# Programming HTML5 Applications

# Programming HTML5 Applications

*Zachary Kessin*

**Programming HTML5 Applications**
by Zachary Kessin

**Printing History:**

# Table of Contents

# Preface

## Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*
> Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
> Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

**`Constant width bold`**
> Shows commands or other text that should be typed literally by the user.

*`Constant width italic`*
> Shows text that should be replaced with user-supplied values or by values determined by context.

> This icon signifies a tip, suggestion, or general note.
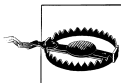
> This icon indicates a warning or caution.

## 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 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: "*Book Title* by Some Author. Copyright 2008 O'Reilly Media, Inc., 978-0-596-xxxx-x."

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*.

## Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

## How to Contact Us

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

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, and any additional information. You can access this page at:

*http://www.oreilly.com/catalog/0636920015116*

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

*bookquestions@oreilly.com*

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

*http://www.oreilly.com*

# The Web as Application Platform

HTML5 makes the Web a first-class environment for creating real applications. HTML5 reinforces JavaScript's existing toolset with key extensions to the browser APIs that make it easier to create applications that feel (and can be) complete in themselves, not just views on some distant server process.

The Web began as a way to share files, stored on a web server, that changed only occasionally. Developers quickly figured out how to generate those files on the fly, taking the first big step toward building applications. The next big step was adding interactivity in the browser client. JavaScript and the Document Object Model (DOM) let developers create Dynamic HTML, as the "browser wars" raged and then suddenly stopped. After a few years, Ajax brought these techniques back into style, adding some tools to let pages communicate with the server in smaller chunks.

HTML5 builds on these twenty years of development, and fills in some critical gaps. On the surface, many of HTML5's changes add support for features (especially multi-media and graphics) that had previously required plugins, but underneath, it gives JavaScript programmers the tools they need to create standalone (or at least more loosely tethered) applications using HTML for structure, CSS for presentation, and JavaScript for logic and behavior.

## Developing Web Applications

In the old days, a complex web application might be a catalog, which would be static pages derived from a database, or a JavaScript loan calculator. But no one would have dreamed of doing complex applications in JavaScript. Those required Java or maybe a dedicated client/server application written in C or C++. Indeed, in the days before the DOM and Ajax, doing so in JavaScript would have been pretty much impossible. However Ajax introduced the ability to interact with the server without reloding the page, and the DOM allowed the programmer to change HTML on the fly.

In 2007 Google introduced Gears, a browser extension that gave the developer a lot more power than had been there before. Gears allowed the browser to work offline, to

store more data in the browser and have a worker pool to ofload long running tasks. Gears has since been discontinued, as most of its features have migrated into HTML5 in modified forms.

The modern web features a full range of sites, from things that are still effectively old-style collections of documents like Wikipedia, to sites that offer interactions with other people such as Facebook, YouTube, and Ebay, to things that can only be described as replacements for desktop applications such as Gmail and Goggle Docs. Many formerly stand-alone applications, such as mail clients, have become part and parcel of the web experience. In the modern Web, the line between applications and pages has blurred. The difference at this point is only in the intent of the site.

Running an application in the browser has some major advantages for both the user and the developer. For the user, there is no commitment to the application: you try it out and if you don't like it you can move on to the next page with nothing left behind to clutter up your disk. Trying new applications is also reasonably safe, in that they run in a sandboxed environment. New versions of the application are automatically down-loaded to the browser when the developer updates the code. Web applications rarely have version numbers, at least public ones.

For the developer, the case is even stronger. First of all, the things that are an advantage to the users are also good for the developers. There is no installation program to write, and new versions can automatically be sent to the users, making small incremental updates not only possible but practical. However there are other bonuses as well.

The web is cross-platform. It is possible to write a web page that will work on Windows XP, Windows Vista, Windows 7, Mac OS X, Linux, the iPhone/iPad, and Android. Doing that with a conventional development tool would be a monumental task. But with the Web and some forethought it almost comes for free. A web application built on standards with a library like jQuery ("jQuery" jQuery on page 0 ) will be able to run on major browsers on all those platforms and a few others. While at one point Sun hoped that its Java applets would define the web as a platform, JavaScript has turned out to become the default web platform.

You can even run web applications on mobile devices, at least the ones today called smartphones. With a wrapper like PhoneGap, you can create an HTML5 application and package it for sale in the App Store, the Android Market, and more. You might create an application that interacts heavily with a web server, or you might create a completely self-contained application. Both options are available.

The real place that the Web, prior to HTML5, traditionally falls short is that a web application, running on a computer with gigabytes of memory and disk space, acts almost like it is running on an old vt320 terminal. All data storage must be done on a server, all files must be loaded from the server, and every interaction pretty much re-quires a round trip to the server. This can cause the user experience to feel slow, espe-cially if the server is far away from the user. If every time the user wishes to look up something there is a minimum response time of 400 milliseconds before any actions

can be taken, the application will feel slow. From my office in Tel Aviv to a server in California, the round trip time for an ICMP ping is about 250 ms. Any action on the server would be extra and slow that down even more. Mobile device communications can, of course, be even slower.

# Adding Power to Web Applications

HTML5 raises the bar for web applications. While it still has to work under security constraints, it finally provides tools that desktop developers have expected for years:

*Local data storage*
> Up to 5 MB of data, referenced with a key-value system.

*Databases*
> Originally a SQLite-based API, the tide seems to have shifted to IndexedDB, a NoSQL system that is natively JavaScript.

*Files*
> While applications still can't freely access the file system, for obvious security reasons, they can now work with files the user specifies and are starting to be able to create files as well.

*Taking it offline*
> When a laptop or phone is in "Airplane Mode", web applications can't communicate with the server. Manifest files help developers get around that by caching files for later use.

*Web Workers*
> Threads and forks have always been problematic, but JavaScript simply didn't offer them. Web Workers provide a way to put applications processes into separate spaces where they can work without blocking other code.

*Web Sockets*
> Hypertext Transfer Protocol (HTTP) has been the foundation of the Web, despite a few updates over time. Web Sockets transform the request-response approach to create much more flexible communications systems.

There's much more, of course, from geolocation to audio and video to canvas graphics to a wide variety of minor new tags, but these provide the foundations for building industrial-strength applications on HTML5.

# The Power of JavaScript

JavaScript is not a difficult language to program, but it can be a challenge to rise to the level of a true JavaScript expert. There are several key factors to becoming a skilled JavaScript programmer. The techniques in this chapter will appear repeatedly in the libraries and programming practices taught in the rest of this book, so you should familiarize yourself with these techniques before continuing with those chapters.

There are a number of excellent tools for JavaScript programming, some of them listed in Appendix. These tools can provide you with a lot of assistance. Specifically, JSLint (JSLint on page 117) will catch a large number of errors that a programmer might miss. Sites such as StackOverflow and O'Reilly Answers will be a good source of other tools.

This chapter is not a full introduction to the power of JavaScript. There are a number of excelent books on Javscript including these.

- *JavaScript, The Good Parts* by Douglas Crockford
- *JavaScript: The Definitive Guide*, by David Flanagan
- *High Performance JavaScript*, by Nicholas C. Zakas
- *JavaScript Patterns*, by Stoyan Stefanov

## JavaScript's Triumph

Though JavaScript has been a key component of web development since it first appeared in 1995, it spent a decade or so with a bad reputation. It offered weak performance, quirky syntax that led to mysterious bugs, and suffered from its dependence on the DOM. Browsers kept it locked in a "sandbox", easing users' security concerns but making it very difficult for developers to provide features that seemed trivial in more traditional desktop application development.

Scripting culture created its own problems. While providing a very low barrier to entry is a good thing, it does come with costs. One of those costs is that such a language often allows inexperienced programmers to do some very ill-advised things. Beginning programmers could easily find JavaScript examples on the web, cut and paste it, change a

few things, and have something that mostly works. Unfortunately, maintaining such code becomes more and more difficult over time.

With the Ajax revival, developers took a new look at JavaScript. Some of those developers worked on improving the engines interpreting and running JavaScript code, leading to substantial speed improvements. Others focused on the language itself, realizing that it had some very nice features and were able to develop best practices like those outlined in Douglas Crockford's JavaScript: The Good Parts (O'Reilly, 2008).

Beyond the core language, developers built tools that made debugging JavaScript much easier. Although Venkman, an early debugger had appeared in 1998, the 2006 release of Firebug became the gold standard of javascript debuggers. It allows the developer to track Ajax calls, view the state of the DOM and CSS, as well as single stepping code and much more. Browsers built on WebKit, notably Apple's Safari and Google Chrome, offer similar functionality built in, and Opera Dragonfly provides support for Opera. Even developers working in the confined spaces of mobile devices can now get Firebug-like debugging with weinre, Web Inspector Remote.

The final key component in this massive recent investment in JavaScript was libraries. Developers still might not understand all the code they were using, but organizing that code into readily upgradeable and sometimes even interchangeable libraries simplified code management.

*jQuery*

>   If anything can be described as the Gold Standard of JavaScript libraries it would have to be John Resig's jQuery library which forms a wrapper around the DOM and other Javascript objects such as the XMLHttpRequest object and makes doing all sorts of things in JavaScript a lot easier and a lot more fun. In many ways jQuery is the essential JavaScript library that every JavaScript programmer should know.

>   To learn jQuery see the jQuery website http://jquery.org, or a number of good books on the subject such as "Head First jQuery" or the "jQuery Cookbook" both published by O'Reilly. Many examples in this book are written using jQuery.

*ExtJS*

>   Whereas jQuery forms a wrapper around the DOM, ExtJS, from Sencha http://sencha.com tries to abstract it away as much as possible. ExtJS features a rich widget set that can live in a web page and provide many of the widgets, such as trees, grids, forms, buttons and so on that desktop developers are familar with. The entire system is very well thought out and fits together well and makes developing many kinds of applications a joy. While ExtJS tends to be heavy in terms of the size of the library it is a good fit for some kinds of application development.

>   On nice feature of ExtJS is that many of its objects know how to save their state. So if a user takes a grid and reorganizes the columns the next time the user views that grid it is possible to set it up so that this state is saved. How to use the HTML5 localStorage to enhance this will be shown in

*Others*

Tools such as GWT allow the programmer to write Java code, which is then compiled down to JavaScript and then can be run on the browser.

# Non-blocking I/O and Callbacks

The first key to JavaScript, past the language itself, is to understand event-driven programming. In the environment where JavaScript runs, operations tend to be asynchronous, which is to say that they are set up in one place and will execute later after some external event happens.

This can represent a major change from the way I/O happens in traditional languages. Take Example 2-1 as a typical example of I/O in a traditional language, in this case PHP. The line `$db->getAll($query);` will require that the database access the disk, and therefore will take orders of mangitude more time to run than the rest of the function. While the program is waiting for the server to execute the query statement is blocked and the program is doing nothing. In a server side language like PHP, where there can be many parallel threads or processes of execution, this isn't usually a problem.

*Example 2-1. Blocking I/O in PHP*

```php
function getFromDatabase()
{
  $db = getDatabase();
  $query = "SELECT name FROM countries";
  $result = $db->getAll($query);
  return $result;
}
```

In JavaScript, however, there is only one thread of execution, so if the function is blocked nothing else happens and the user interface is frozen. As such, JavaScript must find a different way to handle I/O (including all network operations). What JavaScript does is return right away from a method that might be perceived as slow, leaving behind a function that gets called when the operation (say, downloading new data from the web server) is done. The function is known as a *callback*. When making an Ajax call to the server the Javascript launches the request then goes on to do something else. It then provides a function that is called when the server call is finished. This function is called (Hence the name callback) with the data that is returned from the server at the time when the data is ready.

As an analogy, consider two ways of buying an item at a grocery store. Some stores leave items behind the counter, so you have to ask a salesperson for it and wait while she retrieves it. That's like the PHP program just shown. On the other hand, some stores have a deli counter where you can request an order and get a number. You can go off

to do other shopping, and when your order is ready, you can pick it up. That situation is like a callback.

In general, a fast operation can be blocking, because it should return the data requested right away. A slow operation, such as a call to a server that may take several seconds, should be non-blocking and should return its data via a callback function. The presense of a callback option in a function will provide a good clue on the relative time it will take for an operation to run. In a single threaded language like JavaScript a function can not block to wait for the network or user without locking up the browser.

So a major step to JavaScript mastery involves using callbacks strategically and knowing when they'll be triggered. When you use a datastore object with Ajax, for example, the data will not be there for a second or two. Using a closure ("Closure" on page 10) to create a callback is the correct way to handle data loading. All such external I/O (databases, calls to the server) should be non-blocking in JavaScript, so learning to use closures and callbacks is critical.

