

  <xsl:if test="$start > 1">
```

```
    <a href="paginate.xml?start={$start - $perpage};perpage={$perpage}">prev.</a>
  </xsl:if>

  <!-- process *all* the <item> elements in the document to
       build the navbar -->
  <xsl:apply-templates select="/order/item"/>

  <!-- if there are more records, provide a 'next' link -->
  <xsl:if test="$totalitems > $end">
    <a href="paginate.xml?start={$end + 1};perpage={$perpage}">next</a>
  </xsl:if>

</xsl:template>
<!-- end root template -->

<!-- the 'item' template that builds the numbered navbar links -->
<xsl:template match="item">
  <xsl:if test="position( ) mod $perpage = 1 or $perpage = 1">
    <xsl:variable name="pagenum">
      <xsl:value-of select="ceiling(position( ) div $perpage)"/>
    </xsl:variable>
    <a href="paginate.xml?start={position( )};perpage={$perpage}">
      <xsl:value-of select="$pagenum"/>
      <!-- force whitespace in between the numbered links -->
      <xsl:text> </xsl:text>
    </a>
  </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Remember, XSLT is not, nor does it pretend to be, a general-purpose programming language. If the XML documents you are publishing require a significant amount of processing that is beyond the reach of XSLT, or if you are bending over backwards too often to get XSLT to do what you need, consider using XPathScript to transform your documents. Similar to XSLT in many respects, XPathScript adds the ability to process the data contained in your documents using Perl and its myriad functions and modules.

Chapter 5. Transforming XML Content with XSLT

XSLT (The eXtensible Stylesheet Language: Transformations) is an XML application language used for transforming XML documents into other documents. It is implemented by an application called an XSLT processor that takes an XML document and an XSLT stylesheet as input and produces a new document by applying the instructions contained in the stylesheet to the original source XML document. The result of an XSLT transformation can be any text-based format, but the output is typically either another XML document, or a document in a widely deployed markup language such as HTML that can be readily consumed by a given client application.

AxKit is not an XSLT processor, nor does it ship with one. If you want to use XSLT to transform your XML content using AxKit, you need to install an XSLT processor and any necessary Perl interface modules separately. For the list of XSLT processors that AxKit currently supports, see XML Processing Options in Chapter 2. For details about how to use the AxAddStyleMap directive to associate XSLT stylesheets with the processor you install and the various directives that govern which stylesheets are applied to your XML documents, see Chapter 4.

Exhaustive coverage of XSLT is well beyond the scope of this chapter. The goal here is to introduce enough of the basic concepts of writing XSLT stylesheets to allow you to start being productive with AxKit as quickly as possible. For a more detailed look at XSLT, see *XSLT*, by Doug Tidwell, or *Learning XSLT*, by Mike Fitzgerald (both from O'Reilly). All of the samples here use only XSLT 1.0. At the time of this writing, XSLT 2.0 is still very new and not widely implemented, and existing implementations are highly experimental. Rest assured though, XSLT 1.0 is still a viable tool. The topics covered here generally apply to both versions, and support for use of Version 2.0 processors from within AxKit will be added just as soon as stable implementations begin to appear.

6.1 XPathScript Basics

An XPathScript document's basic syntax is similar to that found in Active Server Pages and similar technologies, where literal output is separated from functional code blocks using the familiar `<% . . . %>` pseudotags as delimiters. As with Apache::ASP, the embedded language is Perl and the mixture of code and markup is processed, returning the content as it is encountered from the top down, while replacing the code blocks with the markup they return to produce the complete result:

```
<html>

  <body>

  <p>Greetings. The local time for this server is <% print scalar localtime; %>

  </p>

  </body>

</html>
```

Appending an equal sign «=» to the opening delimiter offers a shortcut that sends the interpolated values returned from any code it encloses directly to the output. The following would send the same result to the browser as shown in the previous example (the `print` statement is gone):

```
<html>

  <body>

  <p>Greetings. The local time for this server is <%= scalar localtime %>

  </p>

  </body>

</html>
```

XML elements and attributes can be added to the result from within code blocks by simply returning the appropriate textual representation:

```
  <select name="day">

  <%

     foreach my $day ( qw(Sun Mon Tues Wed Thu Fri Sat) ) {

        print "<option value='$day'>$day</option>";

     }

  %>

  </select>
```

In terms of lexical scoping, an XPathScript stylesheet behaves like any Perl script in that all variables and subroutines not enclosed within a block, or explicitly declared as part of another package, become part `package::main` and are available throughout the entire stylesheet. Among other things, this allows you to break your stylesheets into logical sections that can provide a somewhat cleaner division between functional code and markup.

```
<%
my @fish_names = qw( tuna halibut salmon scrod );
%>
<html>
  <body>
    <form>
      <p>Choose Your Favorite Fish</p>
      <select name="fave_fish">
      <%
         # @fish_names is still in scope here
         foreach my $fish ( sort(@fish_names) ) {
            print "<option value='$fish'>$fish</option>";
         }
      %>
      </select>
    </form>
  </body>
</html>
```

Any similarity between XPathScript and various Server Pages technologies ends at this superficial syntactic level. Remember, XPathScript was designed as an alternative to XSLT, and as such, it provides both a means for selecting and extracting information for an XML source as well as a means for creating and applying declarative rules for transforming that content—that is, unlike ASP, XPathScript documents are *stylesheets* that are applied to a source XML document to create a transformed result, not simply a source of dynamically generated content. The typical XPathScript stylesheet takes the form shown in Example 6-1.

Example 6-1. Structure of a typical XPathScript stylesheet

```
<%
    # Perl code block that imports any required modules,
    # and performs any required initialization
    use Some::Package;

    # Declarative template rules defined via the
    # special "template hash" $t
    $t->{some:element}->{pre}  = '<new>';
    $t->{some:element}->{post} = '</new>';
    # and so on..
 %>
  <root>
      <!-- begin literal output -->
    <child>
      <%
        # A mixture of literal output and code blocks that
        # call XPathScript functions to select data from
        # or apply templates to the source XML
      %>
    </child>
  </root>
```

From this, you can see that a typical XPathScript stylesheet consists of two parts: a Perl code block that contains any initialization required for the current transformation as well as the template rule configuration via XPathScript's special template hash, $t, and a block of literal output that provides the overall structure of the result and contains escaped code blocks through which the contents of the source XML document are accessed.

6.1.1 Accessing Document Content

Like XSLT, XPathScript uses the XPath language as the mechanism for selecting and evaluating the nodes in an XML document. (We touched on the basics of XPath in Section 5.1 in Chapter 5, and I will not duplicate that introduction here.) Recall that XPath provides a compact syntax for accessing and evaluating the contents of an XML document using a combination of location paths and function evaluation. In XPathScript, these XPath expressions are passed as arguments to one of a handful of Perl subroutines, implemented by the XPathScript processor, that perform the desired actions. The following sections provide an introduction to each of these subroutines.

6.1.1.1 findvalue( )

As the name suggests, the *findvalue* function offers a way to select the value from a given node in the source XML document. The required first argument to this function is the XPath location path that will be evaluated to select the node whose value will be returned.

```
<html>
  <head>
    <title><%= findvalue('/article/artheader/title') %></title>
  </head>
</html>
```

An optional second argument may be passed to the findvalue( ) function. This argument is expected to be a node from the current document and will be used to provide the context in which the expression contained in the first argument will be evaluated.

```
<%
foreach my $chapter ( findnodes('//chapter') ) {
  # select the title of each chapter
  my $title = findvalue('title', $chapter);
}
%>
```

This can be useful, but it is far more common to use the object-oriented interface, calling *findvalue* on the nodes themselves, rather than passing that node as the second argument. Calling *findvalue* as a method on the node has the effect of limiting the value returned to the contents of that specific node and gives the same result as calling *findvalue* as a function and passing in the given node as the second argument:

```
<%
foreach my $chapter ( findnodes('//chapter') ) {
    # Same as above
    my $title = $chapter->findvalue('title');
}
%>
```

As with XSLT's xsl:value-of element, if more than one node is returned from evaluating the given expression, the text descendants of all nodes selected will be concatenated (in document order) into a single string. For example, applying findvalue('para') to the following XML snippet:

```
<para>
    Well, <emphasis>I</emphasis> wouldn't say that.
</para>
```

produces this somewhat more ambiguous snippet:

```
Well, I wouldn't say that.
```

Unlike other XPathScript subroutines designed to select nodes from the source XML document, XPath expressions passed to *findvalue( )* are not limited to location path expressions:

```
<p>
    This document contains <%= findvalue('count(//section)') %> sections.
</p>
```

6.1.1.2 findnodes( )

While *findvalue( )* is used for retrieving the values of the selected nodes, the *findnodes( )* function is used to select the nodes themselves. In a list context, *findnodes( )* returns a list of all nodes that match the XPath expression passed as the first argument; in a scalar context, it returns an XML::XPath::NodeSet object. As with *findvalue( )*, a node may be passed as the optional second argument to constrain the context in which the XPath expression will be evaluated.

```
# print the 'id' attributes from all 'product' elements calling findnodes( ) in a list
context
foreach my $product ( findnodes('/productlist/product') ) {
    print $product->findvalue('@id') . "\n";
}

# the same, but calling findnodes( ) in a scalar context to return an XML::XPath::NodeSet
object
my $products = findnodes('/productlist/product');

foreach my $product ( $products->get_nodelist ) {
    print $product->findvalue('@id') . "\n";

}
```

6.1.1.3 findnodes_as_string( )

The *findnodes_as_string( )* function works just like *findnodes( )*, but it returns the textual representation of the selected nodes rather than a nodeset object. It is used primarily to copy larger document fragments, as is, into the result of the transformation. For example, the following copies the entire contents of the file */include/common_header.xhtml* into the result of the current process:

```
<body>

   <!-- insert the common header -->

   <%= findnodes_as_string('document("/include/common_header.xhtml")') %>

    . . .

</body>
```

Taken together, these subroutines provide direct access to the contents of the source XML document from within a stylesheet's code blocks. They are most useful for cases in which the types of transformations required would benefit from the ability to use Perl's control structures, operators, and functions to generate the result based on nodes from the source document. However, this is only half of the story; XPathScript also offers the ability to set up and apply template rules that will generate new content as each element node in a given set is visited by the XPathScript processor.

6.1.2 Declarative Templates

Declarative templates provide a way to easily set up transformations for all instances of certain elements in the source document. Template rules are defined by adding keys to a global Perl hash reference, $t. During processing, the names of selected nodes are evaluated against the key names in the hash that $t refers to. When a match is found, the node is transformed based on the contents of that template.

The result of applying a template to a matching element is governed by the values assigned to one or more of a specific set of subkeys for each key in the template hash. For example, the value assigned to the pre subkey is added to the output just *before* the matching element is processed, while the value assigned to post key is appended to the output just *after* the node is processed. Adding the following to your XPathScript stylesheet has the effect of replacing all section elements with div elements with a class attribute with the value section:

```
<%

$t->{'section'}{pre} = '<div class="section">';

$t->{'section'}{post} = '</div>';

%>
```

If you are familiar with XSLT, the above corresponds to the following template rule:

```
<xsl:template match="section">

    <div class="section">

        <xsl:apply-templates/>

    </div>

</xsl:template>
```

XPathScript template matches are based solely on the element's name rather than the evaluation of more complex XPath expressions used in XSLT. For example, the first template below matches all instances of the product element, while the second would *never* match, even though a node may exist in the source document that matches the given expression:

```
<%
# convert <product> elements to list items
$t->{'product'}{pre} = "<li>";
$t->{'product'}{post} = "</li>";

# will never match since only element names are tested
$t->{'/products/product'}{pre} = "<li>";
$t->{'products/product'}{post} = "</li>";
%>
```

The fact that matches are based on the element name only is the major difference between declarative templates in XPathScript and those in XSLT. While this may appear to limit XPathScript's power, you will see that XPathScript provides other facilities for more fine-tuned control over the templates' matching behavior. (See Section 6.2, later in this chapter.)

XPathScript is not implemented in XML, so there is really no way to bind an XML namespace URI to a prefix, according to the rules of XML. This means that if you are creating template rules or location expressions to match element and attribute names that use XML namespaces and prefixes, you must use the same literal string prefix used in the source document. Compare this to XSLT, in which a prefix in the stylesheet and the one in the source document can vary as long as they are bound to the same URI.

Now you have a general idea of how to create template rules. Let's examine how to apply those templates to the source content.

6.1.2.1 Applying templates

Templates are applied to the source content using the *apply_templates( )* function. This function accepts a single optional argument that can either be an XPath expression that selects the appropriate nodes to be transformed, or an XML::XPath::NodeSet object containing those nodes. If no argument is passed, the templates are tested for matches against *all* nodes in the current document.

```
# Apply template rules to all <product> elements and their children
print apply_templates('/productlist/product');

# The same, but pass a nodeset instead
my $product_nodes = findnodes('/productlist/product');
print apply_templates( $product_nodes );
```

The XPath expression passed to the *apply_templates* function can be as specific as the case requires, and Perl scalars will be interpolated if the expression is passed as a double-quoted string. Also, the «@» must be escaped in the following, so the processor does not confuse the XPath attribute selection syntax with a Perl array and attempt to interpolate it (this is not needed if the expression is passed as a single-quoted string):

```
<%
my $id = $some_object->get_id( );
my $product = findnodes("/productlist/product[\@id=$id]");

if ( $product->size( ) > 0 ) {
    print apply_templates( $product );

}

else {

   print "<p> Sorry, no product ID match for $id in the current document.</p>";

}

%>
```

When applying templates, you must use Perl's built-in *print* function or the <%= opening delimiter shortcut for the result of the template processing to appear in the final output:

```
<table>

<%

 # part of a much longer Perl block

 if ( $some_condition ) {

     print apply_templates('/productlist/product');

 }

%>

</table>


# The same, but using the <%= delimiter shortcut


<table>

    <%= apply_templates('/productlist/product') %>

</table>
```

In both cases, the result of any template processing appears in the final result at the same location that the print statement or special opening delimiter is found.

6.1.2.2 Importing templates

The *import_template* function provides a way to add template rules to the current stylesheet by importing an external stylesheet. This function's single required argument is the DocumentRoot-relative path to the stylesheet that you want to import.

```
<%

import_template('/path/to/included.xps')->( );

%>
```

Since an XPathScript stylesheet is really a sort of fancy preprocessed Perl script, and the template hash is a global variable within that script, imported templates have the potential to override or add to template rules defined in the current stylesheet. That is, the element hashes will be merged, but since the template hash can only have one unique subkey for each template rule, if your stylesheet has a rule such as $t->{'para'}{pre} = '<p>';, and the imported stylesheet also has a rule to match para elements and defines a pre subkey, then the subkey definition that appears *last* in the stylesheet (e.g., the one that assigns a value to the given subkey last) takes precedence. If different subkeys are defined for the same element match, both subkeys are applied during processing.

6.1.2.3 Expression interpolation

Template rules may also contain information selected from the document tree by adding XPath expressions delimited by curly braces «{ }» to the rule's value. The contents of the braces are passed to the *findvalue( )* function in which the expression is evaluated in the context of the current matching node, and the result is added to the output. For example, the following copies the value of the url attribute from the current <ulink> element into the href attribute of the newly created HTML hyperlink:

```
$t->{ulink}{pre}  = '<a href="{@url}">';

$t->{ulink}{post} = '</a>';
```

To save resources, this sort of interpolation is not turned on by default. It must be explicitly switched on by adding the AxXPSInterpolate configuration directive to your *httpd.conf* or other server configuration file:

```
AxXPSInterpolate 1
```

6.2 The Template Hash: A Closer Look

Understanding how the template hash works is crucial to the effective use of XPathScript. First, the template hash is a bit magical in that (unlike all other Perl variables you may use in your stylesheet) you need not initialize this hash in your stylesheet. It is declared invisibly during execution by the XPathScript processor. This means that you cannot safely initialize a scalar named $t to the top level of your XPathScript stylesheets without causing a conflict. Second, the template hash really takes the form of a hash of hashes. The names given to the top-level keys define the element names that will be matched when *apply_templates( )* is called, and the values for those keys are themselves hashes whose predefined keys determine how the given element will be processed if a match is found. A typical rule takes the following form:

```
$t->{<element name>}{<sub-key name>} = $some_value;
```

element name is the full name (including the namespace prefix) of the element you want to match, and *sub-key name* is one of eight special keys that the XPathScript processor uses to determine how to build the output for matching cases.

Here are all these special subkeys and their associated behaviors:

pre

The scalar value assigned to this key is added to the output just before the matching element is processed.

post

The value assigned is appended to the output just after the matching element is processed.

prechildren

The scalar value assigned is sent to the output before any child elements of the matching element are visited.

postchildren

The scalar value assigned is added to the output after all children of the matching element are processed.

prechild

The value assigned is added to the output before *each* child of the matching element is processed.

postchild

The value assigned is added to the output after each child of the matching element is processed.

showtag

If defined, this key has the effect of copying the current matching element's start and end tags into the output. By default, the start/end tags for all matching elements are stripped.

testcode

The value assigned to this key is expected to reference a subroutine that controls how the matching element is processed.

6.2.1 The Joys of testcode

The testcode key is far and away the most powerful and interesting of the template rule subkeys. The value assigned to this key references a Perl subroutine that handles the output. Each time a match is found for the top-level key, this subroutine is called and passes two arguments: the element node for the current match and a localized template hash reference that can be used to set the other subkeys for the current template. The private template hash can be used in the same way as the global template hash, but its effects are limited to the current node, and there is no need to add the top-level key containing the element name.

```
# Set the pre and post keys for an <item> elements template in the typical way

$t->{item}{pre} = '<li>';

$t->{item}{pre} = '</li>';



# The same, but via testcode

$t->{item}{testcode} = sub {

    my ($node, $template ) = @_;

    $template->{pre}  =  '<li>';

    $template->{post} = '</li>';

};
```

Perl code executed inside the *mod_perl* environment needs to be a bit stricter than that allowed in typical CGI scripts. Specifically, you should avoid creating closures (subroutines that reference lexical variables outside the scope of the subroutine itself) when creating your testcode subroutines, or you will certainly get unexpected results from your XPathScript stylesheets. For more detailed information about closures and *mod_perl*, see the reference pages at http://perl.apache.org/docs/general/perl_reference/perl_reference.html#.

The testcode option also provides a way to conditionally process a given element based on the properties of the element node itself. In XSLT, this kind of property examination often happens when the XPath expression for a given template rule is evaluated, but in XPathScript, template matches are made only against the element name, and you need to examine the node properties (or other conditions) from within the template's testcode subroutine.

Suppose that you are transforming DocBook XML documents into HTML. Even if you are using the simplified subset (*sdocbook.dtd*), the title element can validly appear in 22 different contexts; thus, a single, simple template rule based on the element name cannot suffice. The following shows how you could begin to approach the problem:

```
$t->{title}{testcode} = sub {

    my ($node, $template ) = @_;



    # Get the parent element's name

    my $parent_name = $node->findvalue('name(..)');



    # now alter the local template hash to cover the various cases.

    if ( $parent_name eq 'section' ) {

        my $level = $node->findvalue('count(ancestor::section)');

        $template->{pre}  = "<h$level>";

        $template->{post} = "</h$level>";

        return 1;

    }

    elsif ( $parent_name =~ m/sect(\d+)$/ ) {

        my $level = $1;
        $template->{pre}  = "<h$level>";
        $ttemplate->{post} = "</h$level>";
         return 1;
    }
    else {
        return 0;
    }
};
```

Here, you grab the name of the current title element's parent element and examine that value to provide the context for your conditional processing. If the parent is a section element, you count the number of nested sections and use that to determine the level for the HTML header element that will be used to render the content. If the parent name matches the regular expression /sect(\d+)$/, the level of the header is defined by extracting the numeric value from the element name itself (i.e., sect1, sect2, etc.). Obviously, this does not cover the 22 possible cases in which a title element can appear, but it provides a good start that you can build on later, as needed.

You return different values from your title element's testcode subroutine, depending on the context. This is an important option, since the value returned from the testcode subroutine controls how the XPathScript processor reacts when a matching node is found. A return value of 1 indicates to the XPathScript processor that you want to process the current node and its children, while returning 0 states that you do *not* want to process the current node (nor its children). The following is the list of possible testcode return values:

1

The default behavior for all templates; returning 1 from the testcode subroutine tells the XPathScript processor to process the current node and descend into any child nodes for additional template matches.

-1

A return value of -1 signals the XPathScript engine to process the current node but skip all children and their descendants.

0

Returning 0 from any point in the testcode subroutine tells the XPathScript processor to skip the current node and its descendants altogether.

$string

If a string is returned from the testcode subroutine, it is expected to contain a valid XPath expression that will be used to select the next set of nodes to process.

Given the access that the testcode option provides to the node being selected and the variety of return codes that control what the XPathScript processor does *after* the associated subroutine has been executed, you can safely say that XPathScript's element-name-only template matching rule behavior is in no way inferior to XSLT's template matching rules (in which more sophisticated XPath expressions may be used to decide when a template is applied). The only difference is that, in XPathScript, any additional node properties are examined via the given testcode subroutine, not tested by the match rule itself.

6.2.2 The Special "Catch-All" Template

There is one exception to the rule that template matches can only be made against element names: the special "catch-all" template that is invoked by using a top-level key in the template hash with the name «*». The subkeys added to the catch-all template are invoked for every element node that does not already have an explicit entry in the template hash. Among other uses, this allows you to explicitly prune all elements from the current node that do not have an associated template rule.

```
<%

# Main template rules here . . .



# But block all unexpected elements

$t->{'*'}->{testcode} = sub { return 0; };

%>
```

If you want to be a little fancier, you can log the rogue elements in the Apache *error.log* for debugging during development:

```
<%

# Block and log all unexpected elements

$->{'*'}->{testcode} = sub {

    my ($node, $template) = @_;

    AxKit::Debug(10, "Unexpected element '" . $node->getName . "'

found during XPathScript transformation.");

    return 0;

};

%>
```

6.3 XPathScript Cookbook

Just as we concluded our look at XSLT with a few recipes that offer solutions to common tasks, we will do the same here with XPathScript. Since most XSLT tips are easy to re-create using XPathScript, I will not repeat the same recipes but will focus instead on things unique to XPathScript (or simply not possible with vanilla XSLT 1.0).

6.3.1 Accessing Client Request and Server Data

6.3.1.1 Problem

You need to access information about the current request (POSTed form data, cookies, etc.) or other parts of the Apache HTTP server API from within your stylesheet.

6.3.1.2 Solution

Use the Apache object that is passed as the first argument to the top level of every XPathScript stylesheet transformation.

```
<%

# at the top lexical level of your XPathScript stylesheet:

my $r = shift;

# $r now contains the same Apache object passed to mod_perl handler scripts.

%>
```

6.3.1.3 Discussion

One benefit of running XPathScript inside of AxKit is that each stylesheet can directly access the Apache server environment via the same Apache object that gives *mod_perl* its power and flexibility. This object can be used to examine (and in many cases, control) virtually every aspect of the Apache HTTP Server, from user-submitted form and query data to outgoing headers and host configurations.

6.3.1.4 Accessing form and query data

```
<%

use Apache::Request;

my $r = shift;

my $cgi = Apache::Request->instance($r);

%>

<html>

  <body>

    <p>

     You said that your favorite fish is <%= $cgi->param('fave_fish') %>

    </p>

  </body>

</html>
```

6.3.1.5 Setting cookies

```
<%
use Apache::Cookie;
my $r = shift;
my $out_cookie = Apache::Cookie->new( $r,
    -name => 'mycookie',
    -value => 'somevalue'
);
$out_cookie->bake;
%>
<html>
    <!--Stylesheet contents  -->
</html>
```

6.3.1.6 Redirecting the client

```
<%
use Apache::Constants qw(REDIRECT OK);
my $r = shift;

if ( $some_condition =  = 1 ) {
    $r->headers_out->set(Location => '/some/other.xml');
    $r->status(REDIRECT);
    $r->send_http_header;
}
%>
<html>
    <!--Default stylesheet contents  -->
</html>
```

6.3.2 Generating Fresh Dynamic Content

6.3.2.1 Problem

Your XPathScript stylesheet generates or transforms content based on runtime logic, but AxKit is serving the cached result of a previous transformation.

6.3.2.2 Solution

Pass a nonzero value to the *no_cache( )* method on the Apache object to turn off caching for the current resource:

```
<%

my $r = shift;

$r->no_cache( 1 );

%>

<html>

    <!--Default stylesheet contents  -->

</html>
```

6.3.2.3 Discussion

AxKit's aggressive default caching behavior helps keep it speedy and easy to set up for many common publishing cases, but sometimes, your stylesheets will need to generate or transform content based on data unavailable until runtime. In these cases, the stylesheet behaves as expected for the first request, but once the cache is created, the result of all transformations in the current processing chain is sent directly to the client, and the stylesheet containing the dynamic logic is not applied again until the source XML file (or one of the stylesheet documents) is changed.

Setting the AxNoCache configuration directive to On can do the trick, but setting it up on a resource-by-resource basis can be cumbersome. It gets even trickier if several possible transformation chains can be applied to a given resource and only some of them could benefit from turning caching off altogether.

The most direct way to ensure that your dynamic stylesheet is always applied is to simply pass a true value to $r->no_cache( ) from within your XPathScript stylesheet itself. Be aware that any resources transformed using that stylesheet will *never* be cached (and the entire transformation chain will be run for each request), but if you're generating or transforming content dynamically, that is probably what you want to happen in any case.

6.3.3 Importing Templates Dynamically

6.3.3.1 Problem

You want to apply different sets of template rules based on runtime data or aspects of document content beyond the root element name or DOCTYPE declaration.

6.3.3.2 Solution

Use a simple wrapper stylesheet containing a call to the *import_template( )* function, and generate the path to the imported stylesheet dynamically:

```
<%

my $r = shift;

$r->no_cache( 1 );



my $import_path = '/styles/myapp/';



# Select the import based on the presence of an <error>

# element anywhere in the source document.



if ( findvalue('//error') ) {

    $import_path  .= 'error.xps';
}
else {
    $import_path .= 'default.xps';
}

import_template($import_path)->( );
%>
<html>
    <%= apply_templates( ) %>
</html>
```

6.3.3.3 Discussion

The combination of AxKit's styling directives and StyleChooser plug-ins offers the ability to create chains of transformations, then select the appropriate chain to apply at request time based on a wide variety of environmental factors (the type of client making the connection, the root element name of the source XML document, etc.). However, sometimes there are cases in which explicitly defining (then selecting) a processing chain for all possible sets of conditions can get tedious.

Using the AxAddDynamicProcessor directive (that lets you write a Perl class that will generate the steps in the processing chain dynamically, at request time) offers a solution. However, it can seem like overkill for a lot of cases, and even then, you cannot readily access the content of the source XML document. Letting logic in the XPathScript stylesheet itself decide which set of imported templates to apply allows you to set up a single processing chain while still applying the appropriate styles in response to the environment.

6.3.4 Tokenizing Text into Elements

6.3.4.1 Problem

You have elements in your source XML document with text content that you want to tokenize into child elements or mixed content (containing both text data and child elements).

6.3.4.2 Solution

Use a template with a testcode subroutine that operates on the element's text children directly, and use simple Perl text-handling logic to create the new elements:

```
$t->{'my:element'}{testcode} = sub {
    my ($node, $t) = @_;

    foreach my $child ( $node->getChildNodes ) {
        if ( $child->getNodeType =  = TEXT_NODE ) {
            my $text = $child->getValue( );
            # Process $text as a string, adding pointy brackets as needed
            # to create tokenized  mixed content
            print $text;
        }
        else {
            # otherwise, apply templates manually to the child node

            apply_templates( $node );

        }

        # You have already processed this node and its children, so tell

        # the XPathScript processor not to.

        return 0;

}
```

6.3.4.3 Discussion

Carving up text nodes into tokenized mixed content, sometimes called *up-translation*, can be a tricky proposition (especially using XSLT), but it is not as crazy as it may sound. For example, you may be given documents that contain dates marked up as a date element containing a single text child, but it would simplify processing if the year, month, and day components were defined individually:

```
<!-- What you get -->

<date>2003-06-07</date>



<!-- What you wish you were getting -->

<date>

    <year>2003</year>

    <month>06</month>

    <day>07</day>

</date>
```

Here, the solution is to select the value of the date element, split that value on the «-» character using Perl's built-in *split* function, then print the appropriate textual representation of the new elements:

```
$t->{date}{testcode} = sub {

    my ($node, $t) = @_;

    my $date_string = $node->getValue( );

    my ($year, $month, $day) = spilt(/-/, $date_string);

    print "<date><year>$year</year><month>$month</month><day>$day</day></date>";

    return 0;

};
```

You must return 0 from the testcode subroutine, since you have already printed all the output you need for the current element, and you do not want the XPathScript processor to descend into the old text node and try to add it to the output. Be aware, too, that by taking control of the entire output and returning 0 from the testcode subroutine, you render the other template options (pre, post, prechild, showtag, etc.) useless for the current element. That is, by printing the output and returning 0 you are essentially saying to the XPathScript processor "Do not process this node. I want to do it myself," so the other rules are never applied.

Things get more interesting when the element in question may already have mixed content, and you need to make sure that the child nodes are processed in the typical way. In this case, you need to manually iterate over all the children of the current node and examine the node type of each child. You can then operate on the text nodes, as needed, while explicitly calling *apply_templates* for all other types of nodes.

Example 6-2 illustrates how to cope with the task of tokenizing character data when the elements being transformed already have a mixed content model. This stylesheet examines the text contents of all p elements and their children for the presence of a keyword passed in via a posted form or query string parameter named matchword. A simple Perl regular expression is used to find the matches. When one is found, it is replaced in the result by the same word wrapped in a span element whose style class renders that word in the browser in bold text with a yellow background.

Example 6-2. XPathScript stylesheet for highlighting keywords in XHTML

```
<%
use Apache::Request;
my $r = shift;
$r->no_cache( 1 );
my $cgi = Apache::Request->instance( $r );

sub tokenizer {
    my ($element, $t) = @_;

    if ( my $match_word = $cgi->param('matchword') ) {
        my $name = $element->getName( );

        print "<$name>";
        foreach my $node ( $element->getChildNodes ) {
            if ( $node->getNodeType( ) =  = TEXT_NODE ) {
                my $text = $node->getValue( );
                $text =~ s|\b($match_word)\b|<span class="matched">$1</span> |g;
                print $text;
            }
            else {
                print apply_templates( $node );
            }
        }
        print "</$name>";
        return 0;
    }
    else {
        $t->{showtag} = 1;       return 1;
    }
}

# apply the 'tokenizer' sub to all 'p' elements
# and their children
$t->{'*'}{testcode} = sub {
    my ( $node, $t ) = @_;
    if ( $node->findnodes('ancestor-or-self::p') ) {
       tokenizer( $node, $t );
    }
    else {
        return 1;
    }
};
%>
<html>
  <head>
    <style>
    span.matched { background: yellow; font-weight: bold;}
    </style>
  </head>
  <body>
  <form action="highlight.xml" method="GET">
    <input name="matchword" type="text">
    <input type="submit">
  </form>
  <%= apply_templates( ); %>
  </body>
</html>
```

Certainly, XPathScript is not to everyone's taste. For some, its visible reliance on Perl and the fact that it does not use an XML syntax are enough to send them scrambling for the hills. For others, having an XML syntax for their processing tools (in addition to the content source) is going too far, and those very same properties are what make XPathScript desirable. Reality is that one size never fits all. The fact that AxKit supports a variety of transformation languages is part of what makes it a viable, professional development environment.

Similarly, do not let the fact that XSLT and XPathScript were featured in the last two chapters dissuade you from investigating the other options that AxKit makes available. The goal here has been simply to introduce two of the more popular choices in an effort to give you the means to be productive with AxKit as quickly as possible, not to recommend one technology over another. For example, you may find that one of the other generic transformation tools such as SAX filter chains (Apache::AxKit::Language SAXMachines) or Petal (Apache::AxKit::Language::Petal), is better suited to your needs. In any case, remember that picking one transformative Language module does not mean abandoning all others—AxKit allows you to pick the tool that best suits the task, even mixing different languages within a single processing chain.

Chapter 6. Transforming XML Content with XPathScript

XSLT is a good solution for many cases, but it is certainly not the only game in town when it comes to transforming XML documents. XSLT does a fantastic job of performing recursive transformations of tree-shaped data structures based on declarative rules, but it's not a general-purpose programming language, and the solutions to some common web-publishing tasks (most notably, anything involving text processing) can seem cumbersome and obtuse. Ironically, the people who seem to have the hardest time picking up XSLT are more experienced web developers with a strong history in a language such as Perl or Python. It's not that XSLT is bad or wrong; it just relies on patterns unfamiliar to many web coders, and many approaches that work just fine in Perl simply do not fit into XSLT's model.

Created expressly to bridge the gap between "just print it, already" web-programming techniques and the tree-based operations found in most XML processing tools, the XPathScript language seeks to offer the best of both worlds—powerful declarative rule-based transformations such as those available in XSLT, combined with full access to the flexibility of Perl and its many modules.

Here are some reasons why you may want to use XPathScript:

- You know Perl and do not want to learn XSLT just to work with AxKit.
- Your stylesheets would benefit from dynamic transformations based on external libraries, fine-grained access to the Apache server environment, database queries, etc.
- Using XSLT is working out fine for the most part, but you want to pre- or postprocess your documents in a Perlish way to simplify or avoid certain hairy transformations.

Essentially, if you have a strong background in Perl development and the prospect of learning XSLT to transform XML content in AxKit seems undesirable—or if the types of transformations you are doing seem beyond the reasonable reach of XSLT— then XPathScript is probably the tool for you.

7.1 Introduction to eXtensible Server Pages

Originally created by the developers of AxKit's sister Apache project, Cocoon, eXtensible Server Pages (XSP) offers an XML-centric implementation of the Server Pages model. Unlike XSLT or XPathScript, an XSP document is not a separate stylesheet that is applied to an XML document to transform it; rather, XSP is concerned solely with *generating* content. Literal markup is mixed with functional code and elements from the XSP grammar within a single document to create an XML instance dynamically when that document is passed through the XSP processor. In AxKit's implementation, the embedded programming language is Perl.

As with AxKit's other Language modules, support for XSP must be explicitly enabled using the AxAddStyleMap configuration directive:

```
AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP
```

XSP diverges from the norm a bit in how documents are associated with the XSP processor. Unlike XSLT or XPathScript, XSP is self-contained, and there is no external stylesheet to apply. This means that when setting up the style processing directives for XSP, you use the literal string NULL, whereas if you use another Language module, you would typically put the path to the stylesheet.

```
# The familiar pattern, using XSLT
AxAddProcessor text/xsl /styles/somestyle.xsl

# In XSP there is no stylesheet, so the
# string NULL is used instead
AxAddProcessor application/x-xsp NULL

# The same, but as a conditional processor, based
# on the top-level element name and its namespace URI
AxAddRootProcessor application/x-xsp NULL {http://apache.org/xsp/core/v1}page
```

7.1.1 XSP Basics

XSP's basic syntactic requirements are very simple: every valid XSP document must be a well-formed XML document, and all XSP elements must be bound to the namespace URI http://apache.org/xsp/core/v1 (binding that URI to the prefix xsp is conventional, but certainly not required). In contrast to XPathScript, Active Server Pages, PHP, and similar technologies that use special pseudo tags (such as <% . . . %>, for example) as delimiters to separate code from literal markup, XSP uses an XML application grammar that separates literal and processed output through the use of specific XML elements bound to the XSP namespace. For example, the xsp:logic element creates a block that can contain any block of free-form Perl code, while the xsp:expr element interpolates a single expression into its literal result. The following shows a minimal, valid XSP document that inserts the current time into the content via an xsp:expr element:

```
<?xml version="1.0"?>
<application>
  <time>
     <xsp:expr xmlns:xsp="http://apache.org/xsp/core/v1">scalar localtime( )</xsp:expr>.
  </time>
</application>
```

Although XSP elements and code can be embedded into any XML document and passed to the XSP processor, an XSP page more commonly contains: a top-level xsp:page element containing optional xsp:structure and xsp:logic elements, and a required non-XSP element that becomes the top-level element of the result generated by the XSP processor (this is often called the *user-root* element). The following illustrates this basic structure:

```
<?xml version="1.0"?>

<!--  The  top-level <xsp:page> element, bound to the XSP namespace URI -->

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">



   <!--  optional <xsp:structure> used to import external Perl modules for use

in the current page-->

   <xsp:structure>

      <xsp:import>Some::Perl::Module</xsp:import>

   </xsp:structure>



   <!-- class-level logic (subroutines, global initialization, etc.) -->

   <xsp:logic>

      sub now {

         return scalar localtime( );

      }

   </xsp:logic>



   <!-- The following <application> element is the user-root

     that will become the top-level element after processing -->

   <application>

      <time>

         <xsp:expr>now( )</xsp:expr>

      </time>

   </application>

</xsp:page>
```

The following shows the result returned from both of the above examples:

```
<?xml version="1.0"?>

<application>

  <time>Thu Feb 12 20:26:11 2004</time>

</application>
```

Before we dig further into the details of XSP's syntax, it is important to note that the most effective use of XSP extends the Server Pages model beyond the code-mixed-with-HTML mess that may have driven you to look for an alternative environment such as AxKit in the first place. Yes, XSP allows and even encourages the mixture of XML markup and Perl code, but this is not the same as having code hardwired to generate *presentational* markup (which is arguably the true weakness found in most uses of the Server Pages technologies, not simply the fact that code and markup appear in the same document). To get the maximum benefit from XSP and AxKit, it is best to make sure that the markup that your XSP pages generate conforms to a grammar that best reflects the semantics and state of the application, irrespective of how that content may be presented. As long as your XSP-based applications meet this criteria, the syntactic details of *how* that data may be generated become solely a matter of personal taste.

For many web developers, the XML-influenced approach to generating dynamic content requires a little bit of mental adjustment. One tends to think of generating *pages* or *screens*, rather than constructing semantically meaningful data structures. It helps to remember, though, that the result generated by the XSP processor is really the *beginning* of the processing chain, not the end. In most cases, at least one transformative style will be applied to the XML created, and it is the stylesheet's job to transform the generated data into a usable interface. Therefore, you are free to generate simpler markup that reflects only the application's state and data structures.

Now that this conceptual foundation is laid, let's get to the business of introducing XSP. First, here is a summary of the grammar as a whole:

xsp:page

The optional top-level element for an XSP document. If present, it must be the top-level element of the document.

xsp:structure

A wrapper element for one or more xsp:import elements. May appear only as a child of a top-level xsp:page element and before the opening tag of the document's user-root element.

xsp:import

The text content of this element is expected to contain the name of a Perl class to be loaded during processing (similar to Perl's *use* function). This element may only appear as a child of the xsp:structure element.

xsp:logic

The content of this element is expected to be a free-form Perl block that is evaluated during processing.

xsp:expr

The content of this element is expected to be a Perl expression that is interpolated during processing. The result is added to the output as a literal string.

xsp:content

The content of this element is expected to be a well-formed XML fragment and is added, as is, to the resulting output. Although it may appear anywhere inside the user-root element, it most often appears as the child of an xsp:logic element, as a convenient way to escape out of the code to generate markup.

xsp:element

Creates an XML element whose name is defined by the required name attribute.

xsp:comment

The contents of this element will be wrapped in an XML comment in the resulting output.

xsp:pi

Will be expanded into an XML processing instruction, whose target is defined by the required target attribute.

XSP's grammar is quite small—just nine elements—but these elements, plus Perl's conditional statements, loops, and other logical constructs, can be combined to create very sophisticated XML content.

7.1.1.1 Generating content

First, it is important to keep in mind that all non-XSP elements, not otherwise skipped over based on conditional logic, are passed through the XSP processor verbatim into the generated result:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

<application>

    <user-info/>

</application>

</xsp:page>
```

Running this through the XSP processor gives the following predictable result:

```
<application>

    <user-info/>

</application>
```

Free-form Perl code can be embedded any place inside an XSP document using the xsp:logic element. At the same time, single elements or well-balanced chunks can be generated from within those logic blocks by wrapping the output in an xsp:content element. Also, the xsp:expr element can be used to interpolate a simple Perl scalar expression into a literal string:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

<application>

    <user-info>

    <xsp:logic>

        my $connection = $r->connection;

        my $ip = $connection->remote_ip;


        <xsp:content>

            <ip-address><xsp:expr>$ip</xsp:expr></ip-address>

        </xsp:content>


        if ( $ip =~ /^192\./ ) {

            <xsp:content>

              <is-local>true</is-local>

            </xsp:content>

        }

        else {

            <xsp:content>

              <is-local>false</is-local>

            </xsp:content>

        }
    </xsp:logic>

    </user-info>

</application>

</xsp:page>
```

In most cases, the XSP processor is smart enough to tell the difference between Perl code and literal markup that is meant to be part of the generated result. This makes the xsp:content element optional. Therefore, you could trim the previous example into the following and get the same result:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

<application>

    <user-info>

    <xsp:logic>

        my $connection = $r->connection;

        my $ip = $connection->remote_ip;


        <ip-address><xsp:expr>$ip</xsp:expr></ip-address>


        if ( $ip =~ /^192\./ ) {

              <is-local>true</is-local>

        }

        else {

              <is-local>false</is-local>

        }

    </xsp:logic>

    </user-info>

</application>

</xsp:page>
```

There are a few things to remember about xsp:logic elements. First, any logic block that appears *outside* of the user-root element is considered a special, global block that the XSP processor only interprets *once*. This global section is intended to provide a handy place to declare any subroutines or constant variables that your XSP page may need, while saving the overhead associated with reevaluating the code for every request. This provides a performance boost, but the behavior of the special, global case can sometimes snare the unwary. The following two XSP pages exemplify a common trap:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

    <!-- logic block outside of the user-root -->

    <xsp:logic>

        my $time = scalar localtime( );
    </xsp:logic>

<application>

    <time><xsp:expr>$time</xsp:expr></time>

</application>

</xsp:page>
```

The $time scalar variable is set in an xsp:logic block outside of the user-root application element—that is, in the special, "global" section. Compare that with this very similar example:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

<application>

    <!-- the same block, but inside the user-root -->

    <xsp:logic>

        my $time = scalar localtime( );

    </xsp:logic>

    <time><xsp:expr>$time</xsp:expr></time>

</application>

</xsp:page>
```

The two appear identical, but in the second example, the $time scalar is set in a logic block *within* the user-root element. You may expect that the two pages would return the same results, but remember that any logic block that appears outside of the user-root element is a special case that is only evaluated once. So the result returned from the first page shows the same time string no matter how many times the page is requested, while the second updates the time string for each request.

However, it is perfectly safe to put subroutines in the global block. In this case, it doesn't matter that the code is only evaluated once. The subroutine still *executes* at request time. So if you don't want to put the xsp:logic element inside the user-root, the following still works as you'd expect:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">

    <!-- logic block outside of the user-root -->

    <xsp:logic>

        sub now { return scalar localtime( ); }

    </xsp:logic>

<application>

    <time><xsp:expr>now( )</xsp:expr></time>

</application>

</xsp:page>
```

The second thing to keep in mind when creating inline xsp:logic blocks concerns possible syntactic conflicts between Perl and XML. Certain common Perl expressions can cause XML well-formedness errors unless they are properly escaped. The specific characters you have to watch out for are «&» and «<».

```
<xsp:logic>

    # Parser sees '<' and thinks it's the beginning of a new XML element
    if ( $this_number < $that_number ) {  }

    my $here_doc = <<"TARGET";


    # Parser sees '&' and thinks it's the beginning of an XML entity reference

    if ( $this_condition && $that_condition ) {  }

    my $val = &sone_function( );

</logic>
```

The preferred solution here is to wrap the contents of the xsp:logic element that contain the conflicting characters in a CDATA (character data) section that the XML parser passes over without trying to parse it:

```
<xsp:logic><![CDATA[

   # You are now free to use & and < to your heart's content


]]><xsp:logic>
```

Using a CDATA section to tell the XML parser to skip the contents of the logic block also tells the parser to ignore any XSP elements or literal output, as well, so be sure to break out of the CDATA if you want to return content.