With a few exceptions that should probably be avoided, JavaScript I/O does not block. The three major exceptions to this rule are the window methods `alert()`, `confirm()`, and `prompt()`. These three methods do in fact block all JavaScript on the page from the moment when they are called to the moment when the user dismisses the dialog. In addition the XHR object can make an Ajax call to the server in a syncrous mode, this can be used in a web worker but in the main window will cause the browser UI to lock up. It should be avoided.

# Lambda Functions Are Powerful

Programmers who have come to JavaScript from PHP or other procedural languages will tend to treat JavaScript functions like those in the languages that they have already used. While it is possible to use JavaScript functions in this way, it is missing a large chunk of what makes JavaScript functions so powerful.

JavaScript functions can be created with the function statement (Example 2-2) or the function expression (Example 2-3). These two forms look pretty similar, and both examples produce a function called `square` that will square a number. However, there are some key differences. The first form is subject to *hoisting*, which is to say that the function will be created at the start of the enclosing scope. So you can't use a function statement when you want the function defined conditionally, because JavaScript won't wait for the conditional statement to be executed before deciding whether to create the function. In practice, most browsers allow you to put a function inside an `if`, but it is not a good idea, as what browsers will do in this case can vary. It is much better to use a function statement if the definition of a function should be conditional.

*Example 2-2. Function declaration*

```
function square(x) {
    return x * x;
```

```
} // Note lack of a ;
```

*Example 2-3. Function statement*
```
var square = function(x) {
    return x * x;
};
```

In the second form, the function expression, the function is created when execution gets to that point in the flow of the program. It is possible to define a function conditionally, or to have the function defined inside a larger statement.

The function expression, in addition, assigns no name to the function, so the function can be left anonymous. However, the example shown assigns a name (`square`) on the left side of the equal sign, which is a good idea for two reasons. First, when you are debugging a program, assigning a name allows you to tell which function you're seeing in a stack trace; without it, the function will show up as `anonymous`. It can be quite frustrating to look at a stack trace in Firebug and see a stack of nine or ten functions, all of which are simply listed as `anonymous`. Also assigning a function name allows you to call the function recursively if desired.

A function expression can be used anywhere in JavaScript that an expression can appear. So a function can be assigned to a varible as in Example 2-3, but it can also be assigned to an object member or passed to a function.

JavaScript functions are more like the Lisp lambdas than C functions. In C-type languages (including Java and C++) a function is basically a static thing. It is not an object on which you can operate. While you can pass objects as arguments to functions, there is little ability to build composite objects or otherwise expand objects.

> Back in the 1950s when Lisp was first being created, the folks at MIT were being heavily influnced by Alonzo Church's Lambda Caculus, which provided a mathematical framework for dealing with functions and recursion. So John McCarthy used the keyword *lambda* for dealing with an anonymous function. This has propogated to other languages such as Perl, Python, and Ruby. Although the keyword "lambda" does not appear in JavaScript, its functions do the same things.

As in Lisp, functions in JavaScript are first class citizens of the language. A function in JavaScript is just data with a special property that it can be executed. But like all other variables in JavaScript, a function can be operated on. In C and similar languages, functions and data are in effect two separate spaces. In JavaScript, functions are data and can be used in every place that you can use data. A function can be assigned to a variable, passed as a parameter, or returned by a function. Passing a function to another

function is a very common operation in JavaScript. For example this would be used when creating a callback for a button click (Example 2-4). Also, a function can be changed by simple assignment.

*Example 2-4. ExtJS Button with function as handler*

```
var button = new Ext.Button({
    text: 'Save',
    handler: function() {
        // Do Save here
    }

});
```

# Closure

Access to functions as first class objects in JavaScript would not be worth as much, were it not for the property that goes along with it called *closure*. Closure is yet another element from Lisp that has migrated into JavaScript. When a function is created in JavaScript, the function has access to any lexically scoped variables that were in the environment that created it. Those variables are still available even if the context in which they were originally defined has finished executing. The variables may be accessed and modified by the inner function as well as the outer function.

Closures are often useful for constructing callbacks. When ever a second function will run as a response to some event but need to know what has happened before.

This is often useful when building a function generator, as each time the generator function runs it will have a different outer state, which will be encapsulated with the created function. It is also possible to create more than one function in a generator, all of which are closed onto the same environment.

In the following example the dom element or css selector 'el' is wrapped in a function to allow the HTML content to be set with a simple function call. The outer function (factory) binds the element 'el' to a lexical variable which is used by the inner function to set the dom element via jQuery. The outer function returns the inner function as its return value. By wrapping the entire function in parethesies in this example it is possible to call this function inline. In this example updateElement is set to the inner set function already with the element already bound.

Closure is one of the most powerful features of the JavaScript function. In the simple case it can be used to create functions that can access the variables of an outer scope to allow callbacks to access data from the controlling function. However even more powerful is the ability to create custom functions that bind variables into a scope.

*Example 2-5. Closure*

```
// closure example
//

var updateElement = (function factory (el)
    {
        return function set(html)
        {
            $(el).html(html);
        };
    }(el));
```

It is also possible to make several functions that are closed on one scope. If a function returns several functions in an object or array, all of those functions will have access to the internal variables of the creating function.

Example 2-6 adds to the browser's toolbar the buttons defined in the `tools` array. Each of the buttons gets its own handler, named `clickHandler`. This function has access to the callling function's variables, and embeds the `button` and `tool` variables into its operations. You can easily update the application by adding or subtracting an element from the `tools` array, and the button with all the defined functionality will appear or disappear.

*Example 2-6. Closure in a Button*

```
$('document').ready(function Ready() {
    var button, tools;
    tools = ['save', 'add', 'delete'];
    console.info($('div#toolbar'));
    tools.forEach(function (tool) {
        console.info(tool);
        var button = $('<button>').text(tool).attr({
            css: 'tool'
        }).appendTo('div#toolbar');
        button.click(function clickHandler() {
            console.info(tool, button);
            alert("User clicked " + tool);
        });
    });
});
```

When using closures, it can be hard to know which variables are or are not in the scope of a function. However, both Google Chrome's Dev tools and Firebug will show the list of closed variables.

In Firebug the scope chain can be seen in the Script tab by looking under "Watch". Under all the variables of the current scope will be a ladder of the scopes going up to the main 'window' object.
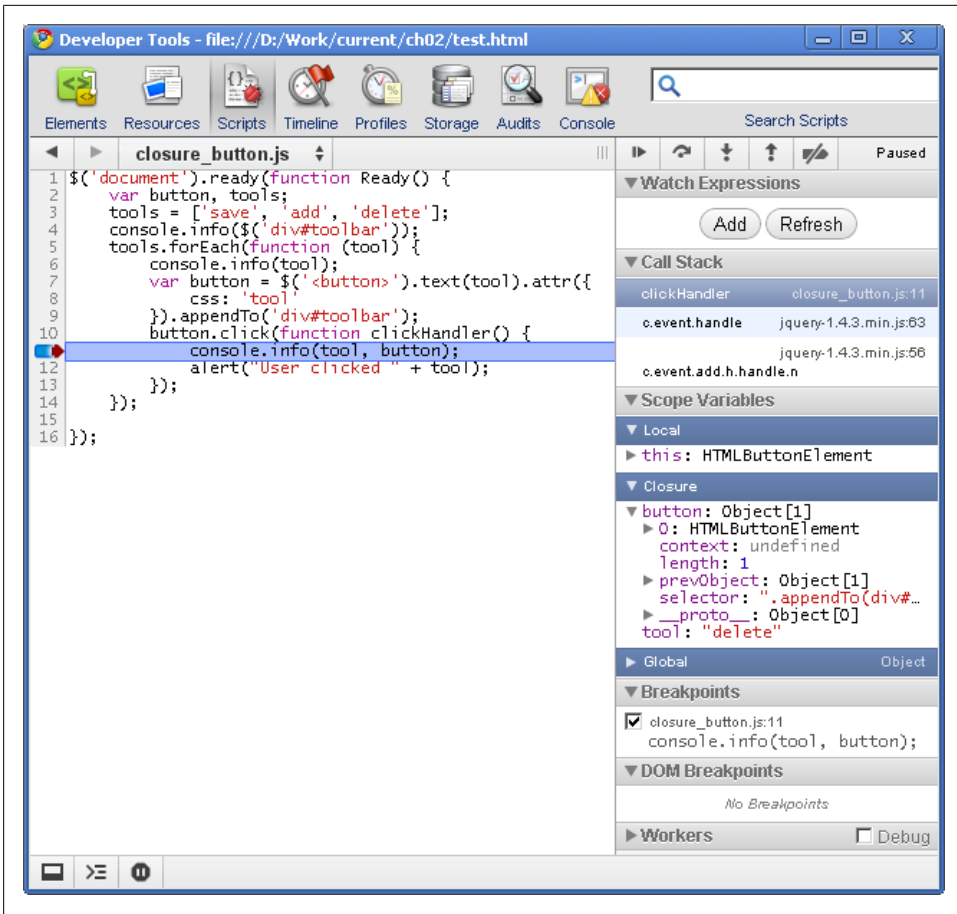
*Figure 2-1. Closures in Google Chrome's Dev tools*

In Dev tools, for example, when the code is halted in the debugger, a subsection called "closure" in the right hand column under Scope Variables will show the closed variables for the current function (Figure 2-1). In this case, it shows that we have clicked on the "delete" button and lists the reference to the jQuery object for the button itself.

# Functional Programming

Functional programming is a methodology that is more commonly associated with languages like Lisp, Scala, Erlang, F# or Haskall, but works quite well in JavaScript also. Functional programming rests on a couple basic assumptions:

- Functions are first class citizens of the language an can be used where any other value can be used.

- Complex behavior can be built by composition of simple functions.
- Functions return values, in many cases a given function will always return the same value for the same inputs.

In mathematics, functions do not have side effects. Take a classical mathematical function like `y = sin(x)`. It just returns a value that `y` can store, but does not change `x` or anything in the global state of the program. By ensuring that functions are "pure" (have no side effects), this practice enables the function to be called from anywhere in the program without causing something strange to happen. The problem with side effects in programming is that they can cause strange dependencies that can be very hard to track down. If calling a method can cause data to be corrupted somewhere else, it greatly increases the potential for bugs that will be very difficult to track down.

JavaScript functions can have side effects and there is no built in way to prevent functions from having side effects. In addition Javascript functions do not by default return values unless the `return` statement has been explicitly invoked to return a value. In the absanse of a `return` statement the function will return `undefined`.

When employing functional programing, the `programmer often falls into the pattern of using many very small functions, often with only two or three lines of code each, to accomplish a goal. This can be a very good design technique, as very short functions are in general easier to make corrrect, as well as being easier to test.

When doing functional programming in Javascript it is often useful to have a function return the implicit `this` variable to allow composition chains.

It is often the case that complex behavior can be built up from simple functions by composition. A chain of functions can be built up from simple functions so that each function in the chain returns `this` allowing the next function to be called.

The jQuery library often uses function changes such as in . In this example jQuery finds a DOM item, sets its text, fades it into view and then sets a click handler on it that will hide it with a second dom chain.

```
 $('div.alert').text("Message").fadein(2000).click(
function ()
{
   $(this).fadeout(2000);
}
);
```

One very powerful pattern of Functional programming is the higher order function. A higher order function takes a function as an argument to abstract out specific behavior while leaving the generic behavior in the outer function.

A good example of a higher order function is the Array map function. "Array iteration operations" on page 21 Which takes an array and returns a new array that is the result of the passed function applied to each element in the array. This model can be applied

to a wide range of circumstances beyond just array manipulation. As a general patern the Higher order function can be used whereever a geenric behavior needs a few specific modifications.

The jQuery library tends to favor functional programing in its interface. The jQuery interface tends to promote selecting a set of DOM nodes from the page and then providing a functional interface to interact with those nodes.

In addition most of the methods of jQuery return a value so that they can be chained. For example to find all the images wider then a set size in a page one could select all the images in the page, filter out all those smaller than 300 pixels then scale all those that are left in the list.

Example 2-7 does exactly that. It selects all the images in a document, then filters out those that have a width greater than 300 pixels (maxWidth), and scales those. By making the filter and scale functions very simple it is possible to be very confidant that the code will work as intended.