New XML elements may also be generated through use of the xsp:element element. The string passed to this element's required name attribute becomes the name of the newly generated element. The value set for the name attribute is usually a literal string, but you can also pass in a Perl scalar expression by wrapping it in curly braces. This is similar to XSLT's attribute value templates and offers the ability to produce elements whose names are generated programmatically at request time.

```
<params>

  <xsp:logic>

  my %query_params = $r->args;

  foreach my $key ( keys( %query_params )) {

      <xsp:element name="{$key}"><xsp:expr>$query_params{$key}</xsp:expr></xsp:element>

  }

  </xsp:logic>

</params>
```

Similarly, XML attributes can be generated via the xsp:attribute element. This element's required name attribute becomes the name of the newly created attribute, while the contents of this element become the attribute's value.

```
<params>

  <xsp:logic>

  my %query_params = $r->args;

  foreach my $key ( keys( %query_params )) {

      <param>

        <xsp:attribute name="name"><xsp:expr>$key</xsp:expr></xsp:attribute>

        <xsp:attribute name="value"><xsp:expr>$query_params{$key}</xsp:expr></xsp:attribute>
```

```
     </param>

  }

  </xsp:logic>

</params>
```

Finally, comments and processing instructions can be generated using the xsp:comment and xsp:pi elements, respectively. Generating comments can be especially useful when debugging your XSP while it is still in development. Rather than dumping information to the host's error log, you can simply produce a comment for nonfatal but unexpected errors, and the result appears directly in the output in a way that's easily distinguished, visibly, from the rest of the content:

```
<xsp:logic>

    if ($var eq 'this' ) {

        <this><xsp:expr>$var</xsp:expr></this>

     }

     else {

         <xsp:comment>Was expecting $var to be 'this' but got <xsp:expr>$var</xsp:expr

> instead.</xsp:comment>

     }

</xsp:logic>
```

7.1.1.2 Using Perl modules in XSP pages

Perl's popularity as a programming language has as much to do with the army of useful, open source extension modules freely available from the Comprehensive Perl Archive Network (CPAN) as it does with any particular feature of the language itself. Once installed (details about installing modules from CPAN can be found by running *perldoc CPAN* at a shell prompt on any machine on which Perl is installed), these modules can be used from within your XSP pages by adding an xsp:structure element as a direct child of the top-level xsp:page element, then adding an xsp:import element containing the name of the Perl package for each module that you want to use in your page:

```
<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">


  <xsp:structure>

                    <xsp:import>Geo::IP</xsp:import>

  </xsp:structure>


<application>

    <user-info>

    <xsp:logic>

        my $connection = $r->connection;

        my $ip = $connection->remote_ip;


                      my $mapper = Geo::IP->new( );
        <ip-address><xsp:expr>$ip</xsp:expr></ip-address>

        <country><xsp:expr>$mapper->country_name_by_addr($ip)</xsp:expr></country>

    </xsp:logic>

    </user-info>

</application>

</xsp:page>
```

You should now have a pretty good idea about XSP's page-level syntax and structure. There's more to the story, though. From what you have learned so far, you can generate application content quickly, but each page exists only unto itself, and the solutions are not reusable. Fortunately, AxKit's XSP implementation allows you to extend this basic framework to include your own custom application grammar extensions that can be reused in any number of pages to meet a variety of needs. These extensions are called *tag libraries*.

7.1.2 XSP Tag Libraries

XSP tag libraries (or taglibs, for short) provide a means to extend XSP's functionality by mapping XML elements in a specific, custom XML grammar to Perl code that expands and replaces those elements with other markup generated dynamically, performs a programmatic task behind the scenes, or most often, both. A tag library's elements are distinguished from literal output and elements in the XSP core grammar by binding those elements to a specific XML namespace. The following shows a simple XSP page that employs the Util tag library (available on CPAN) to return the current time:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1"

       xmlns:util="http://apache.org/xsp/util/v1"


  <application>

      <time>

        <util:time/>

      </time>

  </application>

</xsp:page>
```

XSP taglibs can be implemented in one of two ways: as a special XSLT, XPathScript, or other stylesheet (called a *logicsheet*) used to preprocess the content, expanding the elements in the taglib grammar into the XSP elements and Perl codes that implement the taglib's behavior before the document is passed on the XSP processor, or more commonly, as a special Perl module registered with AxKit and used to extend the functionality of the XSP processor itself. You learn how to create both types of taglibs later in this chapter. For now, let's focus on some existing module-based taglibs and how they can be used to extend your XSP pages. The following explains just a few of the more popular or interesting XSP taglibs available for use in your XSP applications:

AxKit::XSP::Wiki

: A complete Wiki application implemented in XSP. Uses POD as the page syntax.

AxKit::XSP::WebUtils

: A general-purpose utility class that provides access to request and server data. Among other things, you can get or set HTTP headers, issue client redirects, examine whether the current request is using secure HTTP connections, and much more.

AxKit::XSP::Sendmail

Send email using this simple XSP interface to Perl's popular Mail::Sendmail module.

AxKit::XSP::ESQL

Feature-rich taglib for selecting data from any relational database supported by Perl's DBI.

AxKit::XSP::Util

Offers several utility functions for including content from local files, remote URI, and interpolated expressions, as well as the ability to get the current date and time in a variety of formats.

AxKit::XSP::LDAP

An easy-to-use XSP interface for retrieving data from LDAP servers.

AxKit::XSP::PerForm

Robust XSP helper class for creating and validating data entry applications.

AxKit::XSP::Swish

XSP interface to the popular site search/indexing tool, Swish-e.

There are many benefits of using XSP taglibs, but the most important is that they provide XSP developers with the ability to create generic, reusable libraries that implement common features or solve common problems.

7.1.2.1 Installing module-based XSP taglibs

Installing module-based XSP taglibs is exactly the same as installing any Perl module. If the taglib you want to install is available from CPAN, installation can be achieved quickly and painlessly using the CPAN shell that installs with Perl itself. Simply ask your systems administrator to install the taglib. (Be sure to provide her with the exact name of the taglib module you wish to install.) Or if you have permission, become root (superuser) and enter the following at a shell prompt:

```
perl -MCPAN -e shell

install Some::XSP::Taglib
```

Once the module is installed, you must first register the taglib with AxKit before you can start using it with your XSP applications. This is achieved by adding an AxAddXSPTaglib configuration directive to your *httpd.conf* or other Apache configuration file and passing the Perl package name of the taglib module as the sole argument. The following registers the Param and ESQL taglibs with AxKit:

```
AxAddXSPTaglib AxKit::XSP::Param

AxAddXSPTaglib AxKit::XSP::ESQL
```

Adding the taglib modules in this way does two things: it configures AxKit to load the module code (similar to Perl's built-in use statement), and it registers the unique XML namespace associated with that taglib grammar so that the XSP processor knows to dispatch the processing of elements in that namespace to the taglib's Perl package and not to simply pass them through as literal content.

Once a taglib module is registered, you may use the elements from its grammar in any XSP page. You need only to make sure to bind those elements to the XML namespace URI associated with that tag library; the XSP processor handles the rest.

The XSP taglib modules are already available from CPAN and cover an interesting set of cases. They are fantastic for adding reusable features to your own applications with a minimum of effort. However, there will surely be cases when what's out there does not meet your needs, and you should implement your own taglibs. Now that you have an idea about what XSP taglibs are and how they are used, let's examine how to create your own custom tag libraries.

7.1.2.2 Writing logicsheet taglibs

Module-based XSP taglibs (which we will examine shortly) are by far more common, but we would be remiss not to touch on the alternative, *logicsheet taglibs*, first. As I mentioned earlier, a logicsheet taglib is an XSLT, XPathScript, or other stylesheet applied to the source XSP document before it is handed to the XSP processor. During this transformation, elements in the tag library's grammar are expanded and replaced by the core XSP elements and Perl code that are required to implement the taglib's behavior.

Suppose that while designing our next XSP application, you discover that certain bits of environmental data are needed throughout each stage in the application. You decide to encapsulate access to that information in a tag library rather than duplicate the code in each XSP document. Specifically, you need to access the server's environment variable (%ENV), you need to know whether the current user has logged in (based on the presence of an HTTP cookie), and you want a list of links to simplify the creation of the ubiquitous "breadcrumb" navigation bar. Given these requirements, you could define your taglib grammar in the following way:

> The XML namespace URI for this grammar is http://localhost/xsp/myapp, and the preferred namespace prefix is myapp.

myapp:env

> This element is replaced by an env element containing a list of field elements, each containing name and value elements that reflect the name and value of each field in the %ENV hash.

myapp:breadcrumb

> This element is replaced by a breadcrumb element that contains an ordered list of link elements. Each link element represents a link in the breadcrumb chain and contains both an href attribute containing the URI of that step and a title attribute that may be used when creating a hyperlink for that step.

myapp:logincheck

> This element is replaced by a logged-in element whose text contents are either true or false depending on the presence or absence of an HTTP cookie. If the value of the logged-in element is true, a username element containing the current user's name will also appear.

An XSP page using this taglib grammar may look something like Example 7-1.

Example 7-1. minimal_myapp.xsp

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1"

          xmlns:myapp="http://localhost/myapp">

<xsp:structure>

   <xsp:import>Apache::Cookie</xsp:import>

</xsp:structure>

<application>

  <meta>

    <myapp:breadcrumb/>

    <myapp:env/>

    <user>

      <myapp:logincheck/>
    </user>
  </meta>
</application>
</xsp:page>
```

Now you must create the logicsheet that expands the elements in the MyApp grammar into the XSP elements and Perl code that actually make the taglib work as you expect. Example 7-2 shows one possible way to do this, using XSLT to implement the logicsheet.

Example 7-2. myapp.xsl : the MyApp taglib as a preprocessed XSLT logicsheet

```
<?xml version="1.0"?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsp="http://apache.org/xsp/core/v1"
  xmlns:myapp="http://localhost/xsp/myapp"
>

<xsl:template match="myapp:env">
<env>
  <xsp:logic>
    foreach my $key (keys(%ENV)) {
        <field>
          <name><xsp:expr>$key</xsp:expr></name>
          <value><xsp:expr>$ENV{$key}</xsp:expr></value>
        </field>
    }
  </xsp:logic>
</env>
</xsl:template>

<xsl:template match="myapp:breadcrumb">
  <breadcrumb>
  <xsp:logic>
    my $path ='/';

    <link title="home" href="/"/>

    my @steps = split '/', $r->uri;
    for ( my $i = 0; @steps > $i; $i++ ) {
        my $step = $steps[$i];
        next unless length $step;
        $path .= $step;
        $path .= '/' unless $i =  = $#steps;

        <link>
          <xsp:attribute name="title">
             <xsp:expr>$step</xsp:expr>
          </xsp:attribute>

          <xsp:attribute name="href">
              <xsp:expr>$path</xsp:expr>
          </xsp:attribute>
       </link>
    }
  </xsp:logic>
  </breadcrumb>
</xsl:template>

<xsl:template match="myapp:logincheck">
<xsp:logic>
    my $cookie = Apache::Cookie->fetch;
    if ( defined( $cookie->{'username'} ))  {
        my $username = $cookie->{'username'}->value;
        <logged-in>true</logged-in>
        <username><xsp:expr>$username</xsp:expr></username>
    }
    else {
        <logged-in>false</logged-in>
    }
</xsp:logic>
</xsl:template>

<xsl:template match="*">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates />
  </xsl:copy>
</xsl:template>
```

This logicsheet taglib is really just a plain XSLT stylesheet that replaces the elements in the MyApp grammar with the code, XSP elements and literal markup required to implement the taglib. Obviously, for this to work, you must apply the *myapp.xsl* stylesheet to the source document:

```
<FIles minimal_myapp.xsp>

  AxAddProcessor text/xsl /styles/myapp.xsl

</Files>
```

With this snippet added to your configuration file, a request for the *minimal_myapp.xsp* document gives the following result:

```
<?xml version="1.0"?>

<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1" xmlns:myapp="http://localhost/myapp">

<xsp:structure>

   <xsp:import>Apache::Cookie</xsp:import>

</xsp:structure>

<application>

  <meta>

    <breadcrumb><xsp:logic>

    my $path ='/';


    <link title="home" href="/"/>


    my @steps = split '/', $r->uri;

    for ( my $i = 0; @steps > $i; $i++ ) {

        my $step = $steps[$i];

        next unless length $step;

        $path .= $step;

        $path .= '/' unless $i =  = $#steps;


        <link>

            <xsp:attribute name="title">

                <xsp:expr>$step</xsp:expr>

            </xsp:attribute>

            <xsp:attribute name="href">

                <xsp:expr>$path</xsp:expr>

            </xsp:attribute>

        </link>


    }
  </xsp:logic></breadcrumb>
    <env><xsp:logic>
    foreach my $key (keys(%ENV)) {
        <field>
            <name><xsp:expr>$key</xsp:expr></name>
            <value><xsp:expr>$ENV{$key}</xsp:expr></value>
        </field>
    }
  </xsp:logic></env>
    <user>
      <xsp:logic>

    my $cookie = Apache::Cookie->fetch;
    if ( defined( $cookie->{'myapp_username'} ))  {
        my $username = $cookie->{'myapp_username'}->value;
        <logged-in>true</logged-in>
        <username><xsp:expr>$username</xsp:expr></username>
    }
    else {
        <logged-in>false</logged-in>
    }
</xsp:logic>
    </user>
  </meta>
</application>
</xsp:page>
```

As expected, the elements from the MyApp grammar are replaced and all other content is copied through verbatim. You only need to configure AxKit to apply the XSP processor to this intermediate step to get the final result:

```
<Files minimal_myapp.xsp>
  # Preprocess using the XSLT logicsheet
  AxAddProcessor text/xsl /styles/myapp.xsl

  # And send the result to the XSP processor
  AxAddProcessor application/x-xsp NULL
</Files>
```

This simple two-step processing chain delivers the final, desired result:

```
<?xml version="1.0" encoding="UTF-8"?>
<application>
    <meta>
        <breadcrumb>
          <link title="home" href="/"/>
          <link title="axkitbook" href="/axkitbook/"/>
          <link title="samples" href="/axkitbook/samples/"/>
          <link title="chapt07" href="/axkitbook/samples/chapt07/"/>
          <link title="logicsheet.xml" href="/axkitbook/samples/chapt07/minimal_myapp"/>
        </breadcrumb>
        <env>
          <field>
           <name>PATH_INFO</name>
           <value/>
         </field>
         <field>
           <name>PATH</name>
           <value>/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/kip/bin</value>
        </field>
        <field>
           <name>GATEWAY_INTERFACE</name>
           <value>CGI-Perl/1.1</value>
        </field>
       <field>
          <name>MOD_PERL</name>
          <value>mod_perl/1.29</value>
       </field>
    </env>
    <user>
      <logged-in>true</logged-in>
      <username>ubu</username>
    </user>
  </meta>
</application>
```

If this were a real-world XSP page, you would certainly include more that just the bits of metadata that you have here. Also, you would probably have a content or state element that contains the data associated with the current state of whatever application you were implementing, but this example shows enough of the basics to get you up and running.

7.1.2.3 Writing module-based taglibs using TaglibHelper

Logicsheet-based taglibs such as the one you just created can be handy, but in practice, most XSP tag libraries are implemented as Perl modules that are then registered with the XSP processor. There are several benefits to using module-based taglibs rather than logicsheets. First, module-based taglibs do not require the XSP source to be preprocessed before execution; this not only saves the overhead associated with adding an additional transformation, but it often greatly simplifies the style processor configuration needed to serve the XSP pages throughout a given site. Also, module-based taglibs offer additional performance, since the Perl code that implements the taglib functions will be cached in memory, so the XSP processor is able to dispatch the handling of the taglib elements to the external module as quickly as it processes the elements in the core XSP grammar.

While it is possible to write taglib modules that interact directly with the low-level components of AxKit's XSP engine, this approach is repetitive and error-prone, and requires intimate knowledge of the XSP processor's internals. It is usually better to use one of the helper modules available to streamline and simplify the process of writing custom tag libraries. In Example 7-3, the MyApp grammar from the logicsheet sample is reimplemented as a Perl module using Steve Willer's Apache::AxKit::XSP::TaglibHelper module that ships with the core AxKit distribution.

Example 7-3. MyApp.pm

```
package TaglibHelper::MyApp;


use strict;

use Apache::AxKit::Language::XSP::TaglibHelper;

use Apache::Cookie;


use vars qw( @ISA $NS @EXPORT_TAGLIB );

@ISA = qw( Apache::AxKit::Language::XSP::TaglibHelper );

$NS = 'http://localhost/xsp/myapp';


@EXPORT_TAGLIB = (

    'logincheck( )',

    'env( ):listtag=env:itemtag=field',

    'breadcrumb( ):as_xml=1',

);


sub logincheck {

    my $cookie = Apache::Cookie->fetch;

    my $out = {  };


    if ( defined( $cookie->{'username'} ))  {

        my $username = $cookie->{'username'}->value;

        $out->{'logged-in'} = 'true';

        $out->{username} = $username;

    }

    else {

        $out->{'logged-in'} = 'false';
    }
    return $out;
}

sub env {
    my @out = ( );
    foreach ( keys( %ENV ) ) {
        push @out, { name => $_, value => $ENV{$_}};
    }
    return \@out;
}

sub breadcrumb {
    my $out = '<breadcrumb>';
    $out .=   '<link title="home" href="/"/>';

    my $path ='/';
    my $r = AxKit::Apache->request( );

    my @steps = split '/', $r->uri;
    for ( my $i = 0; @steps > $i; $i++ ) {
        my $step = $steps[$i];
        next unless length $step;
        $out .= qq|<link title="$step" href="$path"/>|;
    }

    $out .= '</breadcrumb>';
    return $out;
}
1;
```

There are several things to note here. First, the module is a subclass of the TaglibHelper class itself. This allows taglib module authors to implement only the functions required to react to the tags in that specific taglib while hiding the tedious low-level processing. Next, TaglibHelper works by allowing subclass authors to write simple Perl subroutines whose names match the local name of the elements in the taglib's grammar. Each time a given element from the taglib being implemented is encountered by the XSP processor, the matching subroutine is called.

The arguments passed to the module's subroutines and the way the values returned from those subroutines will be processed are determined by the function specifications passed to the `@EXPORT_TAGLIB` array. Each element in this array takes the form of a string that follows the pattern illustrated here:

```
tagname([argument specification])[: additional options ]
```

While `tagname` is the local name (unprefixed) of the taglib element to match, `argument specifications` is an optional comma-separated list of any child elements of the taglib element that should be passed as arguments to the taglib subroutine, and additional options provide additional information about how the data returned from the taglib subroutine is processed and included in the result of the XSP process. Look back at the @EXPORT_TAGLIB array for your MyApp taglib module:

```
@EXPORT_TAGLIB = (

    'logincheck( )',

    'env( ):listtag=env:itemtag=field',

    'breadcrumb( ):as_xml=1',

);
```

This tells the TaglibHelper parent class to look for three elements from the taglib's namespace, logincheck, env, and breadcrumb. It further indicates that the *logincheck* subroutine expects no arguments (note the empty argument specification) and that the subroutine returns a simple Perl data structure that should be serialized to XML. The specification for the env subroutine shows that it too expects no arguments, that it will return a list reference whose entries should be named field (itemtag=field), and that the entire list should be wrapped in an env element (listtag=env). Finally, the specification for the breadcrumb indicates that it also expects no arguments and that the result returned is a well-balanced chunk of XML that should be parsed and included in the final result (as_xml=1).

With this module installed, you only need to register it by name via the AxAddXSPTaglib directive. You can begin using the tags from the MyApp grammar in your XSP pages without having to preprocess them using the logicsheet you created earlier.

```
# Load and register the taglib module

AxAddXSPTaglib TaglibHelper::MyApp



<Files minimal_myapp.xsp>

  # No need to preprocess, just send the source directly to the XSP processor

  AxAddProcessor application/x-xsp NULL

</Files>
```

The result returned from a request to *minimal_myapp.xsp* using your new taglib module is exactly the same as the result returned for the previous logicsheet taglib, so I will not duplicate the output here.

In the same way that Perl's many extension modules increase the power and value of the language as a whole by providing out-of-the-box solutions for common tasks, the judicious use of XSP tag libraries can add significant value to your XSP applications by reducing common application features to a handful of editor-friendly XML elements. If you are serious about building XSP applications with AxKit, I strongly suggest spending the time to learn the ins and outs of the TaglibHelper class or one of the other similar classes (SimpleTaglib or ObjectTaglib) that seek to streamline the process of writing custom XSP tag libraries—the time spent will more than pay for itself in the long run.

7.1.3 XSP Debugging Tips

When you are first starting out with XSP, it can be hard to track down precisely where something went wrong with your page or taglib code. The reason is that, to make XSP pages fast, flexible, and cache-friendly, AxKit's XSP engine is quite complex. (It's actually a SAX Handler that dispatches events to the core event handlers and various taglib modules while dynamically constructing an intermediate Perl class that uses the DOM interface to generate the results!) A lot of advanced Perl magic happens behind the scenes. There are, however, several AxKit configuration directives that you can use to make debugging your XSP applications more straightforward.