*Example 2-7. Scaling Images*

```
var scaleImages = (function (maxWidth){
  return function (){
    $('img').filter(function (){
       return  $(this).width() > maxWidth;
    }).each(function (){
       $(this).width(maxWidth);
    });


};
}(300));
```

When processing a list of operations in a time consuming process, such as in an Ajax call. Sometimes it is not practical to send the entire list to the server in one request. For example it may be that sending the entire list will cause the server to time out.

In this case to itterate over the list it is useful to think of the list as a head and tail. Take the first element off of the list (or first few elements) and process that, then by use of recurssion process the rest of the list until the list is empty.

At each time threw the loop the userinterface can be updated to show the progress to the user so that he knows what is going on.

For example I have used this when adding data to a REST interface. Each call to the interface takes on average about one second so it is not practical to call it 500 times from one Ajax call. In this case I was able to process the list by recursion.

What's with the entire Ajax versus XHR terminology? The JavaScript object is called `XMLHttpRequest`, which is abbreviated XHR. Ajax derives from the term "Asynchronous JavaScript and XML," which was coined by Jesse James Garret. Actually, in many cases the data being sent over the network is not XML but may be JSON or other data.

*Example 2-8. List Recursion*

```
/* List recurssion example */

function itterateAjax(tasks) {
    function itterator(tasks) {
        $.ajax({
            url: 'index.php',
            data: tasks[0],
            success: function(data, status, XmlHttpRequest) {
                if (tasks.length > 0) {
                    // Do something with the results here
                    itterator(tasks.slice(1));
                }
            }
        });
    }
    itterator(tasks);
}
```

Although building an entire single page web application with only functional programming styles would not be practical, there are many useful ideas here which should not be ignored. Functional programming is very well suited for use with Web Workers (Chapter 8).

Not a lot has been written about functional programming in JavaScript, but there is quite a lot in other langauges that can be applied to JavaScript. For more on functional programming, see these books:

- *Real World Haskell*, by Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart (O'Reilly)
- *Programming Scala*, by Dean Wampler, and Alex Payne (O'Reilly)
- *Structure and Interpretation of Computer Programs*, by Harold Abelson and Gerald Jay Sussman (MIT Press)

# Prototypes and How to Expand Objects

Everything in JavaScript can have methods attached to it. Every element has some basic methods that can be used by the programmer to enhance its usefulness. JavaScript primitives such as Booleans, strings, and numbers have a second life as objects. The

transformation from primitive to object is transparent, so it is possible to apply these methods to a primitive. Actally what happens is that a simple value, for example a string, will be converted to an object, and then converted back if needed.

For strings, a large number of methods that can be called to manipulate them. A few will modify the string in place, but most will return a new string. A full list can be found on the Mozilla developers site, but here are a few highlights.

`string.indexOf('x')`
> Returns the first index of a substring in a string, or -1 if not found.

`string.lastIndexOf('x')`
> Same as `indexOf()`, but starting at the end.

`string.match(/regexp/)`
> Matches a regular expression in a string.

`string.replace(/regexp/)`
> Replaces a regexp (specified as a function or string).

`string.split()`
> Splits up a string into an array of substrings.

`string.slice()`
> Extracts a substring.

However, there may come a time when the predefined methods are not enough and some custom functionality is required. In this case, JavaScript provides something very powerful, a way to extend a built-in object. Although you can always assign a method to a JavaScript object with a simple assignment, that is not always the best way to do it. If you want to add a method to every string, you can attach a method to the `String.prototype` object. In the JavaScript object system, each object inherits from a chain of prototypes, so by adding methods somewhere in that chain you can add to an entire type of object.

Here is an example to illustrate the concept. The goal is to create a new method named `populate` that substitutes values into a template. The template is the object on which the method is called, for instance:

```
Hello {name}
```

The string should contain keywords in curly braces that the programmer wants to replace with specific values. The parameter to `populate` is an object specifying keywords in the template and values to substitute. Thus, if the argument contains a property called `name`, the value for `name` is plugged into the string.

Once the following code is run, the `populate` method is attached to all strings. When `populate` is called, it refers to the string on which it is called through the standard JavaScript object `this`. Having the value of `this`, the `populate` method can use simple substitution to plug in values from its parameter. In general, it is a good idea to not

modify the object on which a method is called, but to return a new instance of the object (an idea from functional programming).

*Example 2-9. String Token Replacement*

```
String.prototype.populate = function populate(params) {
    var str = this.replace(/\{\w+\}/g, function stringFormatInner(word) {
        return params[word.substr(1, word.length - 2)];
    });
    return str;
};

"Hello {name}".populate({
    name: "Zach"
});
```

Of course, Strings are not the only types of objects in JavaScript that have prototypes, Numbers, Arrays, Objects, Functions, and Booleans do as well.
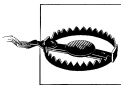
> Extending the prototypes of basic objects such as Object, Array, etc., can sometimes break libraries. The culprit is usually creating a property that already existed on the object. Be sure that the property you are creating does not exist before adding it, and test carefully.
>
> In truth extending base types in Javascript is a practice that attracts a lot of debate. Some people say it should never be done. I think that it is such a powerful tool that it can not be ruled out completely.

The Number prototype works exactly the same way as the one for strings. So it is very possible to define a new method to take care of any need that may be required. If, for example, an application requires squaring numbers on a regular basis, it would be easy to add this method:

*Example 2-10. Using Number.prototype*

```
Number.prototype.square = function square() {
    return this * this;
};
6.square(); // 36
```

# Expanding Functions with Prototypes

In addition to data objects such as strings and arrays, functions also have a prototype. This can be used to create composite functions that are very powerful. By combining simple functions into larger units, using the `Function.prototype` to add methods to the Function object, you can pull apart complex logic into much simpler cases. In fact,

many toolkits do exactly this and provide some methods to accomplish some of these tasks.

One example of prototyping, which may increase the robustness of your code throughout its execution, is to add error-checking before executing a function. In the following example, a cube function runs without checking that its input is a number. The code wraps the function in an interceptor that does the check. Whenever cube is subsequently called, the interceptor runs first, then calls the original cube if the input is a number.

*Example 2-11. Function interceptor*

```
Function.prototype.createInterceptor = function createInterceptor(fn) {
    var scope = {};
    return function() {
        if (fn.apply(scope, arguments)) {
            return this.apply(scope, arguments);
        }
        else {
            return null;
        }
    };
};

var interceptMe = function cube(x) {
    console.info(x);
    return Math.pow(x, 3);
};

//
var cube = interceptMe.createInterceptor(function(x) {
    return typeof x === "number";
});
```

As a more extensive example, say that you wish to compute Fibonacci sequences regularly. A very simple brute force approach would be something like Example 2-12. However something as trivial as fib(40) would take quite a long time to run.

*Example 2-12. Basic Fibonacci*

```
var fib = function fib(n) {
    if (n === 1 || n === 2) {
        return 1;
    }
    return (fib(n - 1) + fib(n - 2));
};
```

A quick trace through a sample run of this function shows that it does a lot of redundant calculations. It would be much faster to have a Fibonacci method that could calculate each value only once. We can do this by wrapping the Fibonacci function with an

interceptor method that caches the result of each iteration (Example 2-13). The interceptor does not need to know anything about how a Fibonacci sequence is generated; it just has to know that for a given input it should always produce the same output. So once fib(*n*) is computed, it becomes a simple matter of doing a lookup, and if it is not known, it can be computed.

Example 2-13 has two parts: the cache method and the actual Fibonacci sequence generator. The cache method does not know anything about the Fibonacci sequence except for one fact: that any given input value will always return the same value, which can be cached. Thus, when decoratedFib(32) is called, the first thing that will happen the cache will check whether it has already computed the result for 32. If it has, it will just return it. If it has not, it will start to compute it. But the Fibbonacci sequence is heavily recursive. So to compute the Fibbonacci sequence for 32, the function first must compute it for 31, and so on. The function will recursively look for a solution until it hits a value it has. If this is the first time running the function, it will find n = 2 and n = 1, which are in the seed values.

Although the Fibonacci sequence is not something many people spend much time on it is a good example of how using a function prototype can take two very short functions and combine them for a very powerful result.

> These examples more complex then it realy has to be in order to show the idea of how to cache function results from functions that have no side effects. Really this is probably not the best way to write this code.

*Example 2-13. Advanced Fibonacci*

```
var smartFib = (function makeFib() {
    var fibsequence = [0, 1, 1];
    var fib = function fib(n) {
        if (fibsequence[n]) {
            return fibsequence[n];
        }
        fibN = fib(n - 1) + fib(n - 2);
        fibsequence[n] = fibN;
        return fibsequence[n];
    };
    return fib;
}());

Function.prototype.decorate = function Decorate(params) {
    return params.decorator(this, params.initialData);
};

var cache = function cache(lambda, initial) {
    return function cacheRunner(n) {
        if (initial[n] !== undefined) {
            return initial[n];
        }
```

```
        else {
            initial[n] = lambda(n);
            return initial[n];
        }
    };
};

var decoratedFib = function fib(n) {
    return decoratedFib(n - 1) + decoratedFib(n - 2);
}.decorate({
    decorator: cache,
    initialData: [0, 1, 1]
});
```

Suppose that you have a function that should run periodically in response to user inputs but should not run more than once in a given time period. It would be simple using the Function prototype to create a wrapper function that would store when that function was last called and keep it from running if it is called again within the designated time. You could choose just to not run it quietly, or throw an exception, as needed by the application.

On the flip side, it is also possible to create a method to a function that will cause it execute after a delay, or periodically. Often, a task needs to run after an event but should not run too often. For instance, you might want to check on what a user is typing, but running the check after every keystroke would be excessive. Setting up a method that will run only once every 250 ms would make sense.

There are two approaches to this. First, you could run the method once and not allow it to run again until the required time has passed. Or you could create a method that will run at some time interval after it is called, and will reset the timer when called. This pattern is useful if the objective is have something run when the user stops typing or when some other set of events pauses. In practice, a new method would act as a wrapper around the basic JavaScript `setTimeout()` and `setInterval()` methods, but make their use more convenient. It is also possible to create a method that will schedule a future task and cancel an existing task if it has already been scheduled.

# Currying and Object Parameters

In functional programming, one common programming model is "currying" a function. Named for a feature of Haskell, currying refers to the practice of combining several parameters into a single object so you can pass them to a function as a single parameter. If a function needs to take a large number of parameters, it is often best in JavaScript to forgo the long parameter list and take a single object as a parameter. By using an object as the parameter, you can turn all the various options into name/value pairs. One benefit of doing so is that the ordering of arguments becomes irrelevant. Furthermore, you can make some or all of the parameters optional. For complex methods

---

accepting many options, this can be a great help. In particular, it is often useful for some of the object creation methods in ExtJS.

The easiest way to curry parameters is to create a function that takes a parameter block and returns a function that will call the original function with the pre-supplied parameters as defaults. This way, you can establish a set of defaults that you don't have to specify each time, while allowing callers to change any parameters they want.

*Example 2-14. Curry Example*

```
Function.prototype.curry = function FunctionCurry(defaults) {
    var fn = this;
    return function(params) {
        return fn.apply(this, defaults.concat(params));
    };

};
```

This pattern can also be applied to object creation. For an object constructor that takes a parameter block, it is possible to subclass the object with a custom class that calls the parent constructor with a set of default parameters overlaid onto what the user passed.

# Array iteration operations

Like other first class objects in JavaScript, Arrays also have methods. The standard array has a number of methods for the programmer. In the more recent versions of Firefox (post 1.5), a number of standard iteration methods have also been created.

The basic idea of these operations is that you take a lambda function and apply it to each element of an array to produce some result. Using this approach with a few smaller functions allows you take an array and create a set of operations on it that can build an algebra of arrays. This in turn lets you build up a very robust set of operations from a basic set of operations.

The first of the array methods is `map()`. Map takes an array and a method. It then applies the method to each element of the array and creates a new array from the return values. So given an array of numbers, it would be possible to create an array of squares of each of the numbers by simply applying a square function in `map()`.
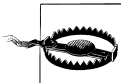
The array methods such as map are avalable on most modern browsers. However If they are not avalable they can be added. Sample code for all of the array methods can be found on the mozilla developer site.

As of Internet Explorer version 9 these methods exist in all major browsers

The called function receives as parameters the current value of the array, the index of the current position in the array, and the entire array. Usually, the map worker function will only need to look at the current value of the array, as in Example 2-15.

*Example 2-15. Array Map*

```
[1, 2, 3, 4 , 5].map(function (x){
return x* x;
});
//Result:  [1,4,9,16,25]
```

However, there are a few cases where that may not be enough. In Example 2-16, the desire is to show a running average of the values in an array, so each element needs to know about its neighbors.

*Example 2-16. Running Average Example*

```
function makeRunningAverage(list, size){
    return list.map(function (current, index, list){
            var start, end, win;
        /* find start and end points of the rolling average window */
            start = index - size < 0? 0: index-size;
            end  = index + size > list.length? list.length : index+ size;
        /* extract that window */
            win = list.slice(start,end);
        /* take an average */
            return win.reduce(function (accumulator, current){return accumulator + current;}, 0)
            / (end - start);

        });

}
```

There are a few big advantages to using array elements over a `for` loop. First of all, it is logically clean and isolates the iteration code, allowing the programmer to think of the array as a whole. Secondly, by avoiding side effects, in the inner function and keeping the functions short, it is possible to write very robust code.

Another case that should be considered is when adding callback handlers. If you were to use a `for` loop to iterate over items and addn handlers, you cannot use the closure property of JavaScript functions. You might be tempted to use something like Example 2-17. However this will not do what you expect. In this case, the method will always show the last element. What is deceptive is that here, the value of `i` that is referenced in all cases is the final value. A closure will always see the current value of a variable, not the value when it was created. The `for` loop changes the value of `i` as it itterates over the list. In the Example 2-18 example, the code will work correctly. In this case

the node referenced in each iteration is independent because it is inside the scope of the function.

*Example 2-17. Loop With For*

```
for (var i = 0; i < nodes.length; i += 1) {
    nodes[i].bind('click', function() {
        console.info(nodes[i]);
    });
}
```

*Example 2-18. Binding with forEach*

```
nodes.forEach(function(node) {
    node.bind('click', function() {
        console.info(node);
    });
});
```

The next method of note is `filter()`. Filter takes an array and returns the subset of the array for which the method returns true. The iterator function here receives the same parameters as it does for `map` but should return a boolean value

*Example 2-19. Filter*

```
[1,2,3,4,5].filter(even); => [2,4]
```

If you wish to know whether some fact is true for all elements of an array, use `every()`. It will apply a method to an array and return true if the method returns true for all elements in the array. It will stop after the first false.

If you wish to know whether some condition is true for *at least one* element of the list, use the `some()` method. This will return true if at least one element of a list returns true. Like `every()`, it will evaluate only enough elements to get a result.

The last two operators in the JavaScript algebra of arrays are `reduce()` and `reduce Right()`. The first method takes an array and reduces it to some single value. It is useful as an accumulator. In this case, the calling function also receives the accumulated value. So using `reduce()` to sum up a list would look like the following listing. You can provide an optional initial value, which will be passed as the previous value on the first iteration. If you do not, it will start by using the first two elements of the array.

*Example 2-20. Reduce*

```
[0,1,2,3,4,5].reduce(function(prev, current){
                return prev + current;
```

```
                },
                initialValue);
```

If the JavaScript array algebra does not provide the method you need for a task, use the
`Array.prototype` object to create it. If you have a list of numbers and you need to create
a standard deviation from that list, for instance, you can simply apply a standard de‐
viation method. In Example 2-21 a Standard Deviation method is added to the array
prototype.

*Example 2-21. Standard Deviation*

```
    var stdDev = [1,2,7,2....].stddev();


Array.reduce.sum = function sum() {
    var sum = this.reduce(function(previous, current) {
        return previous + current;
    });
    return sum;
};

Array.prototype.square = function squareArray() {
    return this.map(function(x) {
        return x * x;
    });
};

Array.prototype.mean = function mean() {
    return this.sum() / this.length;
};

Array.prototype.standardDeviation = function standardDeviation() {
    var mean = this.mean();
    var int1 = this.map(function(n) {
        return n - mean;
    });
    var int2 = int1.square();
    var int3 = Math.sqrt(int2.sum() / mean.length);
};

//Give it a shorter name
Array.prototype.stddev = Array.prototype.standardDeviation;
```

# You Can Extend Objects, Too

If you like the map function in arrays and wish you had that for objects as well, there
is no reason why you could not add it. It is possible to build a map function that will
not only visit all the nodes of a JavaScript object but will also recursively apply a function

---

to each of its subnodes. This can be very useful for turning a data structure into some form of a node tree. The following example checks to see whether the user's browser has already defined `map()` and `filter()` for the object being assigned to, then defines the functions if necessary.

*Example 2-22. Extending map and filter to Objects*

```
if (Object.prototype.map === undefined) {
    Object.prototype.map = function(fn) {
        var newObj = {};
        for (var i in this) {
            if (this.hasOwnProperty(i)) {
                newObj[i] = fn(i, this[i], this);
            }
        }
        return newObj;
    };
}

if (Object.prototype.filter === undefined) {
    Object.prototype.filter = function(fn) {
        var newObj = {};
        for (var i in this) {
            if (this.hasOwnProperty(i)) {
                if (fn(i, this[i], this)) {
                    newObj[i] = this[i];
                }
            }
        }
        return newObj;
    };
}
```

JavaScript Objects can become arbitrarily complex trees in modern applications. It would be nice to be able to find a specific subtree of an object, by a path similar to the way one might specify a path in a file system. This is actually rather easy to accomplish.

The path method takes a path in the form of a Unix file path, */path/to/our/data*. It then uses an inner function to recursively move down the data tree until it finds the requested element and returns it, or realizes the element is not present, in which case it will return undefined. At first glance, it would seem to make sense to just call the top-level path function for each iteration. This would not be a good idea, as it is possible that in some cases there will be an indexed array at some point in the path, causing the iteration to break if the `Array.prototype` is not the same as the `Object.prototype`. By doing the search with an inner function, you avoid this problem.

This method can handle arrays as well as objects. Since both Arrays and objects can be addressed with the square bracket notation if a part of the path is a number, ie */4/* then it will take the 4th element in the array.

*Example 2-23. Select By Path*

```
Object.prototype.path = function FindByPath(path) {
    var elementPath = path.split('/');
    var findItter = function findItter(element, path) {
        // If the element is empty just ignore it and move on
        if (path[0] === '') {
            return findItter(element, path.slice(1));
        }
        if (element[path[0]] === undefined) {
            return undefined;
        }
        if (path.length === 1) {
            return element[path[0]];
        }
        return findItter(element[path[0]], path.slice(1));
    };
    return findItter(this, elementPath);
};
```

# Testing Javascript applications

Testing has become a key element of development in most server-side development platforms. Solid test harnesses can be found in development environments for PHP, Java, Ruby and so on. However the standard method for testing in most of these languages does not work well for Javascript. Let's look at a few reasons why.

Server-side test suites generally have to test the program under just one set of environments. If a REST service is being built with Python, the tester can build the tests with several safe assumptions. For instance, they may know that it will run on Python version 3.0 on Linux, along with specific versions of all the supporting software.

The web application developer has no such confidence. Users will come to the site using Firefox, Internet Explorer, Chrome, Safari, and Opera—and several versions of each. So test suites must be able to handle testing across a number of browsers and operating systems, each of which is a little bit different.

There are two main sources of differences, there are differences in the language itself between different browsers. For example Firefox supports the keyword `const` while Internet Explorer does not. Secondly many HTML interfaces only exist in some browsers or browser versions. Many of the various JavaScript interfaces in this book only exist in some browsers. So tests must be able to adjust to those differences and handle degredation where needed.

The run-time model for JavaScript is also more complex. In a server-side language, a unit test generally consists of a few sequential steps:

1. Set up any needed fixtures.
2. Run the method to be tested.
3. Test the results of the method against some criteria.

However, this model does not fit Javascript, where the results of an action may not be immediate but may happen some time later. A test like this would fail in Javascript as there can be delay between when the method is run and when its results will become avalible. In Javascript, testing a test may go more like this:

1. Set up any needed fixtures.
2. Run the method to be tested.
3. Wait for an Ajax call to complete.
4. Check the DOM for the results of the action (including undesired effects on un-expected parts of the page).

In addition, there is the complication that in many cases the method to run is a callback on a DOM element. In order to run unit tests in this case, you really want to find the DOM Element and send it the correct event. That way, the handler is invoked in a manner as similar as possible to when an actual user is using the application. Even that is not enough: it doesn't reproduce the experience of a user clicking through various complex interfaces in the DOM.

# QUnit

QUnit is a Javascript test suite from the same team that produced jQuery. To create a test suite in QUnit, call the test function with two arguments: the test suite name and a function that will actually run the tests.

> It is also possible to configure QUnit to complain if any new global variables have been introduced by running a test, and because global variables are one of the major sources of Javascript bugs, this is a very helpful feature. To test for leaking global variables, add `?noglobals` to the URL when starting the tests.
>
> Use of JSLint JSLint on page 117 can also be very helpful in finding leaking globals and a lot of other errors. It is possible to run JSLint over a set of javascript files as part of an automated test suite.

## A Simple Example

As an example let us consider a very trivial Ajax application. In Example 3-1 We have a HTML page with a button. When the button is clicked, the JavaScript `handleButton Click()` function will make an Ajax query to the server for a document (in this case, a static HTML document) and then place that document in a <div> in the page. Note that for simplicity I have placed the JavaScript in the document directly. A cleaner implementation would keep the JavaScript in a separate file that the test runner loads along with the tests.

To test this we want to run the button callback and verify the results are correct. There are two ways that this can be done. The test program could send a click event to the button, or it could call the function directly. For testing in QUnit it is simpler to call the function directly, but that requires us to bind the function to some variable that can be seen by the test runner. I have done this in the example, assigning the function

to the variable `handleButtonClick`. after calling the callback the test function waits for a short time to allow the ajax call to run then uses JQuery to check if the element is present.

*Example 3-1. Simple Application*

```
          <!DOCTYPE html>
<html>
<head>
  <script src="http://code.jquery.com/jquery-latest.js"></script>
  <script src='button.js'>
  </script>
  <script>
var handleButtonClick = function handleButtonClick(){
  $().get('document.html', '', function(data, status){
        $("<div>").attr({id:'target_div'}).text(data).appendTo('body');
      });
};

$('button.click_me').click(handleButtonClick);
 </script>
  <link rel="stylesheet"
        href="http://github.com/jquery/qunit/raw/master/qunit/qunit.css"
        type="text/css"
      media="screen" />
  <script    src="http://github.com/jquery/qunit/raw/master/qunit/qunit.js"></script>
  <script src='simple-test.js'></script>
</head>

<body>
 <button id='click_me'>Click Me</button>
 <!-- Stuff required by QUnit-->
 <h1 id="qunit-header">QUnit example</h1>
 <h2 id="qunit-banner"></h2>
 <h2 id="qunit-userAgent"></h2>
 <ol id="qunit-tests"></ol>
 <div id="qunit-fixture">test markup, will be hidden</div>
</body>
</html>
```

The QUnit test runner is loaded by the page on load as any other javascript program would be. In this case it is loaded from *http://github.com/jquery/qunit/raw/master/qunit/qunit.js*. This file will start running any jquery tests on load. The results of those tests will be shown on the page once tests are finished running, which is why QUnit requires those elements to be present in the DOM.

To test the example, we'll load QUnit which the test runner in Example 3-1 does, then calls the `handleButtonClick` method. Example 3-2 waits for one second for the document to be loaded, by passing a value of 1000 to the `setTimeout` method. After that

second, it tests whether the <div> exists in the DOM (the first `equal` call), gets the text from the div, and checks that the first word in the text is "First", which is the expected value (the second `equal` call). A more complete test may choose to check for that element every quarter-second until it appears or until some maximum time is reached. In a real world web page load times can vary depending on external factors including network usage and server load.

A test in QUnit is a javascript function that is called by the test runner. Looking at the simple test in Example 3-2. The test uses several assertion functions that must be satisfied for the test to pass. To test if a value is equal to an expected result use the `equal()` method, it takes three methods, the value to test, the expected result and an optional parameter which is a message for the test to show if it fails. Using this message will help figure out what when a test fails. This is even more useful if a test is being looked at six months after it was written.

*Example 3-2. Simple Test*

```
test("Basic Test", function (){
    // assert that the target attribute does not exist
    equal( $('div#target_div').length, 0,
            "Target element should not exist");
    //run method
    handleButtonClick();
    equal( $('div#target_div').length, 0,
            "Target element should still not exist");
    window.setTimeout(function (){
                start();
                equal($('div#target_div').length,1,
                        "Target element should now exist");
                equal("First",
                        $('div#target_div').text().substr(0,5),
                        "Check that the first word is correct");
            }, 1000  );
    stop();
});
```

## Testing with QUnit

To run QUnit tests, the qunit style sheet and javascript files must be included, these can be pulled directly from github or loaded locally. (See Example 3-1). The DOM must also include a few elements which are used by QUnit to display its results. These can be seen in the bottom of the HTML in Example 3-1. That is all that is required to run tests.