First, be sure to set the AxDebugLevel directive to maximum (10). This dumps copious amounts of information about what AxKit is doing behind the scenes to your web host's error log. Next, be sure that the AxStackTrace directive is set to On. This causes the AxKit error handling mechanism to produce a deep stack trace in the case of a fatal exception.

Also, be sure to set up the AxErrorStylesheet directive properly. As with the style processing directives, this option accepts the MIME type associated with one of the Language processing modules and the path to a stylesheet that will be applied to the XML error document that AxKit produces if this directive is present. For example, the following configures AxKit to apply the */styles/axkit_error.xps* stylesheet to the generated XML stack trace in the case of a fatal processing error:

```
AxErrorStylesheet application/x-xpathscript /styles/axkit_error.xps
```

This sends the transformed version of the Perl error and stack trace to the requesting client instead of the disappointing default 500 Internal Server Error that Apache usually offers. This saves you development time by obviating the need to look at the error log every time something blows up. See the entry about the AxErrorStylesheet directive in Appendix A for details about the XML that AxKit produces in these cases.

Finally, AxKit offers the helpful AxTraceIntermediate directive explicitly to help debug XSP pages and complex processing chains. This directive takes the path to a directory as its sole argument. Then, when a document is requested, AxKit generates a series of files in that directory (with the file extensions 0-n)—one for each transformation in the processing chain. If the chain includes an XSP process, a special *.XSP* file is generated that shows the intermediate Perl class that AxKit generates to implement the XSP page. This invaluable tool can save hours of groping for subtle bugs in your XSP code. To ensure that the results are easy to read, combine this directive with the AxDebugTidy On directive to format the intermediate files for better readability.

A word of caution, though—with the exception of the AxErrorStylesheet directive (which is good to use in any case), each directive listed here has a negative impact on AxKit's performance and should, therefore, only be used in a development environment, not on a production server.

7.2 Other Dynamic XML Techniques

While XSP is the dominant dynamic XML tool, there are other options, which may be appropriate in some cases.

7.2.1 Aggregate Data URIs

An *aggregate data URI* is simply a way to combine content from more than one source and present it as a single URI-addressable resource. While the uses for this technique are many, the most obvious and intuitive use is building collections of metadata about all or some documents on a given site. Suppose you are publishing an online newsletter. In each publishing period, you publish a few articles while archiving the old ones. Obviously, one feature that such a site needs is an article index—you do not want the older content to be forgotten and rendered useless just because it is not featured in this period's issue. Furthermore, you know from experience that such indexes are more useful to your visitors when they contain more than just a list of hyperlinked titles. They need a greater level of detail (date published, article summary, etc.) about how the archived articles are presented in the index. When faced with this task, developers traditionally go one of two ways: they either choose to create and maintain the index by hand, or they store the archived articles in a database and use a script to generate the index dynamically. Using XML, AxKit, and aggregate URIs offers a novel, third choice: you can set up alternate styles for the articles themselves to automatically extract the metadata you need, then combine those article-specific bits of data into a single resource that is the content source for the online index.

The advantage here is that filtered aggregated results are cached on disk after the first request—saving the overhead of reading, parsing, and transforming each individual document.

Example 7-4 shows the stylesheet that extracts the metadata from the articles. To keep things simple, assume that all the articles are marked up using the Simplified DocBook grammar.

Example 7-4. sdocbook_meta.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet

    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

    xmlns:dc="http://purl.org/dc/elements/1.1/"

    version="1.0"

>

<xsl:param name="request.uri.hostname"/>

<xsl:param name="request.uri"/>

<xsl:variable name="fullURI" select="concat('http://', $request.uri.hostname, $$


<xsl:template match="article">

<rdf:RDF>

  <rdf:Description rdf:about="{$fullURI}">

    <dc:format>text/xml</dc:format>

    <dc:title><xsl:value-of select="title|articleinfo/title"/></dc:title>
    <xsl:apply-templates select="articleinfo"/>
  </rdf:Description>
</rdf:RDF>
</xsl:template>

<xsl:template match="publishername">
  <dc:publisher><xsl:value-of select="."/></dc:publisher>
</xsl:template>

<xsl:template match="authorgroup">
<dc:creator>
  <rdf:Bag>
    <xsl:for-each select="author">
      <rdf:li>
        <xsl:value-of select="concat(firstname, ' ', surname)"/>
      </rdf:li>
    </xsl:for-each>
  </rdf:Bag>
</dc:creator>
</xsl:template>

<xsl:template match="author">
  <dc:creator>
    <xsl:value-of select="concat(firstname, ' ', surname)"/>
  </dc:creator>
</xsl:template>

<xsl:template match="articleinfo">
  <xsl:if test="copyright|legalinfo">
    <dc:rights>
      <xsl:if test="copyright">
        Copyright <xsl:apply-templates select="copyright"/>
        <xsl:value-of select="legalinfo"/>
      </xsl:if>
    </dc:rights>
  </xsl:if>
  <xsl:variable name="subjects"
       select="keywordset/keyword|subjectset/subject/subjectterm"/>
   <xsl:if test="count($subjects)">
```

```
    <dc:subject>
      <xsl:choose>
        <xsl:when test="count($subjects) > 1 ">
          <rdf:Bag>
            <xsl:for-each select="$subjects">
              <rdf:li><xsl:value-of select="."/></rdf:li>
            </xsl:for-each>
          </rdf:Bag>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="$subjects"/>
        </xsl:otherwise>
      </xsl:choose>
    </dc:subject>
  </xsl:if>
  <xsl:apply-templates select="./*[local-name( ) != 'copyright']"/>
</xsl:template>

<xsl:template match="copyright">
  <xsl:value-of select="year"/>
  <xsl:text> </xsl:text>
  <xsl:value-of select="holder"/>
</xsl:template>

<xsl:template match="abstract">
  <dc:description><xsl:value-of select="."/></dc:description>
</xsl:template>

<xsl:template match="pubdate">
  <dc:date><xsl:value-of select="."/></dc:date>
</xsl:template>

</xsl:stylesheet>
```

This simple stylesheet extracts the title, author's name, publisher, date, abstract summary, and several other useful bits of information from a Simplified DocBook article. It presents the information as an RDF/XML document using elements from the Dublin Core Metadata Initiative's suggested grammar. This a nice feature that will probably prove useful beyond your immediate goal, but you are not done yet. You need a way to pull the metadata for each article into a single resource. You can achieve this in several ways. Example 7-5 shows a simple RDF/XML document that lists the articles you want to include into the metadata aggregate.

Example 7-5. article_list.rdf

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 <rdf:Seq rdf:about="http://localhost/articles/"
      xml:base="http://localhost/articles/">
   <rdf:li rdf:resource="smithee.dkb" />
   <rdf:li rdf:resource="goldenage.dkb" />
   <rdf:li rdf:resource="nonlinear.dkb" />
 </rdf:Seq>
</rdf:RDF>
```

Note the use of the xml:base attribute. This not only saves you from having to type out the full URL for each article, but it provides the base URI when resolving the included documents. Example 7-6 shows the XSLT stylesheet that handles the task of combining the individual metadata results in one document.

Example 7-6. sdocbook_meta_aggregate.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:dc="http://purl.org/dc/elements/1.1/"
     version="1.0"
>

<xsl:template match="/">
<rdf:RDF>
  <xsl:apply-templates/>
</rdf:RDF>
</xsl:template>

<xsl:template match="rdf:li">
  <xsl:variable name="uri" select="concat(@rdf:resource, '?style=meta')"/>
  <xsl:copy-of select="document($uri, .)/rdf:RDF/*"/>
</xsl:template>

</xsl:stylesheet>
```

By using the two-argument form of the *document( )* function, the XSLT processor uses the base URI that you set using xml:base attribute in the source document to resolve the document's location. In this case, that means that the XSLT processor will make an HTTP request back to your web host for each article, specifying the meta style in the query string, which, in turn, will cause AxKit to apply the *sdocbook_meta.xsl* stylesheet to that document when returning the result. The sum of all of this is that you can now access all metadata from all the articles listed in the *article_files.rdf* file via a single URI. The result looks something like this:

```
<?xml version="1.0"?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

             xmlns:dc="http://purl.org/dc/elements/1.1/">

    <rdf:Description rdf:about="http://myhost.tld/articles/smithee.dkb">

        <dc:format>text/xml</dc:format>

        <dc:title>The History of Alan Smithee</dc:title>

        <dc:rights> Copyright 2001 John Q. Pundette, Esq.</dc:rights>

        <dc:subject>

            <rdf:Bag>

                <rdf:li>History of Cinema</rdf:li>

                <rdf:li>Film Directors</rdf:li>

            </rdf:Bag>

        </dc:subject>

        <dc:creator>John Pundette</dc:creator>

        <dc:date>2001</dc:date>

        <dc:description> Alan Smithee is credited with directing some of the worst films of all

            time-- and that's just the way he likes it. Find out more about this enigmatic and

            controversial figure. </dc:description>

    </rdf:Description>

    <rdf:Description rdf:about="http://myhost.tld/articles/goldenage.dkb">

        <dc:format>text/xml</dc:format>

        <dc:title/>

        <dc:rights> Copyright 2001 John Q. Pundette, Esq.</dc:rights>

        <dc:creator>John Pundette</dc:creator>

        <dc:date>2001</dc:date>

        <dc:description> Hollywood is back with a vengeance, and this time, it's

personal. </dc:description>

    </rdf:Description>

    <rdf:Description rdf:about="http://myhost.tld/articles/nonlinear.dkb">

        <dc:format>text/xml</dc:format>

        <dc:title>Some Assembly Required: Why We Need More Non-Linear Plots</dc:title>

        <dc:rights> Copyright 2004 Indy Filmsnob</dc:rights>

        <dc:creator>Indiana Filmsnob</dc:creator>

        <dc:date>2004</dc:date>

        <dc:description> Modern viewers have had enough of the Hansel and Gretel

School of plot

            development. Let's mix it up, people! </dc:description>
    </rdf:Description>

</rdf:RDF>
```

With this result, you have all the data you need to create a rich, meaningful index for all articles on your site; you only need to process this document with another stylesheet to generate browsable HTML. It does not stop there, though. For example, you may also reuse that same aggregate source to extract all information about articles written by a single author, or ones published during a specific date range. All you need to do is apply the appropriate stylesheets to the aggregate source to get what you need. The following configuration snippet ties together what you have done so far and shows some possibilities for filtering the aggregated metadata:

```
# These rules apply to the /articles/ directory

<Directory /articles/>


    # Add the query string StyleChooser

    AxAddPlugin Apache::AxKit::StyleChooser::QueryString


    # Transform sdocbook into HTML

    <AxStyleName html>

        AxAddProcessor text/xsl /styles/sdocbook_html.xsl

    </AxStyleName>


    # Extract the article's metadata as RDF

    <AxStyleName meta>

         AxAddProcessor text/xsl /styles/sdocbook_meta.xsl

    </AxStyleName>


    # HTML is the default style

    AxStyle html


    # Special rules apply to the metadata aggregate list

    <Files article_list.rdf>

        # The other styles never apply here

        AxResetProcessors

        AxAddProcessor text/xsl /styles/sdocbook_meta_aggregate.xsl


        # Show the aggregated data as an HTML index

        <AxStyleName html>

            AxAddProcessor text/xsl /styles/rdf_aggregate2html.xsl

        </AxStyleName>


        # Reuse the aggregated data for inclusions that filter the data

        # based on params passed that specify author, date range, etc.
        <AxStyleName filtered>
            AxAddProcessor text/xsl /styles/rdf_aggregate_filter.xsl
        </AxStyleName>

        # HTML is the default
        AxStyle html
    </Files>

</Directory>
```

This strategy still requires someone to manually update the *article_list.rdf* file as new articles are added. If you want to make the whole thing more or less maintenance free, you could use an XSP page to create the list of articles. (See Example 7-7.)

Example 7-7. article_list.xsp

```
<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1"
          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<xsp:structure>
  <xsp:import>File::Find::Rule::XPath</xsp:import>
</xsp:structure>

<xsp:logic>
    my $web_base = 'http://localhost/';
    my $finder = File::Find::Rule::XPath->new( );
    $finder->name('*.xml', '*.dkb');
    $finder->relative(1);
    $finder->xpath('/article');

    my @files = $finder->in( $r->document_root . '/articles/');
</xsp:logic>
<rdf:RDF>
 <rdf:Seq rdf:about="http://localhost/articles/"
      xml:base="http://localhost/articles/">
  <xsp:logic>
   foreach my $path ( @files ) {
      <rdf:li rdf:resource="{$path}"/>
   }

  </xsp:logic>

 </rdf:Seq>

</rdf:RDF>

</xsp:page>
```

This deceptively simple XSP page uses Grant McLean's File::Find::Rule::XPath module from CPAN to search the host's */articles/* directory (and its subdirectories) for all files with a *.dbk* or *.xml* extension and those containing a top-level article element. It generates a document with the same structure as the previous static file, except the list is created dynamically.

There is a trade-off here, though. Generating the list dynamically frees you from updating the list of articles, but it also means that all the aggregation processing happens *for every request*. With the static version, the result is cached on the first request, and the process is not repeated until the list of files changes.

To be sure, most aggregate data URI are not as complex as the one you created here. In fact, most of the time, a few simple XInclude elements do the trick. However, combining transformed resources into one resource that is also transformed to meet a specific need illustrates a powerful technique that would be difficult, if not impossible, to achieve outside a dynamic flexible XML environment such as AxKit.

7.2.2 Application ContentProviders

XSP offers a rich environment in which to generate dynamic XML content, but it is not to everyone's taste. For some, using markup application grammar and tag libraries, which map XML elements to Perl code that generates markup during processing, adds an undesirable layer of indirection to their day-to-day coding; others simply prefer to generate content within a familiar, pure Perl environment. As usual, AxKit's modularity offers an alternative. If XSP is not your style, you may consider creating or using an *application ContentProvider* instead.

In the most generic sense, an application ContentProvider is nothing more that an alternative Provider class that generates its content dynamically, based on one or more conditions (in contrast to the default Provider that reads the content from a file on the disk). More often, though, an application Provider is a bridge between an existing application environment and AxKit.

To illustrate this utility and the simplicity that using an application ContentProvider can offer, we will examine a simple two-state web application using the SAWA (Simple API for Web Applications) application environment. We will not delve into SAWA in detail here; just know that it implements the Model, View, Controller design pattern on top of a flexible dispatching mechanism that fires a series of registered event handler methods (across multiple component classes, if need be) in response to the current state of the application. The typical SAWA application consists of two parts: a base application class in which the processing logic for the current application state takes place, and an output class in which the concrete interface that represents state is generated. SAWA has the added benefit of an output class that is also an AxKit Provider. This gives SAWA both a way to pass content directly into AxKit as well as a means to control which transformative styles AxKit applies to the content returned.

To show how SAWA and AxKit can work together, we will create a simple application that accepts a bit of user input and then generate a random pairing for the data entered—essentially, a variation of the typical "Find Your (Gangster/Past Life/Hip-Hop DJ) Name" forms that pop up regularly on the Web. Example 7-8 shows the base application class.

Example 7-8. MovieName::Base.pm

```
package MovieName::Base;

use SAWA::Constants;

use SAWA::Event::Simple;

use strict;



our @ISA = qw( SAWA::Event::Simple );



BEGIN { srand(time( ) ^ ($$ + ($$ << 15))) }
```

```
sub registerEvents {
    return qw/ complete /;
}

sub event_default {
     my $self    = shift;
     my $ctxt   = shift;
     $ctxt->{style_name} = 'moviename_prompt';
     return OK;
 }

sub event_complete {
    my $self    = shift;
    my $ctxt    = shift;

    my @param_names = $self->query->param;

    my @first_names = map { $self->query->param("$_") } grep { $_ =~ /^first\./ }
@param_names;
    my @last_names  = map { $self->query->param("$_") } grep { $_ =~ /^last\./ }
@param_names;

    if ( ((scalar @first_names) + (scalar @last_names) != 6)
       || ( grep { length =  = 0 } @first_names, @last_names ) ) {

        $ctxt->{message} = 'All fields must be filled in. Please try again.';
        $ctxt->{style_name} = 'moviename_prompt';
        return OK;
    }

    $ctxt->{first_name} = $first_names[ int( rand( @first_names )) ];
    $ctxt->{last_name} = $last_names[ int( rand( @last_names )) ];
    $ctxt->{message} = 'Congratulations! Your Movie Star name has been magically determined!';
    $ctxt->{style_name} = 'moviename_complete';
    return OK;
}

1;
```

Two event-handler methods are presented here. The first is the *event_default( )* handler, which is called if no registered event is called. It simply sets the value for the style_name key in the global context hash reference (that is passed to every handler method) to the value stylea_prompt. (You'll see the effect this has when we examine the output class.) Next is the .*event_complete( )* that is called when the client submitted data. In this case, after a little error checking, the handler also sets the style_name key and adds the randomly chosen first and last names to the context hash. Example 7-9 shows the output class in which things get a bit more interesting.

Example 7-9. MovieName::Output.pm

```
package MovieName::Output;

use XML::LibXML;

use SAWA::Constants;

use SAWA::Output::XML::AxKit;

our @ISA = qw/ SAWA::Output::XML::AxKit /;


sub get_style_name {

    my $self = shift;

    my $ctxt = shift;

    $self->style_name( $ctxt->{style_name}  );

    return OK;

}


sub get_document {

    my $self = shift;

    my $ctxt = shift;

    my $dom  = XML::LibXML::Document->new( );

    my $root = $dom->createElement( 'application' );

    $dom->setDocumentElement( $root );


    my $msg_element = $dom->createElement( 'message' );

    $msg_element->appendChild( $dom->createTextNode( $ctxt->{message} ) );


    my $fname_element = $dom->createElement( 'first_name' );

    $fname_element->appendChild( $dom->createTextNode( $ctxt->{first_name} ) );


    my $lname_element = $dom->createElement( 'last_name' );

    $lname_element->appendChild( $dom->createTextNode( $ctxt->{last_name} ) );


    $root->appendChild( $fname_element );

    $root->appendChild( $lname_element );
    $root->appendChild( $msg_element );


    $self->document( $dom );

    return OK;

}


1;
```

First, this class is a subclass of SAWA::Output::XML::AxKit, which together with Apache::AxKit::Provider::SAWA, provides the bridge between SAWA and AxKit. In practical terms, this means that the data you pass to class member accessors from these output methods are forwarded directly to AxKit. In the case of the *get_style_name( )* event handler, you set the class member style_name to the value selected by the event hander in the Base.pm application class. The name selected (moviename_prompt or moviename_complete) corresponds to the AxKit-named style blocks whose processing directives are used to transform the content returned.

The *get_document( )* event handler simply generates an XML::LibXML::Document instance by which you create the XML (via the DOM interface) that is passed off AxKit to transform. (This SAWA class also allows you to return the content as textual XML markup, but then AxKit has to parse it before transformation. Returning a document object saves a little bit of parsing overhead).

Now you have seen how the style is selected and the content created. Let's look at the configuration that enables this to work seamlessly with AxKit:

```
# Load the AxKit<->SAWA "bridge"

PerlModule Apache::AxKit::Provider::SAWA


<Location /moviename>

    # Set AxKit as the handler for this virtual URI

    SetHandler axkit

    AxResetProcessors


    # Add the request params plug-in so XSLT can access request data

    AxAddPlugin Apache::AxKit::Plugin::AddXSLParams::Request

    PerlSetVar AxAddXSLParamGroups "Request-Common"


    # Set the "bridge" as the AxKit ContentProvider for this resource

    AxContentProvider Apache::AxKit::Provider::SAWA


    # Add out two SAWA modules to its dispatcher pipeline

    SawaAddPipe MovieName::Base

    SawaAddPipe MovieName::Output


    # Set up the AxKit  named styles that your SAWA application will select to

    # create the View for the current application state.

    <AxStyleName moviename_prompt>

        AxAddProcessor text/xsl /styles/moviename/prompt.xsl
        AxAddProcessor text/xsl /styles/moviename/common.xsl

    </AxStyleName>


    <AxStyleName moviename_complete>

        AxAddProcessor text/xsl /styles/moviename/complete.xsl

        AxAddProcessor text/xsl /styles/moviename/common.xsl

    </AxStyleName>


</Location>
```

This configuration block loads the Apache::AxKit::Provider::SAWA class as a PerlModule (required, since it implements its own set of custom Apache configuration directives). It then creates a virtual URI, /moviename, which uses that module as the AxKit ContentProvider. Next, the SAWA application and output classes are added to SAWA's pipeline, and the AxKit-named style blocks that the application will select to transform the content are added.

Your little application has two states: the default state (moviename_prompt), which prompts the users to enter the data that will be used to randomly generate their movie star name, and the complete state (moviename_complete), in which the generated result is presented. SAWA determines the application state, then passes the name of the style that reflects that state to AxKit, which performs the actual content transformation.

Let's take a quick look at the stylesheets themselves before moving on. First, Example 7-10 shows the style associated with the default, prompt state.

Example 7-10. prompt.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:import href="params.xsl"/>


<xsl:template match="/">

  <xsl:apply-templates/>

</xsl:template>


<xsl:template match="application">

  <application>

  <xsl:apply-templates/>

  <body>

    <form name="prompt" action="{$request.uri}" method="post">

      <div class="appmain">

        <div class="row">

          <span class="label">Name of classy hotel:</span>

          <span class="inputw">

              <input name="last.hotel" type="text" value="{$last.hotel}"/>

          </span>
```

```
</div>
<div class="row">
  <span class="label">
   Name a street you lived on when you were a teenager:</span>
  <span class="inputw">
      <input name="last.street" type="text" value="{$last.street}"/>
  </span>
</div>
<div class="row">
  <span class="label">Your mother's maiden name:</span>
  <span class="inputw">
      <input name="last.maiden" type="text" value="{$last.maiden}"/>
  </span>
</div>
<div class="row">
  <span class="label">Your middle name:</span>
  <span class="inputw">
      <input name="first.middle_name" type="text" value="{$first.middle_name}"/>
  </span>
</div>
<div class="row">
  <span class="label">The name of your favorite pet:</span>
  <span class="inputw">
      <input name="first.pet" type="text" value="{$first.pet}"/>
  </span>
</div>
<div class="row">
  <span class="label">Name of your favorite model of car:</span>
  <span class="inputw">
      <input name="first.car" type="text" value="{$first.car}"/>
  </span>
</div>
<div style="padding-top: 10px; text-align: center; clear: both;">
  <input name="complete" type="hidden" value="1"/>
  <input type="submit" value="Generate Name"/>
  <input type="reset" value="Start Over"/>
</div>
```

```
      </div>
    </form>
  </body>
  </application>
</xsl:template>

</xsl:stylesheet>
```

As you may expect, this stylesheet does little more than create the data entry interface that visitors will use to type in the information that will be shuffled to select their movie star name. Example 7-11 shows the stylesheet associated with the complete application state.