QUnit provides eight assertion functions. In addition to `equal()`, which appeared in our previous example, there are further tests of equality and the `ok()` method, which

tests whether the value passed to it is true. Also, `strictEqual()` tests according to the javascript "===" operator, whereas `equal()` uses the "==" operation for comparison.

To test whether a more complex data structure is the same, use `deepEqual()`. This does a recursive comparison of the two data structures.

Each of the equality functions has an inverted form which will test for lack of equality: `notEqual()`, `notStrictEqual()` and `notDeapEqual()`. These take the same paramaters as the equal versions but test for the inverted cases.

The final assertion is `raises()` which takes a function as a paramater and expects it to throw an Error conditon.

To test for events that happen in an asynchronous manor using a return value won't work. In this case the test must wait for the action to complete. This can be done by a setting a timeout with `setTimeout()`, which will run after a set time. Or it can be done with a callback from some operation such as a ajax load or other event.

## Mocking and Stubs

In some cases it will be appropriate for a unit test to run the code the way it will be run by users in the production system. At other times, that would cause problems or be too complex. In these cases it is possible to replace the complex object with a simpler object that can serve to isolate the object under test.

Using mock objects is a way to isolate a compont of the system so that that one component can be tested without the complexity of the entire system. If each component of a system can be tested in isolation, tests of the entire system will be greatly reduced when the parts are put together the number, because the developer knows from the simple tests that each component does the correct thing.

For example, if a button is supposed to call a method to cause an object to load some data when it is clicked, a mocking test could replace the data object with a test that clicks the button (or calls the click handler) and replaces the store with a simple class that has an assertion that passes when the correct method is called. This could also be done with an event, if the class defines a useful event. There is no need in this case to actually load the data from the server, as we know from a successful mock test that the button did what it should have done: call the load method. Assuming the data store load has been tested elsewhere, there is no need to retest it, which would just make the test suite run slower.

# Selenium

While QUnit lets you test Javascript code, Selenium takes a different approach. Selenium tests the user interface by simulating the actions that a user may take. A Selenium test will consist of a number of steps that run in a browser such as loading a page,

clicking on a particular element, typing something into a text area, and so on. Interspersed with those actions will be assertions to verify the state of the DOM or other things to be tested . These can include testing that an element or text exists or is absent.

When a Selenium test is run it will actually launch a browser and run it in more or less the same way that user would. So it is possible to watch the test interacting with the browser. It is even possible to manually interact with the browser while a test is running (though this may not be a great idea).

Selenium consists of several, mostly independent parts. One is an IDE implemented as a browser plugin for Firefox. Another is the Selenium RC server, *seleniumrc*, a Java server that can be used to automate running tests in different browsers.

The plugin Selenium IDE for Firefox is a developer's best friend. It allows you to construct tests that are run directly in the browser. The IDE can record the user's actions and replay them as a test later. It also allows you to step throught a test one line at a time, which can be very useful for finding timing problems in a test.

> By default when recording actions the Selenium IDE will use the ID of the various HTML elements. In the case where IDs are not explicity assigned by the programmer some frameworks will assign IDs sequentally as elements are created. These IDs will not be consistant from run to run, so use some other method to identify elements of interest.

The Selenium IDE outputs tests as HTML files that can be run in the IDE itself in Firefox. In addition, these HTML files can be run with the Selenium RC component as a batch job. The Selenium RC server also will allow the HTML tests to be run with any browser, so it is possible to run these tests with IE, Chrome, Opera, or Safari. The Selenium RC server can also be controlled by a traditional test from a test running like phpUnit or jUnit.

The Selenium IDE also is useful for recording web macros in development. For example, If you are testing a wizard in your web application that displays four or five screens before the screen that is being debugged, you can use the IDE to create a Selenium script that you can then invoke as a macro to get you automatically to the point being tested.

> If a test works when it is run in single step mode but not normally it probably needs a few pause statements to allow the browser to catch up, or even better some `waitFor...` statements, which will allow the test and the browser to be in sync.

There are three ways to run tests in Selenium, via the Selenium IDE, from the Selenium RC test runner, and from a programming language. The IDE is easy to use in an interactive setting, but works only with Firefox. The test runner accepts input tests in HTML format, which can be created in the IDE. Finally it is possible to write tests in a unit

test framework in a programming language such as phpUnit. Using the test runner or the programming language based test suite allows testing with a full suite of browsers and can provide reporting and the other functions. This procedure can also be integrated with continous integration tools along with any other tests written in any of the xUnit frameworks.

A Selenium test is constructed from an HTML file containing a table, each step in the test being a row in the table. The row consists of three columns: the command to run, the element on which that will act, and an optional paramater used in some cases. For example, the third column contains the text to type into an input element while testing a form.

Unlike a qUint test, a Selenium test is all about the user interface. Thus Selenium is more about integration testing than unit testing. To test Example 3-1 in Selenium, a different approach is required from QUnit. While the QUnit test called the handler function directly, the Selenium test clicks the button and waits for the <div> to show.

This is done in Example 3-3. Each row in the table document performs an action as part of the test. The first row opens the web page to test, then the second line clicks the button (which is identified by the element ID). The test then waits for the page to display the <div>, identified in this case through xpath.

This test tests the same simple script as the qUnit test above. However instead of testing the function itself it tests the User Interface in a way similar to a human tester may do so. It opens the page, clicks on the `click_me` button. Then waits for the `target_div` div to be present int he dom. It then asserts that the word "First" appears in the page.

*Example 3-3. Simple Selenium Test*

```
          <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head profile="http://selenium-ide.openqa.org/profiles/test-case">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<link rel="selenium.base" href="http://www.example.com/" />
<title>New Test</title>
</head>
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">New Test</td></tr>
</thead><tbody>
<tr>
    <td>open</td>
    <td>/examples/simple.html</td>
    <td></td>
</tr>
<tr>
    <td>click</td>
    <td>click_me</td>
    <td></td>
```

```
</tr>
<tr>
    <td>waitForElementPresent</td>
    <td>//div[@id='target_div']</td>
    <td></td>
</tr>
<tr>
    <td>assertTextPresent</td>
    <td>First</td>
    <td></td>
</tr>

<!-- More tests -->
</tbody></table>
</body>
</html>
```
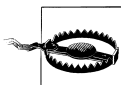
## Selenium Commands

Selenium features a rich command language called Selenese to allow the programmer to create tests. Pretty much any action that a user could take in the browser can be done in a Selenium command. You can create a Selenium test by creating a script consisting of a series of actions and tests.

> Dragging a file from the desktop to the browser "Drag and Drop" on page 71 is one thing that can not be done from Selenium, nor can it be easily tested from qUnit.

With very few exceptions, Selenium commands take as a parameter the location of an element in the DOM to be acted on. This location can be specified in one of several ways, including an element ID, an element name, XPath, CSS classes, a JavaScript call into the DOM, and link text. The options are illustrated in "Selenium Location options" on page 34.

> Using the Element ID will not work in ExtJS, as ExtJS assigns IDs to elements that will change each time. Use CSS classes or other properties of the HTML element. To denote buttons, it is often useful to use the text of the button with xpath like `//button[text()='Save']`. it is also possible to select on an attribute like `//img[@src='img.png']`.

---

### Selenium Location options

*ID*

Supply an HTML ID:

---

```
          id
```

*Name*
> Supply an element name (useful for form inputs):

```
name=username
```

*XPath*
> Use XPath to find an element:

```
//form[@id='loginForm']/input[1]
```

*CSS*
> Find an element by a CSS Selector, a procedure that's familiar to users of jQuery.
> However, the CSS Selector engine in Selenium is more limited than jQuery's:

```
css=div.x-btn-text
```

*Document*
> Use the DOM to find an element:

```
dom=document.getElementById('loginForm')
```

*Link text*
> Find by the text in the href attribute (useful for HTML links)

```
link='Continue'
```

The Selenium base commands do not include any capability for conditionals or loops. A Selenium HTML file is run sequentially from top to bottom, ending when an assertion fails or when the last command is executed. If you need flow control, use the goto_sel_ide.js plugin.

This plugin can be useful when looking for memory leaks or other problems that may occur in an application that users will run for a long time. Javascript still has a way to go to escape from the memory leaks that were not an issue when page reloads were frequent, resetting the state of Javascript and the DOM.

A large number of commands in Selenium can be used to construct tests. The Selenium IDE contains a reference for commands, so once you have learned some of the basics you can easily figure out the correct command for any given circumstance. Table 3-1 shows some of the common Selenium commands. They tend to come in two basic groups, *actions* and *assertions*. Actions include such things as click, type, dblclick, keydown, keyup and many more. Assertions provide the actual tests that allow Selenium to find out how user actions affect the page. Assertions can pause a script but make no changes in the page.

*Table 3-1. Selected Selenium Commands*

| Command | Target | Action |
| --- | --- | --- |
| open | Web page to open | Opens a web page |
| dblclick | Element to double-click | Double-clicks an element |
| click | Element to click | Clicks an element |
| mouseOver | Element over which to move the mouse | Replicates a mouseOver event |
| mouseUp | Element over which to let up the mouse button | Replicates a mouseUp event |
| mouseDown | Element on which to press the mouse button down | Replicates a mouseDown event |
| type | Use a xpath or other selector to choose the element, The third column is the text to type | Simulates text entry |
| windowMaximized | | Maximizes the current window |
| refresh | | Refreshes the browser. Can be useful for resetting Javascript State. |
| dragAndDrop | Offset to drag the element, selector is element to drag. | Drags and drops an element |

> In ExtJS, or any other custom widget, if a `click` event does not do what is expected, try using a `mouseDown`. To select a row in a grid, for example, use a `mouseDown` event instead of a `click`. When you click with a mouse, the browser sends 3 events: `mouseDown`, `mouseUp`, and `click`. Different elements in a User Interface may respond to any of them.

Actions in Selenium all have two forms: a simple form and a second form that will wait for a page to reload. The waiting form of the `click` command, for instance, is `clickAndWait`.

After a sequence of actions, it is necessary to verify that the application actually performed the correct actions. Most tests in Selenium test for the presence or absence of an element or section of text. For example to test adding a new element to an ExtJS grid, the test script would go something like this:

- Click the add button.
- Fill out a form giving default values.
- Submit new record to the server.
- Wait for the server to respond and verify that the text is present in the correct grid.

All of the assertions have three basic forms: the basic form, a verify form and a WaitFor form. The basic command will be similar to `assertElementPresent`, and will stop a test when the assertion fails. `verifyElementPresent` will check whether an element exists, but let the test continue if it is not. This is useful if there are multiple tests and you don't want them to stop after one failure. If an action is supposed to have a result that will

be delayed, use `waitForElementPresent`, which will pause the test script until the condition is met or the test times out. In summary:

*assert...*
> Check that something is true, and stop if it is not.

*verify...*
> Check that something is true, but continue even if it is false.

*waitFor..*
> Wait for an something to happen on the page (often used with Ajax).

## Constructing tests With the Selenium IDE

Selenium tests can be constructed by hand, but it is often much easier to automate it. The Selenium IDE plugin for Firefox will let you record your actions in the browser and save them as a test. The programmer will still have to put in the assertions and wait commands manually, and may have to make adjustments to the script that has been produced. The test is saved as a HTML document which can be checked into version control and run from the IDE as well as from an automatic test runner.

The IDE is a very nice way to try out options in Selenium. It will also let you create test scripts and run them directly from the IDE. The Selenium IDE also allows you to control how fast it executes the scripts, and single step through them.

The left side panel (which is hidden in Figure 3-1) shows a list of all the test cases that have been defined. They can all be run with the first button on the toolbar (just to the left of the speed control). The next button to the right executes just one test.

The bottom panel of the Selenium IDE features for tabs (and more can be added with plugins). The leftmost tab features a log of tests being run. The second tab is a reference panel. When you select a command from the menu in the middle panel, this tab will show information about the selected command, including what arguments it takes.

## Automatically running tests

It is possible to use one of the popular test suites (such as jUnit or phpUnit) to run Selenium tests, letting the tests run across multiple browsers and platforms. If you are running a regular set of unit tests, Selenium tests can be run from your normal test runner (Example 3-4).

> Most or all of what is described here for phpUnit should also work with minor modifications in all similar test suites for other languages.
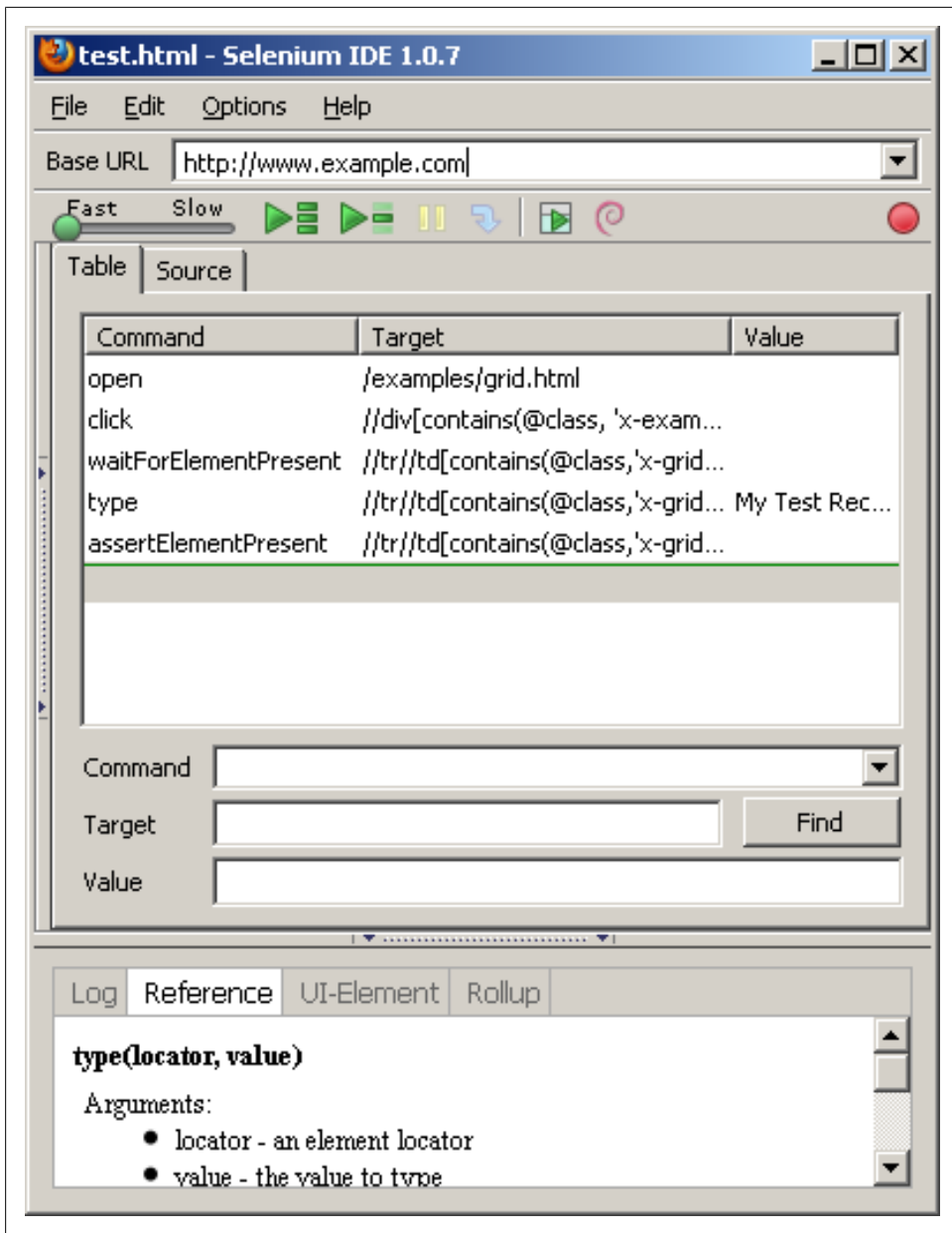
*Figure 3-1. Selenium IDE*

These tests will run just like any other test in the test enviroment. Each HTML file will run as a test in the test suite. (Details of this may vary depending on which test runner is being used). The test runner will run each of the HTML files in sequence in a manor similar to what was done in the IDE.

To run tests from phpUnit one or more test machines need to be designated to run the browsers. Each test machine needs a copy of the Selenium RC program running as well as having the browser or browsers to test installed. The *seleniumrc* binary is a java *.jar* file, so it will run on Windows, Linux, or a Mac.

When a test is run in phpUnit, the test class `PHPUnit_Extensions_SeleniumTestCase` will contact the *seleniumrc* program and ask it to start up a browser instance and then send it commands over a REST interface.

If mulitple browsers are listed in the `$browsers` static member, or via a *phpunit.xml* file (Example 3-5), the `PHPUnit_Extensions_SeleniumTestCase` class will run each test for each browser in sequence. In Example 3-4, for instance, it will run the tests on Safari, Firefox, Chrome, and Internet Explorer. Often it is better to list the browser options in the *phpunit.xml* file, because you can then create multiple files to help you change test options without changing test source code.

The following test just runs one Selenium test from the file *seleneseTest.html*. However we could have it automatically run an entire directory of Selenium HTML test files by setting the `$seleneseDirectory` property of the test class to the path to the files.

The phpUnit test runner can multiplex selenium tests across multiple browsers. Set the class static variable `$browsers` to a list of different browsers, and the test runner will run each test for each browser.

```
public static $seleneseDirectory = '/path/to/files';
```

*Example 3-4. Running Selenium from phpUnit*

```php
        <?php
require_once 'PHPUnit/Extensions/SeleniumTestCase.php';

class WebTest extends PHPUnit_Extensions_SeleniumTestCase
{
    protected $captureScreenshotOnFailure = TRUE;
    protected $screenshotPath = '/var/www/localhost/htdocs/screenshots';
    protected $screenshotUrl = 'http://localhost/screenshots';

    public static $browsers = array(
      array(
        'name'    => 'Safari on MacOS X',
        'browser' => '*safari',
        'host'    => 'mac.testbox',
        'port'    => 4444,
        'timeout' => 30000
      ),
      array(
        'name'    => 'Firefox on Windows',
        'browser' => '*firefox',
        'host'    => 'windows.testbox',
        'port'    => 4444,
        'timeout' => 30000
      ),
```

```php
    array(
      'name'    => 'Chrome on Windows XP',
      'browser' => '*googlechrome',
      'host'    => 'windows.testbox',
      'port'    => 4444,
      'timeout' => 30000
    ),
    array(
      'name'    => 'Internet Explorer on Windows XP',
      'browser' => '*iexplore',
      'host'    => 'windows.testbox',
      'port'    => 4444,
      'timeout' => 30000
    )
  );

  protected function setUp()
  {
      $this->setBrowserUrl('http://www.example.com/');
  }

  public function testSeleniumFile()
  {
      $this->open('http://www.example.com/');
      $this->runSelenese('seleneseTest.html');
  }

}
?>
```

*Example 3-5. phpunit.xml*

```xml
        <phpunit stopOnFailure="true"
    verbose="true"
    strict="true">
  <php>

  </php>
  <selenium>
    <browser name="Firefox"
        browser="*firefox"
        host="192.168.0.10"
        port="4444"
        timeout="30000"/>
    <browser name="Chrome"
        browser="*chrome"
        host="192.168.0.10"
        port="4444"
        timeout="30000"/>
    <!-- Other Browsers -->
  </selenium>
  <testsuites>
```

```
    <testsuite name="Selenium">
      <file>/path/to/MyTest.php</file>
    </testsuite>
  </testsuites>

</phpunit>
```

With all the advantages of Selenium RC, it has one major drawback: it can run only one test at a time. So if you hae a large test suite and a number of different browsers to run that test suite, the full test run can take many hours. Selenium Grid provides a solution to this, letting you run a number of tests in parallel on a group of machines. The Selenium Grid software along with examples and documentation can be found at *http://selenium-grid.seleniumhq.org/*.

If you don't want to build your own test farm, there are cloud selenium farms on the web that can be used. In addition, it is possible to run Selenium on Amazon's EC2 cloud service. This can be very useful for occasional users or for a new startup that may not have the resources to build and maintain a local selenium farm. This can also be very helpful in seeing how an application will perform over a remote network.

## Selenese Command Programing Interface

The Selenium test suite can run a test from a HTML file or directly from the unit test code. The Selenium RC server also has an API that can be called from unit test code in several languages. You can write tests for Selenium in PHP, Ruby, Python, Perl, Java, C#/.NET, and Perl. You can create code test cases for all of these languages via the Selenium IDE or by hand. The Selenium IDE will generate the skeleton of a test for you. To do this record a test in the IDE, the choose the output option for the language to run the test in. The tests will be converted to that language.

> To run Selenium on several browsers, you need to set up the Selenium server. See "Selenium RC and a test farm" on page 45.

Running Selenium directly from a unit test gives you the the full power of the language of the host language, notably its flow control, while the HTML style tests are much more limited. By using the API from a server side programming language, it is possible to create a very rich environment for scripting the web, and of course you have access to libraries on the server side to check data in a database or access web services. You can construct a test that will perform some actions in browser, then check the result in a database or against a log file.

Another advantage of using a server side testing is that if you are using any form of continuous integration, such CruiseControl or phpUnderControl, the Selenium tests

appear to the test system as just more tests in whatever language the team is using. In a team that is using a test framework, this will leverage the existing experience of the team.

Selenese commands consist in general of actions, assertions, and queries. The actions generally reflect the basic forms of Selenium actions: `click`, `mouseOver`, `mouseDown`, `type`, and so on. The various delayed forms that appear in the Selenium IDE are not present, but can be easily simulated by using a loop with a sleep function.

Many of the useful utility methods that are present in the HTML selenium test runner are not present in the server side Selenese interface. Methods such as `WaitForElement Present` or `WaitForElementNotPresent` are not included, but can be easily added in by creating a custom base class for the tests and adding them there.

*Example 3-6. WaitForElementPresent*

```
function waitForElementPresent($el, $timeout = 60)
{
    {
      if($this->isElementPresent($el))
        {
          return true;
        }
    sleep(1);
    }
  $this->fail("Element $el not found");
}
```

Queries allow the programmer to determine the state of the page after actions have happened. These queries allow the programmer to find out whether an element or piece of text exists on the page or otherwise find out about the current state of the page.

The Selenese API also features a number of methods to get data from the HTML document in the context of a unit test. To test whether a given text is present in the page, use the method `$this->isTextPresent()`, which is often quite useful to find out if an element is present or absent. To find out whether an element exists, use the method `$this->isElementPresent()`. To actually retrieve text from the DOM, use the `$this->getText()` method. This will take any form of selector that selenium can accept and will return the text of that element. If the element is not present, the method will throw an exception.

The xPath selector matches multiple elements on the page. It will normally return the first matched element. To find out how many elements are matched, use `$this->getX pathCount()`.

When building tests in PHP, synchronizing the test actions between the interface in the browser and the test itself running in PHP can be a challenge. It is intuitive to write a test that does something such as click one element and then immediately mouse over another. This will inevitably fail, as the Javascript will take some time (maybe 0.1 seconds) to create a piece of user interface or wait for data to load from the server. There are two ways to handle this.

The simple way is to place delays in the PHP code. A few well placed `sleep()` commands at the right place can cause a test to function correctly. However, this may cause a test to take longer to run that is necessary. To speed up tests using something like a `wait ForElementPresent()` with a 1/10 second delay between testing can cause the script to run faster as long as there is a way to tell from the DOM when the browser is ready for the next step in the test.

The second way is to use the method `$this->getEval()` in the Selenese interface to evaluate custom Javascript. Pass this method a string that contains the Javascript to be executed. When you call Javascript via `getEval()`, it will run in the window context of the test runner window, not the test window. Therefore, global variables must be prefixed by the global `window` object, which normally is not required.

In Example 3-7, Selenium executes `getEval()` in JavaScript to extract the global `session_id` variable.

*Example 3-7. Running Javascript from Selenese*

```
$session_id = $this->getEval('window.session_id'):
```

> It is also possible to use Selenium RC to set up manual tests, by having the script open up the page and perform all the steps leading up to the point where you need to test things. When it gets to the end point, it should pause for an extended period of time because the browser will close when the test ends. At the point that the test stops, a human can take over the browser and perform any manual actions that might be needed. This is often helpful when developing a multistep wizard or similar user interface.

## Running QUnit from Selenium

Selenium can run QUnit tests as well. To do so, load the QUnit page in Selenium and run the tests. It is also possible to choose to only run a subset of tests by passing parameters to the URL string. By integrating Selenium with QUnit, you can export the results of browser tests in QUnit into a test runner for continous integration.