Example 7-11. complete.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:import href="params.xsl"/>

<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="application">
  <application>
  <xsl:apply-templates/>
  <body>
   <div>
     <p>
       Your new <i>nom d'cinema</i> is:
       <b>
         <xsl:value-of select="/application/first_name"/>
         <xsl:text> </xsl:text>
         <xsl:value-of select="/application/last_name"/>
       </b>
     </p>
    </div>
    <form name="prompt" action="{$request.uri}" method="post">
      <input name="last.hotel" type="hidden" value="{$last.hotel}"/>
      <input name="last.street" type="hidden" value="{$last.street}"/>
      <input name="last.maiden" type="hidden" value="{$last.maiden}"/>
      <input name="first.middle_name" type="hidden" value="{$first.middle_name}"/>
      <input name="first.pet" type="hidden" value="{$first.pet}"/>

      <input name="first.car" type="hidden" value="{$first.car}"/>

      <input name="complete" type="hidden" value="1"/>

      <div>

        <input type="submit" value="Regenerate"/>

        <input type="button" value="Start Over" onClick="location='{$request.uri}'"/>

      </div>

    </form>

  </body>

  </application>

</xsl:template>


</xsl:stylesheet>
```

This stylesheet presents the randomly selected movie star name (passed through in the XML returned from the SAWA output class) along with a hidden form that allows the user to regenerate a new name using the same set of input data. Also, each of the state-specific stylesheets imports the *params.xsl* stylesheet. (See Example 7-12.) This is really just a tiny stylesheet fragment that allows the other stylesheets to access the CGI and other request parameters needed for the two HTML forms.

Example 7-12. params.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">


<!-- Global Params -->

<xsl:param name="request.uri"/>

<xsl:param name="last.hotel"/>

<xsl:param name="last.street"/>

<xsl:param name="last.maiden"/>

<xsl:param name="first.middle_name"/>

<xsl:param name="first.pet"/>

<xsl:param name="first.car"/>


</xsl:stylesheet>
```

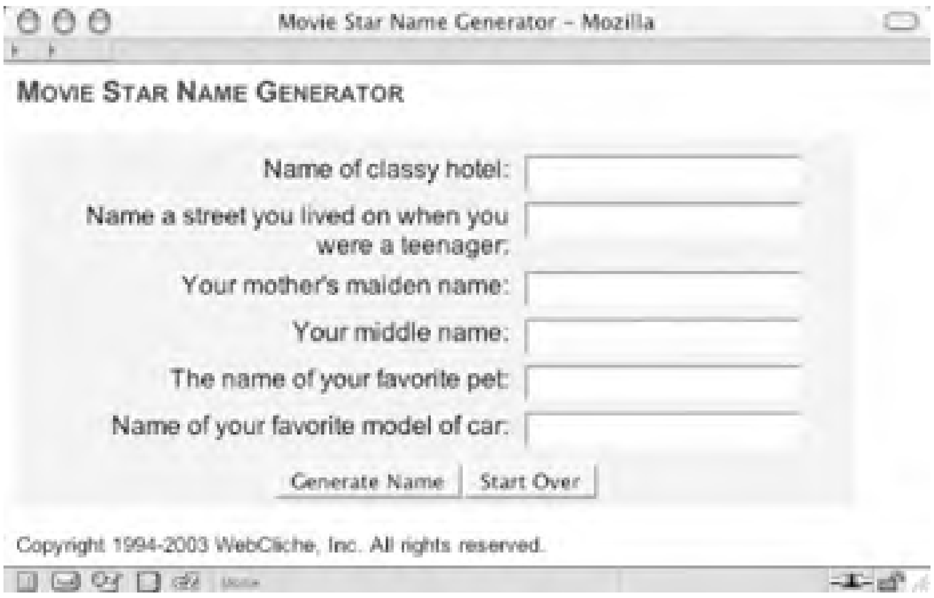
From the earlier configuration, note that the two state-specific stylesheets are first in their respective AxKit style processing chains and that both chains pass their content through to a final *common_html.xsl* stylesheet. (See Example 7-13.) This small stylesheet simply passes through the result of the previous XML to HTML transformations, while adding the global header and footer common to both application states.

Example 7-13. common_html.xsl

```
<![CDATA[
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="html"/>
<xsl:template match="/">
<html>
  <head>
    <title>Movie Star Name Generator</title>
  </head>
  <xsl:apply-templates select="/application/body"/>
</html>
</xsl:template>

<xsl:template match="body">
<body>
  <div class="pagehead">
    Movie Star Name Generator
  </div>
  <div class="message">
    <p>
      <xsl:apply-templates select="//message"/>
    </p>
  </div>
  <xsl:apply-templates select="./*"/>
  <div class="legal">
    Copyright 1994-2003 WebCliche, Inc. All rights reserved.
  </div>
</body>
</xsl:template>

  <!-- copy the HTML from the previous transformation verbatim -->
  <xsl:template match="node( )|@*">
    <xsl:copy>
      <xsl:apply-templates select="@*|node( )" />
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

That's all there is to it. A couple of tiny Perl classes, a few small XSLT stylesheets, and the application is complete. Let's look at the result. Chapter 7 shows the prompt. Figure 7-2 shows the completed result.

Figure 7-1. Movie star name generator: state two (prompt)

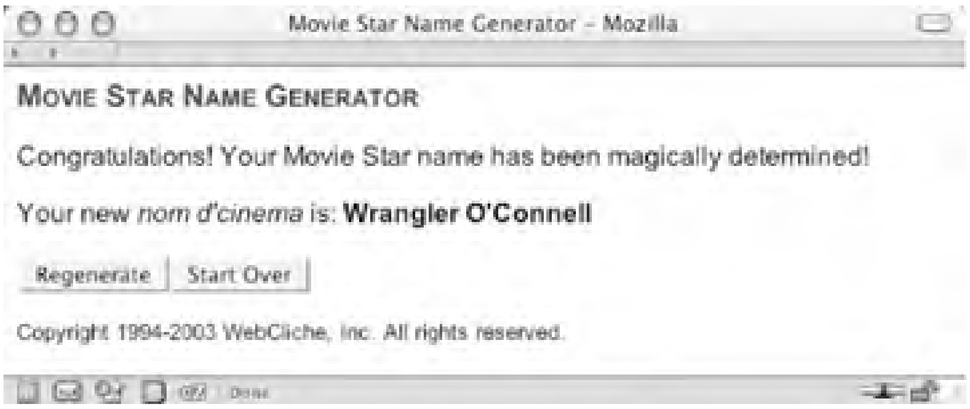


MOVIE STAR NAME GENERATOR

Name of classy hotel:	
Name a street you lived on when you were a teenager:	
Your mother's maiden name:	
Your middle name:	
The name of your favorite pet:	
Name of your favorite model of car:	

Copyright 1994-2003 WebCliche, Inc. All rights reserved.

Figure 7-2. Movie star name generator: state two (complete)



MOVIE STAR NAME GENERATOR

Congratulations! Your Movie Star name has been magically determined!

Your new *nom d'cinema* is: **Wrangler O'Connell**

Copyright 1994-2003 WebCliche, Inc. All rights reserved.

You may wonder why you might combine both SAWA and AxKit in the same application, given that each offers features that the other possesses. It's really a personal choice. Having used both extensively, for me, the combination is a natural fit. SAWA's simple modularity and Perl's pure environment are well suited for generating XML content. This, plus AxKit's flexible styling options and built-in caching facilities, add up to a winning combination. In any case, whether you favor XSP, application ContentProviders, aggregate data URIs, or a combination of all three, Axkit lets developers choose the tools and techniques that suit them best.

Chapter 7. Serving Dynamic XML Content

So far, we have examined how AxKit can be used to transform static XML documents. Transforming such documents via XSLT stylesheets or another means that can alter, create, or include content conditionally provides a certain level of dynamism, but most modern web sites include features or resources that need to change for each unique request. An altered version of a static source is not enough, the *source itself* must be generated programatically.

By now, you are surely aware of the value of separating content from presentation, and how that value increases when the documents being served are marked up using semantically rich grammar that intimately captures (or communicates) the meaning of the information they contain. The same principle holds true when creating XML-based applications—the key difference is that markup generated in an application context should attempt to most accurately reflect the structure and roles of the resources associated with (or required by) the current state of the application, rather than those of a static narrative document.

By generating only the data relevant to the current state of the application, developers working on the backend libraries have simpler and better defined targets to hit. Since the markup being generated does not include any presentational elements, the grammar for the document being produced is usually greatly simplified. Among other things, this means that the generated content can be evaluated in detail using nothing more than a validating XML parser and a Document Type Definition (for instance, one of the XML schema languages) to verify the result. Hence, acceptance testing can become fully deterministic—the coder knows her job is done when the generated grammar passes a validity test, not when it seems to render correctly in one or another client application.

Similarly, having a simplified, well-defined application grammar offers stylesheet authors the ability to develop and test their stylesheets using sample documents that accurately reflect the data that will be generated when the application goes into production. This means that application development can push ahead asynchronously—the stylesheets that will render the application content can be completed at the same time (or even before) the code that generates the content is written.

Also having the ability to reuse the same content to meet the needs and expectations of a variety of clients is a key benefit of an XML application environment. Being able to deliver the same essay as either HTML or PDF is a cool feature; being able to offer the same online *application* to a variety of clients can directly affect the profitability of a web-based enterprise.

In short, the benefits of separating content from interface are especially noticeable when the markup in question is being generated dynamically in the context of an online application. In this chapter, I introduce a few of the tools and techniques that AxKit offers for creating content dynamically and explain how applying transformative styles to that generated content can be used to create flexible, sophisticated online applications.

8.1 AxKit's Architecture

AxKit implements a modular design in which several smaller components are combined to create the larger application. Each component type has an associated configuration option that allows the default components to be changed at runtime on a resource-by-resource basis. In practice, this means that AxKit is extremely malleable and extensible—if one or another component does not do exactly what you want for a given situation, you are free to swap in a new component that does, while still taking advantage of the rest of what AxKit has to offer. In fact, AxKit's continued popularity is arguably due, in large part, to the fact that it provides a set of stock components that covers a great many common cases while still giving developers the ability to easily address special requirements without reinventing the wheel. (See Figure 8-1.)

Plug-ins

Plug-in modules stand between the user's request and the bulk of AxKit's processing cycle. They are used primarily to process the incoming request and to set any internal properties that will help AxKit respond appropriately.

Providers

Provider modules handle the task of getting the source for the documents and stylesheets that are used to serve a given request. Each resource has a ContentProvider that is responsible for getting the XML content and a StyleProvider that fetches the source of any stylesheets to be applied during processing.

Languages

Language modules provide different ways to transform the XML content being served. Often, an AxKit Language module is a thin wrapper around an existing XML processing library. For example, XPathScript is made available to AxKit via Apache::AxKit::Language::XPathScript, while Apache::AxKit::Language::LibXSLT offers XSLT transformations by creating an interface to the Gnome Project's libxslt library.

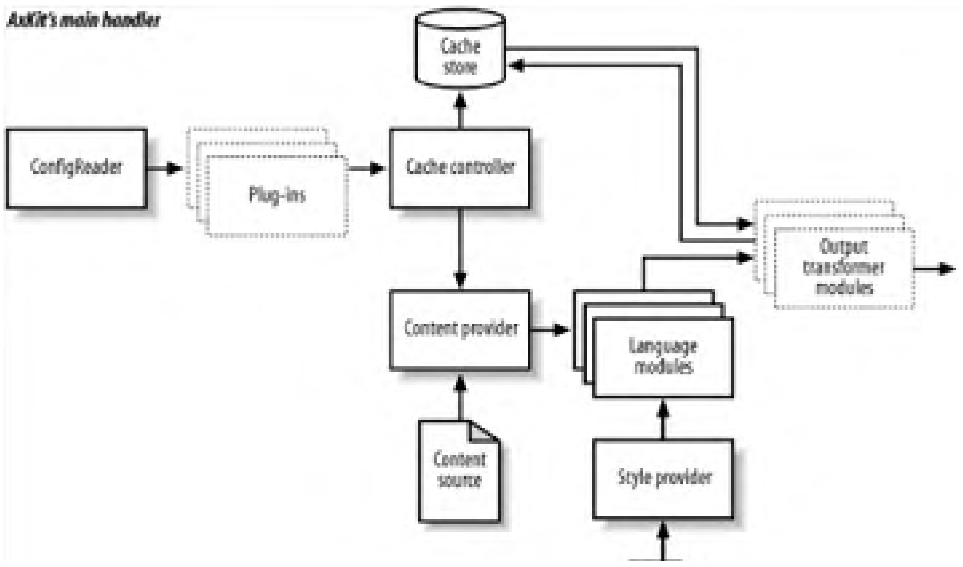
ConfigReaders

ConfigReader modules control how and from where AxKit gets its runtime configuration information.

Caches

Caching modules determine how, where, and under what conditions various resources are cached and delivered to the requesting client.

Figure 8-1. AxKit's order of execution



From Figure 8-1, you can see that first, AxKit initializes the ConfigReader class to get the runtime options needed to handle the client's request. Next, any plug-in modules configured for the current resource are loaded and run (in the order returned from the ConfigReader). Following that, the Provider and Cache modules are initialized. If all caching conditions are met, the pretransformed result is sent to the client, and processing ends there. If the resource is not cached (or the cache has expired, or caching is turned off) the Provider class responsible for fetching the source XML content is loaded, as well as the Language modules associated with the transformations to be applied to that content. Each Language module is then called, and transformed content is sent to the client. There are additional levels of detail that we will touch on while examining the component classes individually, but from the highest level, this is how AxKit works.

Before we dive into the details of extending and replacing AxKit's default components, you should be sure to have a basic understanding of Perl and its capabilities as an object-oriented language. Specifically, you should feel comfortable creating new Perl modules, especially those that are inherited subclasses of other existing modules, and the techniques for installing those modules into places where the Perl interpreter can find them. Of course, since AxKit is implemented on top of *mod_perl*, the more you know about that uniquely powerful and flexible environment, the more creative and productive you are likely to be from the very start.

The Apache configuration directives shown in each of the following sections to demonstrate how to swap in a new component type presume that the default Config- Reader component is being used. Obviously, if you implement a different way for AxKit to get its configuration data, you need to use your own custom interface to set these options.

8.2 Custom Plug-ins

Referring to the order of execution introduced earlier, you can see that plug-ins are the first AxKit components to run after the ConfigReader is created. This makes a plug-in the most logical place to put custom code that alters AxKit's processing behavior based on data from the user's request. In fact, the StyleChooser and MediaChooser modules that ship with AxKit are really only plug-ins with conventional names that hope to provide a clue about their intended use.

If the default ConfigReader class is used, plug-ins are added via the AxAddPlugin directive. Plug-ins are run in the order that they appear in the configuration file:

```
AxAddPlugin Apache::AxKit::Plugin::Passthru


# Allow Cookie-based style selection the 'forums' directory

<Directory forums>

  AxAddPlugin Apache::AxKit::StyleChooser::Cookie

</Directory>


# Reset the plug-ins list and add a new set for a particular application

<LocationMatch /about/contact>

  AxResetPlugins

  AxAddPlugin MyApp::Plugin

</LocationMatch>
```

8.2.1 Plug-in API

All AxKit plug-in modules must implement one function, *handler( )*, which is passed an Apache::Request instance for the current client request as its sole argument. Access to this request object gives plug-ins the same power to examine and alter the incoming request as a standard *mod_perl* content handler. Most often, though, they are used to set one or both of the preferred_style or preferred_media properties in the request object's pnotes table that AxKit uses later to determine which styles apply while delivering the content.

Example 8-1 shows one possible way to provide a friendlier environment for a site's international or multilingual visitors by selecting a different preferred style based on the relationship between the requesting client's Accept-Language header and the host's internal language priority list.

Example 8-1. StyleChooser::AcceptLanguage

```
package StyleChooser::AcceptLanguage;

use strict;

use Apache::Constants qw(OK);

use Cookbook::LanguagePriority;


sub handler {

    my $r = shift;

    my $lp = Cookbook::LanguagePriority->get( $r );
    my $lang_header = $r->header_in('accept-language');
    my $chosen_language = undef;

    if ( defined( $lang_header ) ) {
        $chosen_language = best_language_match( $lang_header, @{$lp->priority} );
    }
    $chosen_language ||= $r->dir_config('DefaultStyleLanguage');

    # Modifies the preferred style name
    # to <stylename>_<language>
    my $current_style = $r->notes('preferred_style') || 'default';
    $r->notes('preferred_style', $current_style . '_' . $chosen_language );

    return OK;
}

sub best_language_match {
     my $accept_lang = shift;
     my @priority_list = @_;

     # The Schwartzian Transform in action
     my @user_languages =
     map $_->[0],
     sort { $b->[1] <=> $a->[1] }
     map {
        my ( $lang, $range ) = split/\s*;\s*/, $_;
        $range = defined( $range ) && $range =~ /^q=["']?(.+?)["']?$/ ? $1 : 1;
        $lang =~ s/^(.+?)-(.+?)$/$1/;
        [ $lang, $range ];
     } split /\s*,\s*/, $accept_lang;

    foreach my $lang ( @user_languages ) {
        for ( @priority_list ) {
            return $_ if $_ eq $lang;
        }
    }
}

1;
```

With this module installed, you only need to add the appropriate AxAddPlugin directive and named styles to your configuration. Again, the named styles take the form of <stylename>_<language> that your plug-in produces.

```
# Add the AcceptLanguage plug-in
AxAddPlugin StyleChooser::AcceptLanguage

# Set the default language:(referenced by the plug-in)
PerlSetVar DefaultStyleLanguage en

# Now the styles
<AxStyleName style1_en>
    AxAddProcessor text/xsl /styles/style_one.xsl
    AxAddProcessor text/xsl /styles/english.xsl
</AxStyleName>

<AxStyleName style1_de>
    AxAddProcessor text/xsl /styles/style_one.xsl
    AxAddProcessor text/xsl /styles/german.xsl
</AxStyleName>

<AxStyleName style2_en>
    AxAddProcessor text/xsl /styles/style_two.xsl
    AxAddProcessor text/xsl /styles/english.xsl
</AxStyleName>

<AxStyleName style2_de>
    AxAddProcessor text/xsl /styles/style_two.xsl
    AxAddProcessor text/xsl /styles/german.xsl
</AxStyleName>
. . .
```

8.3 Custom Providers

By default, AxKit presumes that the documents that it is serving and processing are stored as plain files on the filesystem. While this is typically adequate for most cases, in some situations you may need to get data from another source. In AxKit, the mechanism for slurping in data for further processing is called a Provider.

Providers come in two flavors: ContentProviders and StyleProviders. As their names suggest, ContentProviders are responsible for fetching the source of the content being delivered, while StyleProviders handle getting the source of the stylesheets to be used to transform that content. The default class for both types, Apache::AxKit::Provider::File, reads data from XML sources on the filesystem. Alternate classes can be configured for both the ContentProvider and the StyleProvider for a given resource using the corresponding AxKit configuration directive:

```
# Set each type of Provider explicitly

AxContentProvider My::Provider

AxStyleProvider My::Other::Provider



# Or, configure both to use the same alternate class

AxProvider My::Generic::Provider
```

Custom Providers can be used to fetch content from non-XML data, to get XML data from sources other than flat files, or a combination of the two. In some cases, you are looking to take advantage of Perl's capable XML tools and other data processing facilities to *generate* an XML instance based on another source of data. In others, you simply want to read in XML for a source other than a plain file on the disk. For example, some Providers may use a SAX generator class to dynamically generate an XML document from a directory listing or Excel spreadsheet, while others may be used to extract existing XML content from a zip archive or relational database.

But wait, as you saw in Chapter 7, AxKit offers several fine options for generating dynamic content, so why would you use a Provider instead of a taglib? There is no hard and fast rule, but, in general, defining the real *source* of the content decides the matter. For example, a shopping cart application that includes a list of products from a database is probably best implemented through a taglib, while a content management system that returns a complete document from the same database may best be integrated into AxKit as a custom ContentProvider. That is, in the shopping cart page, the product list is only one component that is *included* into the content, while the data returned from the CMS *defines* the content. This distinction may seem a bit arbitrary, and from a technical point of view, it is, given that either task could be achieved by either means. Spending a few minutes considering the best approach for the task at hand can save hours of development time in the long run.

8.3.1 Provider API

Generally, a Provider is expected to offer access to the sources of the content or stylesheets that are associated with the current request, as well as certain key pieces of metadata about those sources. The data for the given resource must be returned from one of the *get_fh* (get filehandle), get_strref (get string reference), or get_dom (get Document Object instance) methods. These methods are simply variations that allow the source to be passed to AxKit in different data formats. Only one must be implemented for the Provider to work. All custom Providers should be inherited subclasses of AxKit's base Provider class Apache::AxKit::Provider or one of its subclasses:

```
package My::Provider;



use strict;

use Apache::AxKit::Provider

use vars qw( @ISA );

@ISA = qw( Apache::AxKit::Provider );
```

```
 # Override some class methods and

 # add few of your own.


1;
```

8.3.1.1 init( )

Called before all other methods, the *init* method gives the Provider a chance to perform any initialization logic needed to prepare for further processing. It is most commonly used for things such as instantiating any objects that the Provider needs to handle the request, initializing instance variables, and so on. In classes that inherit from AxKit's base Provider class (and most should), it is passed the same arguments that were passed to the constructor for the current instance:

```
sub init {

    my $self = shift;

    my %args = @_;


    $self->{content_application} = My::App->new( );

    # and so on . . .

}
```

8.3.1.2 process( )

The *process* method is used to communicate whether or not the Provider provides content for the given resource. It is passed no arguments (apart from the instance reference passed as the first argument to all Perl methods) and is expected to return 1 (or any nonzero value) to indicate that all conditions are met for the Provider to handle the request and 0 or undef, otherwise. For cases in which the Provider cannot continue, it is strongly recommended that an appropriate exception be thrown, providing an explanation as to what may have gone wrong:

```
sub process {

    my $self = shift;


    my $uri = $self->apache_request->uri( );


    # Get the data based on some URI-to-data mapping method

    # implemented elsewhere in your custom Provider

    my $data = $self->map_uri_to_data( $uri );


    if ( defined( $data ) ) {

        $self->{data} = $data;

        return 1;

    }

    else {

        throw Apache::AxKit::Exception::Error( -text => "No data associated
with URI '$uri'." );

    }

}
```

8.3.1.3 mtime( )

Used with AxKit's caching mechanism, the *mtime* method is expected to return the last modification time in seconds, since the epoch, for the current resource. If the document being provided will always be dynamic (based on user input, etc.), returning the result of Perl's built-in *time( )* function ensures that the data is never cached unexpectedly:

```
sub mtime {

    return time( ); # content is always fresh.

}
```

Implementing *mtime* correctly for cases in which the data being provided is not a plain file on the disk but is an aggregate of data from more than one source can be tricky. For example, if the content is being built as the result of an SQL query that joins several tables that may have been updated at different times, how does one determine the true last modification time for that resource? The answer is always very application-specific, and I will avoid making dubious generalizations here. It is enough to say that being able to take advantage of AxKit's caching facilities wherever possible and appropriate is a *huge* performance gain. The time spent implementing *mtime* is usually worth the investment.

8.3.1.4 get_styles( )

Called only on ContentProvider classes, the *get_styles* method is responsible for returning the final list of processors to be applied to the given resource. It is expected to return a reference to a list of style definitions that AxKit uses to transform the content. Styles are applied in the order that they appear—the first style is applied to the source content, the second to the result of the first, and so on. The style definitions take the form of an anonymous HASH reference containing two required keys: href whose value contains the DocumentRoot-relative path to the stylesheet to be applied, and type, whose value declares the MIME type associated with the Language processor to be used to apply the stylesheet:

```
my @styles = ( { type => "text/xsl",

                 href => "/styles/style1.xsl"

               },

                type => "text/xsl",

                href => "/styles/style2.xsl"

               }

             );
```

In the default ContentProviders, *get_styles( )* is used to map the current preferred style and media to any xml-stylesheet processing instructions contained in the source XML. If no matching styles are found, the ConfigReader's *GetMatchingProcessors* method is called, the document's root element name and Document Type Definition are evaluated against all AxAdd*Processor configuration directives in the current scope, and any matching styles are used instead. In all, *get_styles* is a crucial method whose default implementation provides much of AxKit's expected behavior. It should be overridden only with caution and a clear purpose.

That said, some Providers, most notably those implementing a bridge between AxKit and a content creation application that needs to define one or more stylesheet transformations to create the "view" of a given set of data for a particular application state, may need explicit control over the list of styles to apply to the content. In these cases, *get_styles* offers the most direct, least ambiguous way to define the styles to be applied. Overriding the default implementation of this method does not mean abandoning the use of the preferred style and media properties that an upstream plug-in may have set—these values are passed in as arguments to *get_styles*. The following shows how an application-based Provider may conditionally override the current list of styles, while still falling back to any default styles defined via configuration directive or xml-stylesheet processing instruction:

```
package My::Provider;

use vars qw( @ISA );

@ISA = qw( Apache::AxKit::Provider );


 . . .


sub get_styles {

    my $self = shift;

    my ( $preferred_media_name, $preferred_style_name ) = @_;


    my $app = $self->{some_content_application};

    my @style_list = $app->get_axkit_styles( $preferred_media_name,

$preferred_style_name );


    # If your application returned styles, use those; otherwise, fall back to the

    # default implementation in your parent Provider class.

    if ( scalar( @style_list > 0 ) ) {

        return \@style_list; # you return a reference, not the list itself.

    }

    else {

        return $self->SUPER::get_styles( $preferred_media_name, $preferred_style_name );

    }

}
```

Or, here's how an application-driven ContentProvider may alter the preferred media and style properties while letting the default Provider handle the low-level details:

```
sub get_styles {

    my $self = shift;

    my ( $preferred_media_name, $preferred_style_name ) = @_;


    my $app = $self->{some_content_application};


    my $new_preferred_style = $app->get_axkit_stylename( ) || $preferred_style_name;

    my $new_preferred_media = $app->get_axkit_medianame( ) || $preferred_media_name;


    return $self->SUPER::get_styles( $new_preferred_media, $new_preferred_style );

}
```

8.3.1.5 get_strref( )

One of three methods available for returning content, *get_strref* (get string reference) offers the ability to return the XML content for the current resource as a reference to a scalar containing the entire document as a string. For example, the following shows how a custom ContentProvider built on XML::Generator::DBI (which generates SAX events from the result of a database query) may implement *get_strref* to return a generated XML instance:

```
sub get_strref {
    my $self = shift;
    my $content = undef;

    my $writer = XML::SAX::Writer->new( Output => \$output );

    my $generator = XML::Generator::DBI->new(
                           Handler => $writer,
                           dbh => $self->{db_handle}
                           );

    eval {
        $generator->execute( $self->{sql_statement} );
    };

    if ( my $error = $@ ) {
        throw Apache::AxKit::Exception::Error( -text => "Error generating XML: $error" );
    }

    if ( length( $content ) ) {
        # you return a reference, not the scalar itself
        return \$content;
    }
    else {
        throw Apache::AxKit::Exception::Error( -text => "No data
was returned from SQL $self->{sql_statement}." );
    }

}
```

8.3.1.6 get_fh( )

Similar to *get_strref*, the *get_fh* (get filehandle) method offers a way to return data as an open filehandle. In some circumstances too complex to detail here, a filehandle requires fewer system resources than a scalar variable that contains the same document as a plain string; *get_fh* offers a way to take advantage of that optimization.

```
 # As above, but return a filehandle instead
sub get_fh {
    my $self = shift;

    # Use the Apache-friendly way to create a new filehandle
    my $handle = $self->apache_request->gensym( );

    my $writer = XML::SAX::Writer->new( Output => \$handle );

    my $generator = XML::Generator::DBI->new(
                             Handler => $writer,
                             dbh => $self->{db_handle}
                             );

    eval {
        $generator->execute( $self->{sql_statement} );
    };

    if ( my $error = $@ ) {
        throw Apache::AxKit::Exception::Error( -text => "Error generating XML: $error" );
    }

    return $handle;
}
```

8.3.1.7 get_dom( )

The *get_dom* method offers a way to return the XML data for the current resource as an XML::LibXML::Document instance. It is used most often as a means to pass the content from application frameworks such as SAWA and CGI::XMLApplication without incurring the overhead of serializing that DOM object via its *toString* method and reparsing it once it is passed into AxKit.

```
sub get_strref {
    my $self = shift;
    my $content = $self->{XML_APP}->getDom( );

    unless ( $content ) {
        throw Apache::AxKit::Exception::Error( -text => "Error generating XML,
no document object returned" );
    }
    return $content;

}
```

8.3.1.8 key( )

Called throughout AxKit, the Provider's *key* method should return a string that can be used as a persistent, unique identifier for the current resource. It is used extensively by AxKit's default caching mechanism (along with *mtime*) to both look up content that may be cached on the disk or to create the ID for a new cache entry if caching is turned on and none previously existed.

In the default file Provider, *key* simply returns the filename associated with the current request, which is sufficient in most cases. File-based alternate Providers are encouraged to do the same, or to inherit from Apache::AxKit::Provider::File and avoid implementing the *key* method altogether. In cases in which there is no one-to-one mapping between the current request URI and a file on the filesystem, a smarter *key* method is almost always required.

Suppose you use a content management system for part of your site that stores the source XML documents in a relational database. You are now faced with creating the public interface to that data. Let's go a step further and say that your CMS offers an internal hierarchical mapping that allows content objects to be selected using a path interface. Setting up the interface is easy. You only need to create a virtual URL with a <Location> directive and set your custom Provider as the ContentProvider for that URL. Then you can simply use any additional path information from the incoming request as the path passed to your application to retrieve the content. You must explicitly set the cache directory for this resource, since, by default, AxKit attempts to write the cache to the same directory as the requested content—in this case, a directory that does not actually exist.

```
# virtual URI for public side of your CMS

<Location /cmsapp/content>

  AxContentProvider My::CMS::Provider

  # always set an explicit cache for virtual URIs

  AxCacheDir /.mycachedir

</Location>
```

Given that all requests for content within this resource always have the same value from the request object's *filename*, you cannot just use the same strategy as the default Providers. You must use the full URI (including the additional path information) in the string returned to create a unique cache key for each document in the document store:

```
sub key {

    my $self = shift;

    my $r = $self->apache_request( );


    return $r->uri( );

}
```

You can achieve the same effect using a unique property from the content object itself:

```
sub key {

    my $self = shift;

    return $self->{content_object}->id( );

}
```

8.3.1.9 exists( )

This method is expected to return 1 (or any nonzero value) if the resource exists and is readable and 0 or undef, otherwise. Typically, a class member added to the current instance during *init* or *process* can be examined and the appropriate value returned.

```
sub process {
    my $self = shift;


    my $uri = $self->apache_request->uri( );
    my $data = $self->map_uri_to_data( $uri );


    if ( defined( $data ) ) {
        $self->{data} = $data;
                    $self->{exists} = 1;
        return 1;
    }
    else {
        throw Apache::AxKit::Exception::Error( -text => "No data associated with
URI '$uri'." );
    }
}


sub exists {
    my $self = shift;
    if ( defined( $self->{exists} ) ) {
        return 1;
    }
    else {
        return 0;
    }
}
```

It is worth mentioning again: as subclasses of one of AxKit's default Providers, most custom Providers only ever need to implement a few of these methods. Often, implementing both the *process* method to fetch and preprocess the data from the given source and one of the *get_** methods to return that data to AxKit are all that is required for a working Provider. To bring this all together, a simple Provider allows you to transparently serve both content and stylesheets from *zip* archives. (See Example 8-2.)

Example 8-2. Provider::Zip

```
package Provider::Zip;

use strict;
use vars qw($VERSION @ISA);

use Apache::AxKit::Provider::File;
use Archive::Zip qw(:ERROR_CODES);

# Inherit from the default file Provider class
@ISA = ('Apache::AxKit::Provider::File');

Archive::Zip::setErrorHandler(\&_error_handler);

sub _error_handler {
    my $error = shift;
    AxKit::Debug(3, $error);
}

sub exists {
    my $self = shift;
    return defined( $self->{zip_member} );
}

sub mtime {
    my $self = shift;
    return $self->{zip_member}->{lastModFileDateTime};
}

sub process {
    my $self = shift;
    my $zip = Archive::Zip->new( );
    if ($zip->read($self->{file}) != AZ_OK) {
        throw Apache::AxKit::Exception::IO (-text => "Couldn't read archive file
'$self->{file}'");
    }

    my $r = $self->apache_request;
    my $member;

    my $path_info = $r->path_info;
    $path_info =~ s|^/||;

    if ( $self->{zip_uri} ) {
        $member = $zip->memberNamed($self->{zip_uri});
    }
    else {
        if ($path_info) {
            $member = $zip->memberNamed($path_info);
        }
        else {
            $member = $zip->memberNamed('index.xml') || $zip->memberNamed('index.xsp');
        }
    }

    unless ( $member ) {
        throw Apache::AxKit::Exception::Declined(
                -text => "Document could not be retrieved from $self->{file}"
                );
    }

    $self->{zip_member} = $member;
    return 1;
}

sub get_strref {
    my $self = shift;

    my ($data, $status) = $self->{zip_member}->contents( );

    my $r = $self->apache_request( );

    if ($status != AZ_OK) {
        throw Apache::AxKit::Exception::Error(
                -text => "Document could not be retrieved from $self->{file}: $status"
                );
    }

    # Allow images to be served correctly
    if ( $r->path_info =~ /\.(png|gif|jpg)$/ ) {
        my $image_type = $1;
        $r->content_type( 'image/' . $image_type );
        $r->send_http_header( );
        $r->print( $data );
        throw Apache::AxKit::Exception::Declined(
                -text => "Image detected, skipping further processing."
                );
    }

    return \$data;
}

1;
```

Obviously, a production-worthy implementation would be a bit more complex, but the basic functionality exists. Once this custom Provider is installed, you only need to configure AxKit to process zip archives and then to set up special Alias and Location directives for each zip to make browsing the archived content seem transparent:

```
# Set AxKit to process zip archives
AddHandler axkit .zip

# Add an Alias, so the zipped content appears
# to be a native part of the site.
Alias /help /www/sites/myaxkithost/zips/helpdocs.zip

# And set AxKit to use Provider::Zip to fetch both
# content and stylesheets for the zipped help docs.

<Location /help>
  AxProvider Provider::Zip
</Location>
```

With these directives in place, a request to http:://localhost/help/index.xml causes AxKit to extract the file *index.xml* from the top level of the *helpdoc.zip* archive. In addition, any xml-stylesheet processing instructions found in that document whose href attribute pointed to a document at or below that same level in the archive also cause that stylesheet document to be extracted and applied at request time.

8.4 Custom Language Modules

AxKit Language modules provide various ways to transform XML content during delivery. For example, Apache::AxKit::Language::XSP provides AxKit's eXtensible Server Pages implementation, while Apache::AxKit::Language::LibXSLT is one of two Language modules that offers the ability to transform content using XSLT stylesheets. Usually, a Language module is simply a wrapper around the implementation of proven or promising XML transformation technology that allows AxKit to pass data to that processor and capture the result for delivery or further processing.

New Language modules are added to AxKit at runtime using the AxAddStyleMap directive. This directive requires two arguments: the MIME type used to associate transformations with the given Language processor and the Perl package name of the module that implements that processor:

```
# Add the Language module

AxAddStyleMap application/x-mylang Language::MyCustomLanguage



# Then, later, an AddProcessor directive that uses your new Language to apply

# the 'default.mlg' stylesheet

AxAddProcessor application/x-mylang /styles/default.mlg
```

8.4.1 Language API

The Language module interface consists of a single *handler* method expected to encapsulate the entire transformation process. This method is passed, in order: the current Apache object, a reference to the ContentProvider that will pull (or has pulled) in the original XML source, a reference to the StyleProvider that the Language module can use to get sources for any stylesheet to be applied, and a Boolean flag that indicates whether the current Language processor is the last entry in the processing pipeline.

Access to the source content (the data being transformed) varies based on two factors: whether the current Language processor is first in the processing chain, and, if not, whether the previous Language processor stored its result as a string or as an XML::LibXML::Document instance. If the given Language processor is first in the pipeline, it must call one of the ContentProvider instance's accessors methods, *get_fh( )*, *get_strref( )*, or *get_dom( )* to fetch the content to transform—the method used should be based on which is the most efficient for the processor associated with the Language module to consume. If the current Language module is *not* the first processor in the chain, the content was stored by the preceding processor in the Apache instance's pnotes table, either in the xml_string or dom_tree fields, depending on what the previous Language module returned. It sounds a little messy and convoluted, but this approach provides Language modules that are implemented on top of XML::LibXML with an easy way to accept and return preprocessed document instances (avoiding the overhead of serializing the objects to a string and reparsing them at each stage), while still allowing those modules based on other XML libraries to simply consume the content as a string.

```
sub handler {

    my ($r, $content_provider, $style_provider, $last_in_chain ) = @_;



    # Your processor wants its XML as a string



    my $content;



    # Always check for previous processing, first

    if (my $string = $r->pnotes('xml_string') ) {

        $content = $string;

        delete $r->pnotes('xml_string');
    }
    elsif ( my $dom = $r->pnotes('dom_tree') ) {
        $content = $dom->toString;
        delete $r->pnotes('dom_tree');
    }
    # If you make it this far, you are the first in the processing chain
    # and need to use the ContentProvider
    else {
        $content = $content_provider->get_strref;
    }

    my $output

    # The real processing happens here, ending with
    # $output containing the result of the transformation.

    $r->pnotes('xml_string', $output );
    return Apache::Constants::OK;
}
```

What to do with the result of the Language module's transformation also varies. Language modules whose result is returned as a string containing a well-formed XML document should store that result in the xml_string field in the Apache object's pnotes table. If the result is an XML::LibXML::Document instance, it may be stored in the pnotes dom_tree field instead. Modules that store their results in pnotes in this way are most generic, since they can appear at any stage in the processing pipeline.

However, some Language modules should not use pnotes to store their results because of the type of content generated during the transformation process. For example, the AxPoint Language module that ships with AxKit generates PDF slideshows from XML content—a result that would choke any downstream processors expecting to consume a well-formed XML document. In cases in which the result of the transformation cannot be an XML document (or XML::LibXML::Document instance), the Language module is expected to use the Apache object to send the result directly to the requesting client. While doing so limits these modules to the last position in the processing chain, it means that practically any format can be generated (PDF document, binary image, RTF text, etc.). Whether the content is stored in pnotes for further processing or sent directly to the client, the *handler* method should always return OK from the Apache::Constants class to let AxKit know that the process completed successfully

```
sub handler {
    my ($r, $content_provider, $style_provider, $last_in_chain ) = @_;


    unless ( $last_in_chain ) {
        throw Apache::AxKit::Exception::Error(
          -text => _ _PACKAGE_ _ . " is not a generic Language module and may only
appear as the last processor in the chain."
          );
    }


    my $result;
    # The transformation happens here, storing a binary
    # or other non-XML result in $result.

    # You are the last in the chain, so you can
    # send the data directly to the client using the Apache object.
    $r->print( $result );

    return Apache::Constants::OK;
}
```

As an example of an AxKit Language module, let's create an interface to Petr Cimprich's Perl implementation of STX (Streaming Transformations for XML), XML::STX. The code in Example 8-3 reflects an early beta implementation of STX. The AxKit interface used for the final implementation is likely to be quite different.

Example 8-3. Language::STX

```
package Language::STX;

use strict;
use vars qw( @ISA );
use XML::STX;
use XML::SAX::ParserFactory;
use Apache::AxKit::Language;
use XML::SAX::Writer;

@ISA = qw( Apache::AxKit::Language );

sub handler {
    my $class = shift;
    my ($r, $xml, $style, $last_in_chain) = @_;

    my ($xmlstring, $xml_doc);

    if (my $dom = $r->pnotes('dom_tree')) {
        $xml_doc = $dom;
    }
    else {
        $xmlstring = $r->pnotes('xml_string');
        delete $r->pnotes( )->{'xml_string'};
    }
    my $stx_style = undef;

    # get the source for the STX stylesheet from the StyleProvider and parse
    # it to get the compiled XSLT processor.
    my $stx_compiler = XML::STX::Compiler->new( );

    my $parser = XML::SAX::ParserFactory->parser( Handler => $stx_compiler );

    my $style_stringref = $style->get_strref( );

    eval {
        $stx_style = $parser->parse_string( $$style_stringref );
    };


    if ( my $error = $@ ) {
        throw Apache::AxKit::Exception::Error( -text => "Error parsing
STX stylesheet: $error ." );
    }

    my $result = '';
    my $error;
    my $output_handler = XML::SAX::Writer->new( Output => \$result );
    my $stx_handler = XML::STX->new( Handler => $output_handler, Sheet => $stx_style );

    if ( $xml_doc ) {
        eval {
            require XML::LibXML::SAX::Parser;
            my $p = XML::LibXML::SAX::Parser->new( Handler => $stx_handler );
            $p->generate( $xml_doc );
        };
    }
    else {
        my $p = XML::SAX::ParserFactory->parser( Handler => $stx_handler );

        if ($xmlstring) {
            eval {
                $p->parse_string( $xmlstring );
            };
            $error = $@ if $@;
        }
        else {
            $xmlstring = ${$xml->get_strref( )};
            eval {
                $p->parse_string( $xmlstring );
            };
            $error = $@ if $@;
        }
    }

    if ( length( $error ) ) {
        throw Apache::AxKit::Exception::Error(
                    -text => "STX Processing Error: $error"
                    );
    }

    delete $r->pnotes( )->{'dom_tree'};
    $r->pnotes('xml_string', $result);
    return Apache::Constants::OK;
}

1;
```

8.5 Custom ConfigReaders

By default, AxKit gets all its runtime configuration information from a set of extensions to Apache's standard configuration directives. In fact, whether the task simply required adding a new XSP taglib or setting up a complex processor-to-resource style mapping, all example applications that we have examined throughout this book rely on the fact that the default ConfigReader is being used. This does not have to be the case. Apart from the top level PerlModule directive that loads AxKit in the first place and an AxConfigreader directive that loads the new class responsible for loading the configuration data, AxKit's setup can be completely uncoupled from the Apache configuration system.

Reasons for implementing a custom ConfigReader vary. For example, the people responsible for setting up style-to-resource mappings (the part of an AxKit configuration that changes most often) may have limited or no access to the Apache configuration files. Here, providing a different way to set up those mappings while avoiding possible risks associated with doling out write access to *httpd.conf* and *.htaccess* files (or, worse, forcing developers to nag the webmaster for every change) makes everyone's life a bit easier. Or, you may decide that an XML-based configuration system such as the sitemaps in Apache Cocoon 2 fits your needs best—all you need to do is implement a ConfigReader that can extract and process the relevant AxKit configuration information from your XML configuration documents. Custom ConfigReaders are swapped in using the AxConfigReader directive:

```
AxConfigReader My::Custom::ConfigReader
```

8.5.1 ConfigReader API

The ConfigReader class implements a small army of methods responsible for providing AxKit with all its runtime configuration data. From the ContentProvider and StyleProvider classes used to get the data for the current request, to the list of plug-ins that will run, to the default style definitions—every aspect of AxKit processing can be controlled from the methods in the ConfigReader. Given this wide scope, unless you intend to separate AxKit's runtime setup as much as possible from the Apache configuration system, you should consider the wisdom of implementing your custom ConfigReader as a subclass of the default Apache::AxKit::ConfigReader and implementing only those methods required to suit your specific purpose.

8.5.1.1 get_config( )

Called during object initialization and passed a copy of the current Apache object as its sole argument, the *get_config* offers the simplest path to creating a custom Config- Reader class. This method is expected to return a hash reference containing all or some of the data used to control AxKit's configuration. While it is generally better to explicitly override a given option's accessors method, the value returned from the default classes' implementation can be altered by directly setting the appropriate underlying data structure in the reference returned from *get_config*:

```
# Return a simple, mostly hard-coded mapping for certain options

sub get_config {

  my $self = shift;

  my $r = shift;


  my %config = (

      ContentProvider => 'My::Custom::Provider',

      CacheDir          => '/www/mysite/.axkitcache'.