Selenium just opens the QUnit URL and then stands back and waits for the test to finish. To let the test runner know whether the tests passed or failed, QUnit provides a simple micro format (Example 3-9) that shows how many tests were run and how many passed or failed. The unit test can then look for this data by an xPath selector and make sure all tests passed.

In Example 3-8, the PHP program opens up the QUnit test from the start of this chapter and then waits for the test to run. When the test finishes, the `qunit-testresult` element will be inserted into the DOM. At this point Selenium can find the number of tests that were run and how many passed or failed.

*Example 3-8. Selenium test to run QUnit*

```php
        <?php
require_once 'PHPUnit/Extensions/SeleniumTestCase.php';

class WebTest extends PHPUnit_Extensions_SeleniumTestCase
{
  function testQUnit()
  {
    //Put the URL of your HTML file with the QUnit test here
    $this->open('http://host.com/simple.html');

    $this->waitForElementPresent("//p[@id='qunit-testresult']");
    $failCount  = $this->getText("//p[@id='qunit-testresult']/span[@class='failed']");
    $passCount  = $this->getText("//p[@id='qunit-testresult']/span[@class='passed']");
    $totalCount = $this->getText("//p[@id='qunit-testresult']/span[@class='total']");
    $this->assertEquals($passCount, $totalCount,
                        "Check that all tests passed $passCount of $totalCount passed");
    $this->assertEquals("0", $failCount,
                        "Checking result of QUnit tests $failCount/$totalCount tests failed");

  }

  function waitForElementPresent($element, $timeout = 60)
  {
    $time = 0;
    while(!$this->isElementPresent($element))
      {
        $time++;
        if($time > $timeout)
        {
            throw New Exception("Timeout: $element not found!");
        }
        sleep(1);
      }
  }
}
```

*Example 3-9. QUnit Result Micro format*

```
        <p id="qunit-testresult" class="result">
   Tests completed in 221 milliseconds.<br/>
   <span class="passed">1</span> tests of
   <span class="total">2</span> passed,
   <span class="failed">1</span> failed.
 </p>
```

*Example 3-10. PHP code to extract data from QUnit*

```
        <?php
$failCount  = $this->getText("//p[@id='qunit-testresult']/span[@class='failed']");
$passCount  = $this->getText("//p[@id='qunit-testresult']/span[@class='passed']");
$totalCount = $this->getText("//p[@id='qunit-testresult']/span[@class='total']");
$this->assertEquals($passCount, $totalCount,
                     "Check that all tests passed $passCount of $totalCount passed");
$this->assertEquals("0", $failCount,
                     "Checking result of QUnit tests $failCount/$totalCount tests failed");
```

## Selenium RC and a test farm

It is important to make sure that an application runs well not just on one browser, but on a wide number of browsers and platforms. In most cases you can assume that your application may be run on Windows XP, Windows Vista, Windows 7, MacOS X, and Linux. In addition, the users may be using some combination of Firefox, Chrome, Internet Explorer, Safari and Opera, and probably several different versions of each.

The implementations of JavaScript and various interfaces are similar among these different browsers, and using a framework like jQuery will smooth over some of the differences, but the browsers are still not exactly the same. So code that works well in Firefox may suddenly break in Chrome or Safari. And, of course, code could will work well in one version of a browser but not on an older or newer version. So testing across a wide matrix of browsers is vital. However, it is probably too much to expect a QA team to do this manfully, hence the need for automation.

Selenium RC makes it possible to test all of these various combinations using a network of machines. Each test machine must have Java installed. In many cases Java will be installed by default, but if it is not, download and install it. Then download the Selenium RC package from *http://seleniumhq.org/download/* and unpack it. On each server, start the Selenium server jar with the command shown bellow. It is probably a good idea to have the Selenium server start up automatically on machine boot in the case of a machine that will be a member of a test farm. Normally the Selenium server will be installed with reasonable defaults, but it offers a number of command-line options that

allow some degree of customization. Specifically, you can change the port from the default 4444, if needed, with the `-port` option. This can be used to run several instances of the server on one server in order to test on several browsers at once.

```
java -jar selenium-server.jar [-port 4444]
```

> It is not necessary to use a separate physical machine for each test server. Any of the common virtual machine technologies can work well for this. A reasonably powerful server with enough RAM should be able to run a small but useful virtual test farm.

If you need to test your application on Android or iOS Selenium, can do that as well. There is an Android Driver for Selenium *http://code.google.com/p/selenium/wiki/AndroidDriver* as well as an iPhone driver *http://code.google.com/p/selenium/wiki/IPhoneDriver*. Note that in order to run the iPhone driver you need to have a Mac as well as the iPhone development set up. Because the Selenium driver is not in the iPhone store, you need to be able to install it on your phone with a provisioning profile, or use the simulator in XCode.

# Local Storage

The web browser provides Javascript a wonderful environment for building applications that run in the browser. Using ExtJS or jQuery, it is possible to build an application that for many uses can rival what can be done in a desktop application, and provide a method of distribution that is about as simple as it gets. But however nice the browser is in terms of providing a user experience, used to fall flat when it comes to data storage.

Historically, browsers did not have any way to store data. They were in effect the ultimate thin client. The closest that could happen was the HTTP cookie mechanism, which allows a piece of data to be attached to each HTTP request. However, cookies suffer from several problems. First of all, each cookie is sent back and forth with every request. So the browser sends the cookie for each Javascript file, image, Ajax request, and so on. This can add a lot of bandwidth use for no good reason. Secondly, the cookie specification tried to make it so that a cookie could be shared between different sub domains. If a company had `app.test.com` and `images.test.com`, a cookie could be set to be visible to both. The problem with this is that outside of the USA, three-part domain names become common. For example it would be possible to set a cookie for all of the hosts in `.co.il` which would allow a cookie to leak to almost every host in Israel. And it is not possible to simply require a three-part domain name whenever the name contains a country suffix, because some countries such as Canada do not follow the same convention.

Having local storage on the browser can be a major advantage in terms of speed. A normal Ajax query can take anywhere from half a second to several seconds to execute, depending on the server. However, even in the best possible case it can be quite slow. A simple ICMP ping between my office in Tel Aviv and a server in California will take an average of about 250ms. Of that 250ms, a large part is probably due to basic physical limitations: data can travel down the wire at just some fraction of the speed of light. So there is very little that can be done to make that go faster, as long as the data has to travel between browser and sever.

Local storage options are a very good option for data that is static or mostly static. For example, many applications have a list of countries as part of their data. Even if the list

includes some extra information, such as whether a product is being offered in each country, the list will not change very often. In this case it often works to have the data preloaded into a Local Storage object, and then do a conditional reload when necessary so that the user will get any fresh data, but not have to wait for the current data.

Local Storage is, of course, also essential for working with a web application that may be offline. Although Internet access may seem to be everywhere these days, it should not be regarded as universal, even on smart phones. Users with devices such as the iPod touch will have access to the Internet only where there is WiFi, and even smart phones like the iPhone or Android will have dead zones where there is no access.

With the development of HTML 5, a serious movement has grown to provide the browser with a way to create a persistent local storage, but the results of this movement have yet to gel. There are currently at least three different proposals for how to store data on the client.

In 2007, as part of Gears, Google introduced a browser-based SQLite database. Webkit-based browsers, including Chrome, Safari, and the browsers on the iPhone and Android phones, have implemented a version of the Gears SQLite database. However, SQLite was dropped from the HTML5 proposal because it is a single-sourced component.

The `localStorage` mechanism provides a Javascript object that persists across web reloads. This mechanism seems to be reasonably well agreed on and stable. `localStorage` is good for storing small-sized data such as session information or user preferences.

This chapter explains how to use current `localStorage` implementations. In the following chapter, we'll look at a more complicated and sophisticated form of local storage that has appeared on some browsers, IndexedDB.

## The localStorage and sessionStorage Objects

Modern browsers provide two objects to the programmer for storage, `localStorage` and `sessionStorgage`. Each can hold data as keys and values. They have the same interface and work in the same way, with one exception. The `localStorage` object is persistent across browser restarts, while the `sessionStorage` object resets itself when a browser session restarts. This can be when the browser closes or a window closes. Exactly when this happens will depend on the specifics of the browser.

Setting and getting these objects is pretty simple, as shown in Example 4-1.

*Example 4-1. Accessing localStorage*

```
//set
localStorage.sessionID = sessionId;
localStorage.setItem('sessionID', sessionId);

//get
```

```
var sessionId;
sessionId = localStorage.sessionID;
sessionId = localStorage.getItem('sessionId');

localStorage.sessionId = undefined;
localStorage.removeItem('sessionId');
```

Browser storage, like cookies, implements a "same origin" policy, so different web sites can't interfere with each other or read each other's data. But both of the storage objects in this section are stored on the user's disk (as cookies are), so a sophisticated user can find a way to edit the data. Chrome's developer tools allow a programmer to edit the storage object, and you can edit it in Firefox via Firebug or some other tool. So while other sites can't sneak data into the storage objects, these objects still should not be trusted.

Cookies are burdened with certain restrictions: they are limited to about 4KB in size and must be transmitted to the server with every Ajax request, greatly increasing network traffic. The browser's `localStorage` is much more generous. The HTML5 specification does not list an exact size limit for its size, but most browsers seem to limit it to about 5 MB per web host. The programmer should not assume a very large storage area.

Data can be stored in a storage object with direct object access or with a set of access functions. The session object can store only strings, so any object stored will be type cast to a string. This means that an object will be stored as `[object Object]`, which is probably not what you want. So to store an object or array, convert it to JSON first.

Whenever a value in a storage object is changed, it fires a storage event. This event will show the key, its old value, and its new value. A typical data structure is Example 4-2. Unlike some events, such as clicks, storage events cannot be prevented. There is no way for the application to tell the browser to not make a change. The event simply informs the application of the change after the fact.

*Example 4-2. Storage Event Interface*

```
var storageEvent = {
    key: 'key',
    oldValue: 'old',
    newValue: 'newValue',
    url: 'url',
    storageArea: storage // the storage area that changed
};
```
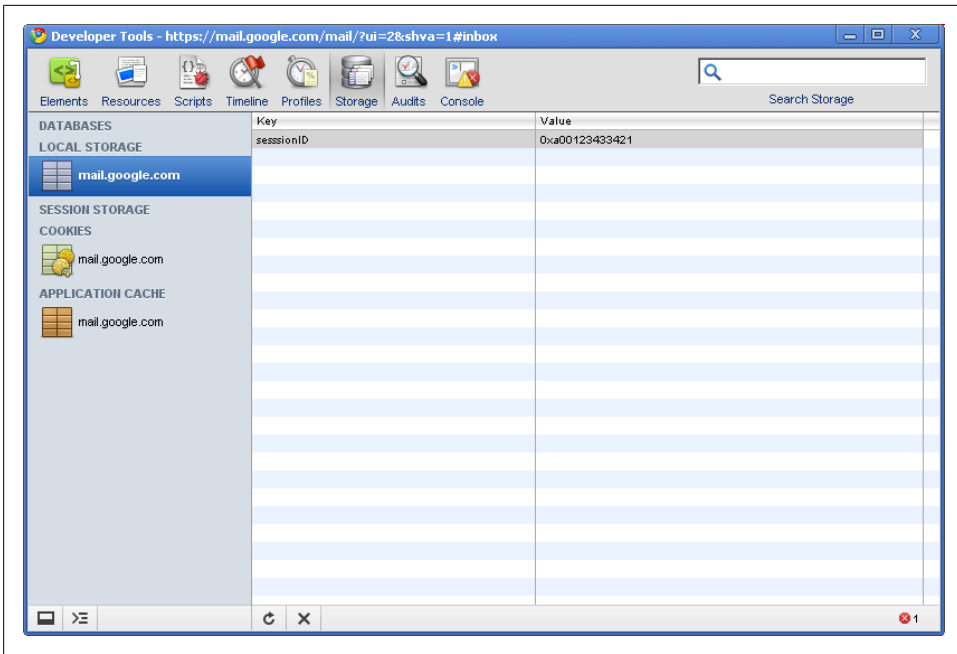
*Figure 4-1. Chrome Storage Viewer*

Webkit provides a screen in its developer tools where a programmer can view and edit the `localStorage` and `sessionStorage` objects (Figure 4-1). From the Developer tools, click on the Storage tab. It will show the `localStorage` and `sessionStorage` objects for a page. The Storage screen is also fully editable: keys can be added, deleted and edited there.