      DebugLevel        => 10,

      );

  return \%config;

}
```

8.5.1.2 StyleMap( )

The *StyleMap* method associates stylesheet MIME types with the Language processor used to apply those stylesheets to the content. It is expected to return a reference to a HASH whose keys are the MIME types and whose values are the package names of the Language modules associated with those types:

```
# a simple, hard-coded mapping

sub StyleMap {

    my %mapping = (

                  'text/xsl' => 'Apache::AxKit::Language::LibXSLT',

                  'application/x-xpathscript' => 'Apache::AxKit::Language::

XPathScript',

                  'application/x-xsp' => 'Apache::AxKit::Language::XSP',

                 );

    return \%mapping;

}
```

8.5.1.3 CacheDir( )

This method defines the directory that AxKit uses to store the cache files for the current request.

8.5.1.4 ContentProviderClass( )

The *ContentProviderClass* is expected to return the package name of the module used to fetch the source XML content for the current request:

```
sub ContentProviderClass {

    my $self = shift;


    # Emulate a FilesMatch block for zip archived content.

    if ( $self->{apache}->uri =~ /\.zip/ ) {

        return 'Provider::Zip';

    }

    else {

        return 'apache::AxKit::Provider::File';

    }

}
```

8.5.1.5 StyleProviderClass( )

Similar to its sister method, *ContentProviderClass*, this method should return the package name of the module responsible for getting the source for any stylesheets to be applied to the content.

8.5.1.6 DependencyChecks( )

By default, AxKit builds a list of file dependencies that it encounters during processing. For example, an XSLT stylesheet that contains an xsl:import element can be said to *depend on* the file that contains the stylesheet being imported. To make caching work as expected, the list of dependencies for each resource is also cached. Then during each subsequent request, the files in the dependency list are examined for changes. If a dependency has changed, the cache for the parent resource is considered invalid, and the sources are reprocessed. This design provides AxKit with its fine-tuned caching facilities, where dynamic or changed content is always fresh, but anything that can be cached is saved from further processing. In most cases, this is exactly the behavior that you want and expect.

In cases in which file dependencies are known to be stable and static, however, even greater cache performance can be gained by configuring AxKit to *skip* its dependency checks and serve cached content based solely on the last modification time of the top-level resources. This behavior is achieved by returning 0 or undef from this method.

8.5.1.7 PreferredStyle( )

The value returned by the *PreferredStyle* method declares the preferred style name for the current resource. This name is passed, along with the value returned from the ConfigReader's *PreferredMedia* method, into the ContentProvider's *get_styles* method. There it is used to match against any default configuration-based named styles or any alternate styles declared via xml-stylesheet processing instructions in the content document. See Section 4.3.1 for an overview of how named styles work and are typically used.

ConfigReader classes implementing this method do well to examine the preferred_style entry in the notes table of the Apache::Request instance for the current request. Otherwise, the default StyleChooser and other plug-in modules that use that property do not work as expected. If no style name applies to the current resource, an empty string should be returned, rather than simply undef.

```
sub PreferredStyle {
    my $self = shift;

    # give the plug-ins preferred_style precedence
    my $style = $self->{apache}->notes('preferred_style') ||
                $self->{some_application_object}->preferred_style_name( ) ||
                '';

    return $style;
}
```

8.5.1.8 PreferredMedia( )

Similar to *PreferredStyle*, the *PreferredMedia* method has final say over the media type name used to map the current resource to the appropriate styles. As with *PreferredStyle*, precedence should be given to any value stored in the preferred_media field in the notes table. That allows the MediaChooser or other plug-in modules configured for the current resource to work as expected. On the other hand, if your goal is to intentionally short-circuit the default behavior of the MediaChoosers for a given case, this is the place to do it—just be sure that you know what you are giving up.

```
sub PreferredMedia {
    my $self = shift;

    # Override the plug-in's preferred_media, but fallback to it
    # if no style is returned by the application object

    my $media = $self->{some_application_object}->preferred_media_name( ) ||
                $self->{apache}->notes('preferred_media') ||

                # explicitly set the default value here.
```

```
            'screen';


    return $media;

}
```

> If all you need to do is pragmatically control the preferred style or media name for the current request, consider using a custom plug-in instead of a ConfigReader.

8.5.1.9 CacheModule( )

This method is expected to return a single scalar containing the name of the Perl package used as the cache module for the current content and stylesheet documents.

8.5.1.10 DebugLevel( )

Expected to return a value between 0 and 10, *DebugLevel* determines the level of detail and number of errors that will appear in the host's error log. The following shows how a subclass of the default ConfigReader may be used to make developers' lives a little easier by allowing the debugging level to be controlled at request time:

```
sub DebugLevel {

    my $self = shift;

    my %params = $self->{apache}->args( );


    # if you get a query string param called 'noisylog'

    # crank the debugging level to maximum


    if ( defined( $params{noisylog} ) ) {

        return 10;

    }


    # otherwise, fallback to the default

    return $self->SUPER::DebugLevel( );

}
```

8.5.1.11 LogDeclines( )

A defined return value from the *LogDeclines* configures AxKit to provide additional information in cases in which a Declined exception has been thrown.

8.5.1.12 HandleDirs( )

If this method returns a defined value, AxKit will offer up an XML representation of the items in a directory listing (which can then be transformed for delivery) instead of the default Apache directory index.

8.5.1.13 IgnoreStylePI( )

The *IgnoreStylePI* method determines whether or not the ContentProvider for the current XML source takes any xml-stylesheet processing instructions contained in that document into account when mapping the document to the current list of styles. If this method returns a nonzero value, the processing instructions are ignored.

8.5.1.14 AllowOutputCharset( )

Returning a defined value from this method indicates the intention and ability to send the final transformed content for the current resource in a character encoding other than the one that the final Language processor may have used. It is used by some of the more esoteric Language processors to avoid corrupting the (usually nontextual) data on the way to the client. Therefore, any implementation of this method must allow the value to be set by the classes that call it, rather than simply returned.

```
sub AllowOutputCharset L
    my $self = shift;


    # Use the method to set as well as get
    if ( @_ ) {
        $self->{allow_output_charset} = shift;
    }


    return self->{allow_output_charset};
}
```

8.5.1.15 OutputCharset( )

The *OutputCharset* method provides a way to configure AxKit to send the final transformed content to the client translated into a specific character encoding. Any value returned by this method causes AxKit to add a character set OutputTransformer to the current process. That value will be used to define the character encoding. Implementations of this method should always examine the return value from *AllowOutputCharset* to ensure that such translation is possible and allowed for the current request.

8.5.1.16 ErrorStyles( )

The *ErrorStyles* method is expected to return a list of style definitions used to transform a generated XML backtrace for the error encountered. If this method returns no style definitions, the XML backtrace is not generated, and the client receives the disappointing default 500 Server Error page instead. See the Section 8.3.1.4 in Section 8.3.1 earlier in this chapter for the details about the structure of the style definitions that this method is expected to return.

8.5.1.17 GzipOutput( )

Returning a defined value from this method indicates the intention to send gzipped content to the requesting client. The *DoGzip* method determines whether that actually happens.

8.5.1.18 DoGzip( )

Given a defined return value from *GzipOutput*, the *DoGzip* method is responsible for examining the requesting client's ability to handle gzipped content. If the client is found incapable, this method should return 0 or undef in spite of any value *GzipOutput* returns.

8.5.1.19 GetMatchingProcessors( )

Called from the ContentProvider's *get_styles* method, *GetMatchingProcessors* offers the ability to alter or define the list of processors applied to the current resource. Assuming that the default Provider class is used, it is passed several arguments:

- The preferred style name as defined by the *PreferredStyle* method
- The preferred media type name as defined by the *PreferredMedia* method
- The PUBLIC identifier from the DOCTYPE declaration in the source XML content
- The SYSTEM identifier from the DOCTYPE declaration in the source XML
- The top-level element name of the source document
- A reference to a list of any matching style found while examining the xml-stylesheet processing instructions in the source document
- A reference to the ContentProvider instance

If one of the default Providers (or subclasses thereof that do not implement the *get_styles* method) is used, the list of styles returned from the *GetMatchingProcessors* overrides the styles extracted from any xml-stylesheet processing instructions contained in the source document. The default ConfigReader gives explicit precedence to these styles by simply returning the list of styles passed in from the Provider if that list is not empty:

```
sub GetMatchingProcessors {
    my $self = shift;
    my ($media_name, $style_name, $dtd_public, $dtd_system, $root_name,
$styles_list, $provider) = @_;

    # give preference to the Provider is it found any matching style
    # in the processing instructions
    return @$styles_list if @$styles_list;
     . . .
}
```

See the Section 8.3.1.4 for more details about the structure and properties of the style definitions that this method is expected to return.

8.5.1.20 XSPTaglibs( )

Used by the XSP Language module to load the desired tag libraries for the current scope, the *XSPTaglibs* method is expected to return a list of the Perl package names that implement those libraries. If the XSP processor is invoked and the modules returned for this method are not yet loaded into memory, each is registered and cached into memory on the first request.

8.5.1.21 OutputTransformers( )

The *OutputTransformers* method is expected to return a list of Perl package names that define the OutputTransformer classes that will be applied to the content on its way to the client. Each class is called in turn, in the order returned from this method.

8.5.1.22 Plugins( )

The *Plugins* method is expected to return a list of the Perl package names for the plug-in modules to run for the current resource. These packages are loaded and called in the order returned from this method. For example, if you are mostly happy with using the default Apache directive-based configuration but have a plug-in that you want to be run for every request *after* any other plug-ins set up via the configuration directive, you may subclass the default ConfigReader and do the following:

```
sub Plugins {
    my $self = shift;


    # Get the list from the default implementation
    my @plugins = $self->SUPER::Plugins( );


    # And append your global plug-in to the end
    push @plugins, 'Plugin::MyGlobalPlugin';


    return @plugins;
}
```

As an example ConfigReader, let's create a subclass of the default that allows each method to be used to *set* each property explicitly. (See Example 8-4.) The goal is to provide access to the ConfigReader interface from plug-ins and other classes that may want to override certain aspects of the current configuration for a special case, while falling back to the default ConfigReader where a given property is not expressly set in the local instance. Note the addition of a *push_style* method that allows style definitions to be added to the local instance's list of styles and the localized *GetMatchingProcessors* method that returns those styles.

Example 8-4. ConfigReader::OpenAPI

```
package ConfigReader::OpenAPI;
use strict;
use Devel::Symdump;
use Apache::AxKit::ConfigReader;
use vars qw( @method_list @ISA );
@ISA = qw( Apache::AxKit::ConfigReader );


BEGIN {
    @method_list = Devel::Symdump->functions( 'Apache::AxKit::ConfigReader' );


    foreach my $method_name (@method_list) {
        my $full_mname = $method_name;
        $method_name =~ s/.+:://;
        next if grep { $method_name eq $_ } qw/ new GetMatchingProcessors dirname basename
 get_config /;
        my $full_name = _ _PACKAGE_ _ . "::$method_name";


        # Avert your eyes if hacking the symbol table scares you
        {
          no strict 'refs';
          *{$full_name} =
              sub { my $self = shift;
                  if ( @_ ) {
                      $self->{$method_name} = shift;
                  }

                  unless ( defined( $self->{$method_name} ) ) {
                      return &$full_mname($self);
                  }

                  if ( wantarray and ref( $self->{$method_name} ) eq 'ARRAY' ) {
                      return @{$self->{$method_name}};
                  }
                  else {
                      return $self->{$method_name};
                  }
                  };
        }
        }
}



sub GetMatchingProcessors {
    my $self = shift;
    my ( $medianame, $stylename, $dtd_public, $dtd_system, $root, $styles, $provider ) = @_;
    return @$styles if @$styles;

    if ( defined( $self->{style_defs} )) {
         return @$self->{style_defs};
    }
    else {
        return $self->SUPER::GetMatchingProcessors($medianame, $stylename, $dtd_public,
                                            $dtd_system, $root, $styles, $provider);
    }
```

```
}

sub push_style {
    my $self = shift;
    $self->{style_defs} ||= [  ];
    push @{$self->{style_defs}}, @_;
}

1;
```

After this module is installed and configured, each AxKit runtime option can be set directly from within any other component class—the most likely candidate being a plug-in:

```
# Add your ConfigReader
AxConfigReader ConfigReader::OpenAPI

# Now, add a plug-in to take advantage of the open interface:
AxAddPlugin Plugin::OpenAPI
```

Here's a sample plug-in that uses the more open configuration interface:

```
package Plugin::OpenAPI;

use strict;
use AxKit;
use Apache::Constants qw(OK);
use vars qw( $config );
sub handler {
    my $r = shift;

    local $config = $AxKit::Cfg;

    # $config now contains a reference to the
    # OpenAPI ConfigReader that you can use
    # set the various runtime options.

    $config->AllowOutputCharset(1);
    $config->OutputCharset( 'Shift_JIS' );
    # etc..
    return OK;
}

1;
```

8.6 Getting More Information

Given the many ways that custom implementations of AxKit's components can be combined to meet specific needs, there's no way that the introductory module and method samples in this chapter can do more than hint at what you can do with AxKit, if you are willing to get your hands a little dirty. If you intend to extend AxKit in earnest, other sources of information can prove invaluable.

First, examine the code that is there. When getting a sense of how you can implement a given custom component, nothing is quite so instructive as stepping through one of the default implementations of that component type. Often, you will find that one of the default components does 90% of what you need to accomplish and that simply subclassing that module and covering the variance is all that's required.

Also, be sure to join the axkit-devel and axkit-users mailing lists and share your ideas. Knowing the easiest way from point A to point B is mostly a result of experience—avail yourself of the AxKit communities' wealth of talent, experience, and wild ideas. Topics range from nuts-and-bolts code-level discussions of AxKit itself to general XML web-publishing best practices and nifty AxKit tricks. You can subscribe to these lists by going to http://axkit.org/mailinglist.xml and following the directions there.

Chapter 8. Extending AxKit

For many developers, AxKit will be a black box through which they transform and serve XML content. They will neither know nor care about its lower-level components or how things work behind the curtain. That's fine. After all, one need never know anything about lasers to use a CD player for its intended purpose. If this "I don't care, so long as it works" approach characterizes your interest in AxKit, you may want to skim the architectural overview presented shortly and skip the gory details that make up the balance of this chapter. If, on the other hand, you are the type of person who takes the "serviceable by authorized technician only" sticker on a CD player as a personal challenge, then this chapter is for you.

9.1 The Template Toolkit

Full of built-in features and optional extensions, Andy Wardley's Template Toolkit has proven a popular choice for web templating. Although TT2 is useful for generating practically any format, it is especially suited for generating and transforming XML content.

The Template Toolkit implements a modest set of simple directives that provide access to common templating system features: inline variable interpolation, inclusion of component templates and flat files, conditional blocks, loop constructs, exception handling, and more. Directives are combined and mixed with literal output using [% . . . %] as default delimiters. We will not cover TT2's complete syntax and feature set in any detail here (it would likely fill a book), so if you are not familiar with it, visit the project home page at http://template-toolkit.org/. A simple TT2 template looks like this:

```
<html>

<head>

  [% INCLUDE common_meta title="My Latest Rant" %]

</head>

 <body>

  [% INSERT static_header %]



  <p>

    Pleurodonts! Crinoids! Wildebeests! Lend me your, um, ears . . .

  </p>



  [% INSERT static_footer %]

 </body>

</html>
```

This simple HTML template includes and processes a separate common_meta template, setting that template's local title variable. It then inserts static page header and footer components around the document's main content. The directive syntax is not Perl, though it is rather Perl-minded (and you can embed raw Perl into your templates by setting the proper configuration options, if you want), nor is it really like XML.

Part of what makes the Template Toolkit a good fit with an XML environment is precisely that its directive and default delimiter syntax is so markedly different from the syntax of both Perl and XML. For example, XSP and other XML grammars that use Perl as the embedded language often require logic blocks to be wrapped in CDATA sections to avoid potential XML well-formedness errors caused by common Perl operators and "here" document syntax. Similarly, documents created for templating systems that use angle brackets as delimiters often cannot be processed with XML tools. Those that can often require the use of XML namespaces to distinguish between markup that is part of the application grammar and markup meant to be part of the output. The Template Toolkit's syntax deftly avoids these potential annoyances by limiting its operators to a handful of markup-friendly characters and letting different things (embedded directives) be and look substantially different from the literal output.

When considering how the Template Toolkit can be used within the context of AxKit, it is natural to first think of creating a Provider class that simply returns the content generated from the template expansion; it would be easy enough to do so. However, a closer look at the Template Toolkit's features shows that it can be useful for *transforming* XML content, as well. This means that it is probably best implemented as an AxKit Language module, rather than a Provider. In fact, integrating the Template Toolkit via the Language interface creates a unique medium that offers an alternative to both XSP (for generating content), and XSLT and XPathScript (for transforming it).

Based on what you already know from the details covered in Section 8.3, creating an AxKit Language interface for the Template Toolkit is pretty straightforward—you simply store the result of the template process in the appropriate Apache pnotes field. But to pull double duty as both a generative and transformative language, the Template Toolkit Language module needs to be a bit smarter. That is, in cases in which you would use the language to generate content, the source XML itself contains the template to be processed. There is no external stylesheet. When you want to use the Template Toolkit's powers to transform XML, then an external template is required. To be truly useful, the AxKit Template Toolkit Language module needs to be able to distinguish between these two cases. Example 9-1 is an excerpt from a simplified version of the Apache::AxKit::Language::TemplateToolkit2 class.

Example 9-1. Language::TemplateToolkit2

```
package Language::TemplateToolkit2;


use strict;

use vars qw( @ISA );

use Apache::AxKit::Language;

use Template;

@ISA = qw( Apache::AxKit::Language );


sub handler {

    my $class = shift;

    my ($r, $xml, $style, $last_in_chain) = @_;


    AxKit::Debug( 5, 'Language::TT2 called');


    my $input_string = undef;

    my $xml_string = undef;

    my $result_string = '';


    my $params = { Apache => $r };




    if (my $dom = $r->pnotes('dom_tree')) {

         $xml_string = $dom->toString;    }

    else {        $xml_string = $r->pnotes('xml_string');

        delete $r->pnotes( )->{'xml_string'};

    }

    $xml_string ||= ${$xml->get_strref( )};


    # process the source as the template, or a param passed to

    # an external template

    if ( $style->{file} =~ /NULL$/ ) {

        $input_string = $xml_string;

    }

    else {

        $params->{xmlstring} = $xml_string;

        $input_string = ${$style->get_strref( )};
    }


    my $tt = Template->new( get_tt_config($r) );

    $tt->process(\$input_string, $params, \$result_string ) ||

        throw Apache::AxKit::Exception::Error( -text => "Error processing TT2

template: " . $tt->error( ) );


    delete $r->pnotes( )->{'dom_tree'};

    $r->pnotes('xml_string', $result_string);

    return Apache::Constants::OK;

}


# helper functions, etc.

1;
```

Essentially, this module's main *handler( )* method examines the path to the stylesheet for the current transformation. If that option contains the literal value NULL as the final step, then the source content is expected to contain the template document to be processed, much the same way that XSP works. Otherwise, that path is used as the location of the external template to which the source content is passed as a parameter, named xmlstring, for further processing. This allows you to use the Template Toolkit in both generative and transformative contexts while using AxKit's configuration directive syntax and conventions to differentiate between the two cases. The following configuration snippet illustrates both uses:

```
# Add the Language module

AxAddStyleMap application/x-tt2 Apache::AxKit::Language::TemplateToolkit2


# TT2 as a generative Language such as XSP (source defines the template)

AxAddProcessor application/x-tt2 NULL


# TT2 as a transformative Language such as XSLT (external template applied to source)

AxAddProcessor application/x-tt2 /styles/mytemplate.tt2
```

9.1.1 Generating Content

Generating content using the Language::TemplateToolkit2 module follows the most common uses of the Template Toolkit, in general. It combines literal markup with special directives to construct a complete result when processed by the Template engine. Used in this way, the top-level template acts like an XSP page; that is, no external stylesheet is applied. The source document itself is what gets processed. Because the template is the source content, it must be a well-formed XML document, since AxKit will extract the root element name, document-type information, and any xml-stylesheet processing instructions that may be contained there for use with the style processor configuration directives (as it does with all content sources).

```
<?xml version="1.0"?>

<products>

    [% USE DBI( 'dbi:driver:dbname', 'user', 'passwd' ) %]

    [% FOREACH product DBI.query( 'SELECT * FROM products' ) %]
    <product id="[% product.id %]">
      <name>[% product.name %]</name>
      <description>[% product.description %]</description>
      <price>[% product.price %]</price>
      <stock>[% product.stock %]</stock>
    </product>
    [% END %]
</products>
```

This simple template uses the Template Toolkit's DBI plug-in to select product information from a relational database and generate an appropriate product element for each row returned. To configure AxKit to process this template as expected, you only need to add the correct directives to the host's *httpd.conf* or *.htaccess* file:

```
<Location /products.xml>
  AxAddProcessor application/x-tt2 NULL</emphasis>
  AxAddProcessor text/xsl /styles/generic.xsl
</Location>
```

Or, if you do not mind hardcoding styling information into the document itself, you can achieve the same effect by adding xml-stylesheet processing instructions to the template.

```
<?xml version="1.0"?>
<?xml-stylesheet type="application/x-tt2" href="NULL"?>
<?xml-stylesheet type="text/xsl" href="/styles/generic.xsl"?>]]><products>
    [% USE DBI( 'dbi:driver:dbname', 'user', 'passwd' ) %]
    [% FOREACH product DBI.query( 'SELECT * FROM products' ) %]
    <product id="[% product.id %]">
      <name>[% product.name %]</name>
      <description>[% product.description %]</description>
      <price>[% product.price %]</price>
      <stock>[% product.stock %]</stock>
    </product>
    [% END %]
</products>
```

9.1.2 Transforming Content

While XSP is solely concerned with generating content, TT2's filtering capabilities make it useful as a transformative language as well:

```
<html>
 <head><title>Our Products</title></head>
[% USE xpath = XML.XPath(xmlstring) %]
<body>
  <h1>Our Products</h1>
  [% PROCESS productlist %]
</body>
</html>

[% BLOCK productlist %]
  <table>
    <tr>
     <th>ID</th><th>Name</th><th>Description</th><th>Price</th><th>In Stock</th>
    </tr>
    [% FOREACH product = xpath.findnodes('/products/product') %]
       [% PROCESS product item = product %]
    [% END %]
  </table>
[% END %]

[% BLOCK product %]
  <tr>
    <td>[% item.getAttribute('id') %]</td>
    <td>[% item.findvalue('name') %]</td>
    <td>[% item.findvalue('description') %]</td>
    <td>[% item.findvalue('price') %]</td>
    <td>[% item.findvalue('stock') %]</td>
  </tr>
[% END %]
```

This is how the same transformation can be done with XSLT:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
  <html>
  <head><title>Our Products</title></head>
  <body>
     <h1>Our Products</h1>
     <xsl:call-template name="productlist"/>
  </body>

</html>

</xsl:template>


<xsl:template name="productlist">

  <table>

    <tr>

     <th>ID</th><th>Name</th><th>Description</th><th>Price</th><th>In Stock</th>

    </tr>

    <xsl:for-each select="/products/product">

       <xsl:apply-templates select="."/>

    </xsl:for-each>

  </table>

</xsl:template>


<xsl:template match="product">

  <tr>

    <td><xsl:value-of select="@id"/></td>

    <td><xsl:value-of select="name"/></td>

    <td><xsl:value-of select="description"/></td>

    <td><xsl:value-of select="price"/></td>

    <td><xsl:value-of select="stock"/></td>

  </tr>

</xsl:template>


</xsl:stylesheet>
```

If you favor another templating system and want to make it available as an AxKit Language module, please see Section 8.4 for more detail about creating your own custom AxKit Language module.

9.2 Providing Content via Apache::Filter

Filter chains are made available by adding the Apache::Filter extension to *mod_perl* via a PerlModule configuration directive to your *httpd.conf* file. Then, to set the filter chain for a specific resource, you set *mod_perl* as the main Apache handler and pass a whitespace-separated list of Perl classes to the PerlHandler directive. That done, the output of the first Perl content handler is sent as the input to the next, and so on (similar to AxKit's own style transformation chains):

```
PerlModule Apache::Filter


<Location /path/to/my/app>

  SetHandler perl-script

  PerlSetVar Filter On

  PerlHandler Filter1 Filter2 Filter3

</Location>
```

AxKit's standard distribution includes a simple, but powerful, Provider class, Apache::AxKit::Provider::Filter, that uses the Apache::Filter interface to capture the result of one or more *mod_perl* content handlers for further processing inside AxKit. This means that any Apache::Filter-aware handler can be used to provide XML content to AxKit.

AxKit requires that data passed in through the ContentProvider interface be a well-formed XML document. That does not mean that the source of that content must be XML. As long as the *returned result* is well-formed XML, you are free to use whatever means necessary to generate it. AxKit does not care how the XML gets there, only that it does.

9.2.1 CGI

First, let's examine how you can use a basic Perl CGI script to generate XML that is transformed and delivered by AxKit. The script itself is quite straightforward: it uses Perl's built-in *print( )* function to generate a simple XML document with a top-level root element that contains a series of field elements. Each field element contains a name attribute that holds the name of one of the server's environment variables as well as text that contains the value of that variable:

```
#!/usr/bin/perl

use strict;


print qq*<?xml version="1.0"?>

<root>

*;


foreach my $field ( keys(%ENV) ) {
   print qq*<field name="$field">$ENV{$field}</field>*;

}


print qq*</root>

*;


1;
```

Running this script as a CGI inside Apache produces a document like the following:

```
<?xml version="1.0"?>

<root>

    <field name="SCRIPT_NAME">/axkitbook/samples/chapt09/env.cgi</field>

    <field name="HTTP_ACCEPT_ENCODING">gzip,deflate</field>

    <field name="HTTP_CONNECTION">keep-alive</field>

    <field name="REQUEST_METHOD">GET</field>

    <field name="SCRIPT_FILENAME">/www/site/axkitbook/samples/chapt09/env.cgi</field>

    <field name="HTTP_ACCEPT_CHARSET">ISO-8859-1,utf-8;q=0.7,*;q=0.7</field>

    <field name="QUERY_STRING"></field>

    <field name="REQUEST_URI">/axkitbook/samples/chapt09/env.cgi</field>

    <field name="GATEWAY_INTERFACE">CGI-Perl/1.1</field>

    <!-- and so on -->

</root>
```

For this experiment to fly, you need to transform the generated XML into a form easily viewable in all web browsers. Applying the following XSLT stylesheet transforms the generated XML into an HTML document containing a two-column table of key/value pairs. For added visual clarity, the background of every second table row will be light silver.

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="/">

  <html>

  <head><title>Server Environement</title></head>

  <body>

     <xsl:apply-templates/>

  </body>

</html>

</xsl:template>


<xsl:template match="root">

  <table>
  <tr><th>Key</th><th>Value</th></tr>

     <xsl:apply-templates/>

  </table>

</xsl:template>


<xsl:template match="field">

<tr>

  <!-- alternate row colors -->

  <xsl:if test="position( ) mod 2 != 1">

    <xsl:attribute name="bgcolor">eeeeee</xsl:attribute>

  </xsl:if>


  <td><xsl:value-of select="@name"/></td>

  <td><xsl:value-of select="."/></td>

</tr>

</xsl:template>


</xsl:stylesheet>
```

You need to place the XML generated from the *env.cgi* script into AxKit so that the XSLT stylesheet can be applied. To do this, add a few key directives to the host's *httpd.conf* or a local *.htaccess* file. The following shows a sample snippet with a <Files> block containing the necessary directives:

```
<Files env.cgi>

 Options ExecCGI

 SetHandler perl-script

 PerlSetVar Filter On

 PerlHandler Apache::RegistryFilter AxKit

 AxContentProvider Apache::AxKit::Provider::Filter

 AxAddProcessor text/xsl styles/env.xsl

</Files>
```

First, set *mod_perl* as the main Apache handler for all requests by passing the value perl_script to the SetHandler directive. Next, enable content filtering by using the PerlSetVar directive to set the Perl variable Filter to On and by passing the two Perl handlers that you want to run to the PerlHandler directive—in this case, Apache::RegistryFilter and AxKit. Apache::RegistryFilter is simply a version of the Apache::Registry handler that allows most Perl CGI scripts to run unaltered inside the stricter *mod_perl* environment and that takes advantage of the Apache::Filter interface. Then, set Apache::AxKit::Provider::Filter as AxKit's ContentProvider, so the data returned from the CGI script will be passed into AxKit via the Apache::Filter. Finally, configure AxKit to apply the *env.xsl* XSLT stylesheet to that generated content by passing the appropriate MIME type and file path to the AxAddProcessor directive.

You can use this same basic configuration pattern to plug AxKit into any Perl handler that implements the appropriate Apache::Filter hooks. Once the Apache::Filter extension is loaded, setting Apache::AxKit::Provider::Filter as the ContentProvider for a given resource invisibly passes on the result of the upstream Perl handlers into AxKit; the only thing that often changes list of Perl handlers themselves.

9.2.2 Apache::ASP

Intended as a direct port of Microsoft's Active Server Pages technology, Apache::ASP offers a complete implementation of that model for use in an Apache and *mod_perl* environment. A predictable, object-oriented API provides easy access to session, application, and server data. Although Apache::ASP offers its own facilities for transforming generated content with XSLT, it assumes that you either need just one predetermined stylesheet per resource or that you will build conditional transformations by calling out to an XSLT processor directly from within the page's code. In any case, Apache::ASP's basic goals differ from AxKit's and therefore lack the extreme flexibility in chaining and choosing styles at runtime. Fortunately, the existence of Apache::Filter and AxKit's Filter Provider class means that developers can take advantage of the strengths of both environments. Here is a tiny page that reimplements the CGI script from the previous example as an ASP page:

```
<?xml version="1.0"?>

<root>

<%

my $env = $Request->ServerVariables;



foreach my $field ( keys( %$env) ) {

    $Response->Write ( qq*<field name="$field">$env->{$field}</field>* );

}

%>

</root>
```

The configuration block needed to enable the ASP-generated XML content to be processed by AxKit looks almost identical to the block used for the CGI example. In this case, a list of handlers configures Apache to process the document using Apache::ASP (instead of Apache::RegistryFilter), before passing the result to AxKit:

```
<Files env.asp>

 SetHandler perl-script

                              PerlHandler Apache::ASP AxKit

 PerlSetVar Filter On

 AxContentProvider Apache::AxKit::Provider::Filter

 AxAddProcessor text/xsl styles/env.xsl

</Files>
```

9.2.3 HTML::Mason

Another popular *mod_perl* web-application framework is HTML::Mason. As with ASP and XPathScript, Mason combines literal markup and inline Perl code, set off by special delimiters, to construct a dynamic result. Syntactically, Mason differs a bit from the others in that it offers more than one type of delimiter. Lines beginning with `%` are treated as free-form Perl code for creating conditional blocks. Blocks delimited by `<% . . . %>` are used for interpolating generated data into the result. Expressions contained by `<& . . . &>` are treated as paths and arguments passed to Mason components. Components are smaller, reusable bits of code and markup combined by higher-level components to create the final result. Much of Mason's popularity is based on the flexibility and modular extensibility that its component-based approach provides.

The following shows a simple Mason component that returns the list of server environment variables in the form from the previous two examples. By default, Mason looks for components, starting at the current host's DocumentRoot. Create a directory called `mason_components` inside that directory and save the following as a plain-text file named *env*:

```
% while (my ($key,$value) = each(%ENV)) {

  <field name="<% $key %>"><% $value %></field>

% }
```

You need the document that calls the env component. You obtain this by passing the path to components between a pair of special component delimiters. And, since that component returns only a flat list of elements, you need to wrap its interpolated value in a single top-level element to ensure that the resulting document is well-formed XML:

```
<?xml version="1.0"?>

<root>

<& /mason_components/env &>

</root>
```

The configuration block required to hand off your Mason-generated XML to AxKit for further processing is essentially identical to those from the previous two examples: set *mod_perl* as the Apache handler, toggle on the Apache::Filter extension, define the chain of Perl handlers, and set AxKit's ContentProvider to the Apache::AxKit::Provider::Filter class. Again, the only difference is that you pass the document through HTML::Mason::ApacheHandler instead of Apache::ASP or Apache::RegistryFilter, before pulling it into AxKit.

```
<Files env.html>

 SetHandler perl-script

 PerlHandler HTML::Mason::ApacheHandler AxKit

 PerlSetVar Filter On

 AxContentProvider Apache::AxKit::Provider::Filter

 AxAddProcessor text/xsl styles/env.xsl

</Files>
```

The flexibility and power provided by chaining AxKit together with other larger application frameworks, as you have done here, does not come without cost. Depending on the feature set of the framework in question, your web server processes can grow shockingly large, and you probably will not be able to serve lots of dynamic content for a popular site from a single, repurposed desktop PC. On the other hand, developer time is typically more costly than new hardware. If you or members of your team are creating application content that's been proven to be productive with a tool such as Mason, then there's no reason to abandon it. You can use Mason for what it is good for while taking advantage of the features that AxKit offers. At the very least, the Apache::Filter-based solutions discussed here provide a gentle migration path for those who want to experiment with AxKit but don't want to learn XSP or XPathScript, or how to write custom Provider classes right away.

Chapter 9. Integrating AxKit with Other Tools

Perl excels at munging textual data. It was specifically designed to simplify the task of extracting and reporting data from text-based formats. As a result, it offers many built-in high-level text-processing facilities. Perl's nature is quite liberal. It presumes (rightly, I believe) that most people who write programs on a daily basis are *not* classically trained computer scientists. Therefore, it strives to be useful at every level of expertise. Given these features and the reality that the vast majority of web-programming tasks involve extracting data and presenting it in a textual format (often HTML markup) under tight deadlines, it's no wonder that Perl enjoys wide success as a web-development technology. And, by extension, you should not be surprised to find CPAN (the Perl community's extension repository) bursting with modules and packages that attempt to streamline the mundane details of web publishing.

Among the most popular types of Perl web modules are templating systems and web-application frameworks. Though the two often overlap in practice, you can distinguish between them. Templating systems are concerned solely with generating consistent document content. Web-application frameworks typically do the same but also provide a development environment that simplifies common back-end programming tasks (database access, etc.), reduces redundancy, and fosters good coding practices. From this perspective, AxKit itself is perhaps best seen as an application framework with built-in support for a variety of XML-centric templating systems.

Choosing the right system is not trivial. At the very least, it takes time to learn a given tool's syntax and common patterns well enough to decide whether they suit your needs and style. If you are like most coders, once you find a tool that flies, you tend to stay with it. If, for example, you learn the ins and outs of a tool such as Mason, and it works well for the project at hand, then you are more likely to use it for your next project and less likely to examine other solutions. On the positive side, this loyalty is the very foundation of healthy and active open source project; like-minded developers congregate around a given system, and their accumulated knowledge, code contributions, and advocacy benefit the larger user community and help bring new developers on board. On the negative side, blinkered devotion to one toolset often means that other, perhaps more suitable, tools and techniques are dismissed in favor of what is familiar.

Fortunately, AxKit does not make you choose. This chapter examines its modular, pluggable design means that many popular frameworks and templating systems can easily be integrated to provide content for, or transform content within, AxKit.

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 animals on the cover of XML Publishing with AxKit are tarpans (Equus caballus gmelini). Now extinct (the last known true tarpan died in captivity in 1876), the tarpan was the original, prehistoric European wild horse. Tarpans lived in southern France and in Spain and eastward toward central Russia, and they inhabited the ancient forests and wetlands of Poland until the 18th century. Peasants who captured and tamed these wild horses would crossbreed them with the local domestic horse. The tarpans that are around today are genetic recreations of the original wild breed, using several European pony breeds that were descendants of the prehistoric tarpan. The tarpan's body is smoky gray in color, with a darker face and legs. Though it is considered a small horse, it has a large head, massive jaws, and a thick neck.

Tarpans became extinct because of the destruction of the forest, their natural habitat, to make room for villages, cities, and agriculture for the growing European human population. There are about 50 tarpans in North America today and possibly only 100 in the world. Most of the tarpans are in the United States and are owned by private individuals. They are friendly, curious, and affectionate and have a very calm disposition, which makes them suitable for children to ride; they are also presently being used in programs to aid the mentally and physically handicapped. Tarpans are very strong, sturdy, and agile, and can cover long distances without horseshoes.

Matt Hutchinson was the production editor for XML Publishing with AxKit . GEX Publishing Services provided production support. Darren Kelly, Emily Quill, Jamie Peppard, and Claire Cloutier provided quality control.

Emma Colby designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Library of Natural History, Volume 2. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

Melanie Wang designed the interior layout, based on a series design by David Futato. 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 Janet Santackas.

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.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

Who Should Read This Book

This book is intended to be useful to any web developer/designer interested in learning about XML publishing, in general, and the practical aspects of XML publishing, specifically with the Apache AxKit XML application and publishing server. While AxKit and its techniques are the obvious focus, many ideas presented can be reused in other XML-based publishing environments. If you do not know XML and dread the thought of consuming a pile of esoteric specifications to understand what is being presented, don't worry—this book takes a fiercely pragmatic approach that will teach you only what you need to know to be productive with AxKit. A quick scan of XML's basic syntax is probably all the XML knowledge you need to get started.

Although AxKit is written in Perl, its users need not know Perl at all to use it to its full effect. However, developers who do know Perl will find that AxKit's modular design allows them to easily write custom extensions to meet specialized requirements. Similarly, AxKit users are not expected to be Apache HTTP server gurus, but those who do know even a bit about how Apache works will find themselves with a valuable head start:

- Web developers will learn XML publishing techniques through a variety of practical, tested examples.
- Perl programmers will see how they can use XML to build on their existing skills.
- Markup professionals will discover how AxKit combines standard XML processing tools with those unique to the Perl programming language to create a flexible, easy-to-use environment that delivers on XML's promise as a publishing technology.

What's Inside

This book is organized into nine chapters and one appendix.

Chapter 1, *XML as a Publishing TEchnology*, puts XML into perspective as a markup language, presents some of the topics commonly associated with XML publishing, and introduces AxKit as an XML application and publishing environment.

Chapter 2, *Installing AxKit*, guides you through the process of installing AxKit, including its dependencies and optional modules. This chapter also covers platform-specific installation tips, how to navigate AxKit's installed documentation, and where to go for additional help.

Chapter 3, *Your First XML Web Site*, guides you through the process of creating and publishing a simple XML-based web site using AxKit. Special attention is paid to the basic principles and techniques common to most projects.

Chapter 4, *Points of Style*, details AxKit's style processing directives. It gives special attention to how to combine various directives to create both simple and complex processing chains, and how to conditionally apply alternate transformations using AxKit's StyleChooser and MediaChooser plug-ins.

Chapter 5, *Transforming XML Content with XSLT*, offers a "quickstart" introduction to XSLT 1.0 and how to use it effectively within AxKit. A Cookbook-style section offers solutions to common development tasks.

Chapter 6, *Transforming XML Content with XPathScript*, introduces AxKit's more Perl-centric alternative to XSLT, XPathScript. The focus is on XPathScript's basic syntax and template options for generating and transforming XML content. The chapter also contains a Cookbook-style section.

Chapter 7, *Serving Dynamic XML Content with XPathScript*, presents a number of tools and techniques that can be used to generate dynamic XML content from within AxKit. The focus is on AxKit's implementation of eXtensible Server Pages (XSP) and on how to create reusable XSP tag libraries that map XML elements to functional code, as well as on how to use Perl's SAWA web-application framework to provide dynamic content to AxKit.

Chapter 8, *Extending AxKit*, introduces AxKit's underlying architecture and offers a detailed view of each of its modular components. The chapter pays special attention to how and why developers may develop custom components for AxKit and provides a detailed API reference for each component class.

Chapter 9, *Integrating AxKit with Other Tools*, shows how to use AxKit in conjunction with other popular web-development technologies, from plain CGI to Mason and the Template Toolkit.

Appendix A, *The AxKit Configuration Directive Reference*, provides a complete list of configuration blocks and directives.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and Unix utilities.

`Constant width`

Indicates commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, XML tags, HTML tags, macros, the contents of files, or the output from commands.

`Constant width italic`

Shows text that should be replaced with user-supplied values.

`Constant width bold`

Shows commands or other text that the user should type literally.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and by quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*XML Publishing with AxKit*, by Kip Hampton. Copyright 2004 O'Reilly Media, Inc., 0-596-00216-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

How to Contact Us

We at O'Reilly have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

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

We have a web page for this book, where we list errata, examples, or any additional information. You can access this page at:

> http://www.oreilly.com/catalog/xmlaxkit/

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

> bookquestions@oreilly.com

You can sign up for one or more of our mailing lists at:

> http://elists.oreilly.com

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

> http://www.oreilly.com

You may also write directly to the author at khampton@totalcinema.com.

Acknowledgments

I would like to thank my editor, Simon St. Laurent, for his wisdom and feedback, and the good folks at O'Reilly for standing behind this book and seeing it through to completion. Thanks to Matt Sergeant for coding AxKit in the first place and to Matt, Barrie Slaymaker, Ken MacLeod, Michael Rodriguez, Grant McLean, and the many other members of the Perl/XML community for their tireless efforts and general markup processing wizardry. Thanks, and a hearty and heartfelt "DAHUT!" to Robin Berjon, Jörg Walter, Michael Kröll, Steve Willer, Mike Nachbaur, Chris Prather, and the other cryptid denizens of the AxKit cabal. Finally, special thanks go out to my family, especially to my brother, Jason, whose patience, support, and encouragement truly made this book possible.

Preface

This book introduces Apache AxKit, a *mod_perl*-based extension to the Apache web server that turns Apache into an XML publishing and application environment.