Although Firebug does not provide an interface to the `localStorage` and `sessionStorage` objects as Chrome and other Webkit-based browsers do, the objects can be accessed via the Javascript console, and you can add, edit, and delete keys there. Given time, I expect someone will write a Firebug extension to do this.

Of course, it is possible to write a customized interface to view and edit the storage objects on any browser. Create a widget on screen that exposes the objects using the `getItem` and `removeItem` calls shown in Example 4-1 and allow editing through text boxes. The skeleton of a widget is shown in Example 4-3.

*Example 4-3. Storage Viewer*

```
(function createLocalStorageViewer()
{
  $('<table></table>').attr(
  {
    "id": "LocalStorageViewer",
    "class": 'hidden viewer'
  }).appendTo('body');
```

```
    localStorage.on('update', viewer.load);

  var viewer =
  {
    load: function loadData()
    {
      var data, buffer;
      var renderLine = function (line)
      {
        return "<tr key='{key}' value='{value}'>\n".populate(line) +
          "<td class='remove'>Remove Key</td>" +
          "<td class='storage-key'>{key}</td><td>{value}</td></tr>".populate(line);
      };

      buffer = Object.keys(localStorage).map(function (key)
      var rec =
      {
        key: key,
        value: localStorage[data]
      };
      return rec;
      });

  };

  $("#LocalStorageViewer").html(buffer.map(renderLine).join(''));

  $("#LocalStorageViewer tr.remove").click(function ()
  {
    var key = $(this).parent('tr').attr('key').remove();
    localStorage[key] = undefined;
  });

  $("#LocalStroageViewer tr").dblclick(function ()
  {
    var key = $(this).attr('key');
    var value = $(this).attr('value');
    var newValue = prompt("Change Value of " + key + "?", value);
    if (newValue !== null)
    {
      localStorage[key] = newValue;
    }
  });
};
}());
```

## Using localStorage in ExtJS

ExtJS, some examples of which appeared in earlier chapters, is a very popular JavaScript framework allowing very sophisticated interactive displays. This sections shows how to use localStorage with ExtJS.

One nice feature of ExtJS is that many of its objects can remember their state. For example, the ExtJS grid object allows the user to resize columns, hide and show them, and reorder them, and these changes are remembered and redisplayed when a user comes back to the application later. This allows each user to customize the way the elements of an application work.

ExtJS provides an object to save state, but uses cookies to store the data. A complex application can create enough state to exceed the size limits of cookies. An application with a few dozen grids can overflow the size of a cookie, which can lock up the application. So it would be much nicer to use the `localStorage`, taking advantage of its much larger size and avoiding the overhead of sending the data to the server on every request.

Setting up a custom state provider object is in fact pretty easy. The provider in Example 4-4 extends the generic provider object and must provide three methods: `set`, `clear`, and `get`. These methods simply read and write the data into the store. In Example 4-4 I have chosen to index the data in the store with the rather simple method of using the string `state_` with the state ID of the element being saved. This is a reasonable method.

*Example 4-4. ExtJS Local State Provider*

```
Ext.ux.LocalProvider = function() {
    Ext.ux.LocalProvider.superclass.constructor.call(this);
};

Ext.extend(Ext.ux.LocalProvider, Ext.state.Provider, {
    //***********************************************************
    set: function(name, value) {
        if (typeof value == "undefined" || value === null) {
            localStorage['state_' + name] = undefined;
            return;
        }
        else {
            localStorage['state_' + name] = this.encodeValue(value);
        }
    },

    //***********************************************************
    // private
    clear: function(name) {
        localStorage['state_' + name] = undefined;
    },

    //***********************************************************
    get: function(name, defaultValue) {
        return Ext.value(this.decodeValue(localStorage['state_' + name]), defaultValue);
    }
});

// set up the state handler
Ext.onReady(function setupState() {
    var provider = new Ext.ux.LocalProvider();
```

```
    Ext.state.Manager.setProvider(provider);
});
```

It would also be possible to have all the state data in one large object and to store it into one key in the store. This has the advantage of not creating a large number of elements in the store, but makes the code more complex. In addition, if two windows try to update the store, one could clobber the changes made by the other. There is no great solution to the issue of race conditions here. In general in places where it can be a problem it is probably better to use IndexedDB or some other solution.

## Offline Loading With a Data Store

When some of the persistent data used in an application will be relatively static, it can make sense to load it to local storage for faster access. In this case, the `Ext.data.Json Store` object will need to be modified so that its `load()` method will look for the data in the `localStorage` area before attempting to load the data from the server. After loading the data from `localStorage`, `Ext.data.JsonStore` should call the server to check whether the data has changed. By doing this, the application can make the data available to the user right away at the cost of possibly short term inconsistency. This can provide make a user interface feel faster to the user and reduce the amount of bandwidth that the application uses.

For most requests, the data will not have changed, so using some form of ETag for the data makes a great deal of sense. The data is requested from the server with a HTTP GET request and an `If-None-Match` header. If the server determines that the data has not changed, it can send back a `304 Not Modified` response. If the data has changed, the server sends back the new data, and the application loads in into both the `Ext.data.JsonStore` object and the `sessionStorage` object.

The `Ext.data.PreloadStore` object stores data into the session cache as one large JSON object (Example 4-5). It further wraps the data that the server sends back in a JSON envelope, which allows it to store some meta data with it. In this case, the ETag data is stored as well as the date when the data is loaded.

*Example 4-5. Ext.data.PreloadStore offline data format*

```
{
    "etag": "25f9e794323b453885f5181f1b624d0b",
    "loadDate": "26-jan-2011",
    "data": {
        "root": [{
            "code": "us",
            "name": "United States"
        },
        {
            "code": "ca",
            "name": "Canada"
```

```
        }]
    }
}
```

> When building an ETag, make sure to use a good hash function. MD5 is probably the best choice. SHA1 can also be used, but since it produces a much longer string it is probably not worthwhile. In theory, it is possible to get a MD5 collision, but in practice for cache control it is probably not something to worry about.

Data in the `localStorage` object can be changed in the background. As I already explained, the user can change the data from the Chrome Developer tools or from the Firebug command line. Or it can just happen unexpectedly because the user has two browsers open to the same application. So it is important for the Store to listen for an update event from the `localStorage` object.

Most of the work is done in the `beforeload` event handler. This handler checks the data store for a cached copy of the data, and if it is there, loads it into the store. If there is data present, the handler will reload the data as well, but use the `Function.defer()` method to delay the load until a time when the system has hopefully finished loading the web page, so that doing the load will be less likely to interfere with the user.

The `store.doConditionalLoad()` method makes an Ajax call to the server to load the data. It includes the `If-None-Match` header so that, if the data has not changed, it will load the current data. It also includes a `force` option that will cause the `beforeload` handler to actually load new data and not try to refresh the store from the `localStor age` cached version of the object.

I generally define constants for `SECOND`, `MINUTE`, and `HOUR` simply to make the code more readable.

*Example 4-6. Ext.data.PreloadStore*

```
Ext.extend(Ext.data.PreloadStore, Exta.data.JsonStore, {
    indexKey: '',

    //default index key
    loadDefer: Time.MINUTE,
    // how long to defer loading the data
    listeners: {
        load: function load(store, records, options) {
            var etag = '25f9e794323b453885f5181f1b624d0b';
            var jsonData = this.reader.jsonData;
            var data = {
                etag: etag,
                date: new date(),
                data: jsonData
```

```
            };
            sessionStorage[store.indexKey] = Ext.encode(data);

        },
        beforeload: function beforeLoad(store, options) {
            var data = sessionStorage[store.indexKey];
            if (data === undefined || options.force) {
                return true; // Cache miss, load from server
            }
            var raw = Ext.decode(data);
            store.loadData(raw.data);
            // Defer reloading the data until later
            store.doConditionalLoad.defer(store.loadDefer, store, [raw.etag]);
            return false;
        }
    },
    doConditionalLoad: function doConditionalLoad(etag) {
        //TODO check that this works
            this.proxy.headers["If-None-Match"] = etag;
            this.load({force: true});
        });
    },
    forceLoad: function() {
        // Pass in a bogus ETag to force a load
        this.doConditionalLoad('');
    }
});
```

## Storing Changes for a Later Server Sync

In the event that an application may be used offline, or with a flaky connection to the Internet, it can be nice to provide the user a way to save her changes without actually needing the network to be present. To do this, write the changes in each record to a queue in the localStorage object. When the browser is online again, the queue can be pushed to the server. This is similar in intent to a transaction log as used in a database.

A save queue could look like Example 4-7. Each record in the queue represents an action to take on the server. The exact format will of course be determined by the needs of a specific application.

*Example 4-7. Save Queue*

```
[
  {
      "url": "/index.php",
      "params": {}
  },
  {
      "url": "/index.php",
      "params": {}
  },
```

```
    {
        "url": "/index.php",
        "params": {}
    }
]
```

Once the web browser is back online, it will be necessary to process the items in the queue. Example 4-8 takes the queue and sends the first element to the server. If that request is a success, it will take the next element and continue walking down the queue until the entire queue has been sent. Even if the queue is long, this process will execute it with minimal effect on the user because Ajax processes each item in an asynchronous manner. To reduce the number of Ajax calls it would also be possible to change this code to send the queue items in groups of say five at a time

*Example 4-8. Save Queue*

```
var save = function save(queue)
{
  if (queue.length > 0)
  {
    $.ajax(
    {
      url: 'save.php',
      data: queue.slice(0, 5),
      success: function (data, status, request)
      {
        save(queue.slice(5));
      }
    });
  }
};
```

# JQuery Plugins

If the uncertainty of all the client-side storage options is enough to drive you crazy, you have other options. As with many things in Javascript, a bad and inconsistent interface can be covered up with a module that provides a much better interface. Here are two such modules that can make life easier.

## DSt

DSt *http://github.com/gamache/DSt* is a simple library that wraps the `localStorage` object. DSt can be a free-standing library or work as a JQuery plugin. It will automatically convert any complex object to a JSON structure.

DSt can also save and restore the state of a form element or an entire form. To save and restore an element, pass the element or its ID to the `DSt.store()` method. To restore it later, pass the element to the `DSt.recall()` method.

To store the state of an entire form, use the `DSt.store_form()` method. It takes the ID or element of the form itself. The data can be restored with the `DSt.populate_form()` method.

*Example 4-9. DSt Interface*

```
$.DSt.set('key', 'value');
var value = $.DSt.get('key');

$.DSt.store('element'); // Store the value of a form element
$.DSt.recall('element'); // Recall the value of a form Element

$.DSt.store_form('form');
$.DSt.populate_form('form');
```

## jStore

If you don't want to venture to figure out which storage engines are supported on which browsers and create different code for each case, there is a good solution: the jStore plugin for jQuery. This supports `localStorage`, `sessionStorage`, Gears SQLite, and HTML 5 SQLite, as well as Flash Storage and IE 7's proprietary solution.

The jStore plugin has a simple interface that allows the programmer to store name/value pairs in any of its various storage engines. It provides one set of interfaces for all the engines, so the program can degrade gracefully when needed in situations where a storage engine doesn't exist on a given browser

The jStore plugin provides a list of engines that are available in the `jQuery.jStore.Avail ability` instance variable. The program should select the engine that makes the most sense.

For applications that require multi browser support this can be a useful addition. See the jStore web page for more details.

# IndexedDB

The Local Storage Interface ([Chapter 4](#)) provides a very nice interface for storing small amounts of data, however the browser limits this storage space to 5mb in many cases. If the storage needs of an application go beyond that, or if the application needs to able to have query access to structured data, Local Storage is not the best choice. In this case, having a more robust strorage mechanism can be useful for the application developer. IndexedDB provides this mechanism on mny browsers. For example a programmer may store product catalog data into an IndexedDB store so that when the user searches for an item, the server does not need to make a trip to the server to find the data for a particular item.

IndexedDB is a NoSQL database that will feel familiar to people who have used such products as MongoDB or CouchDB. A program can store Javascript data directly into an IndexedDB data store.

IndexedDB has been in Firefox from version 4 on. It has also been introduced into Google Chrome starting with version 11. It will probably be available in other browsers in the next year or two. Formally, IndexedDB is now a [draft proposal from the W3C](#).

> Because IndexedDB is supported by only two browsers at this point, use should probably be restricted to internal applications where you can limit use to browsers of your choice. Using it in a general app should be done with extreme caution, as Microsoft Internet Explorer, Opera, and Safari do not support this feature yet (at least as of July 2011).

Like localStorage, IndexedDB has a strict same origin policy. So a database created by one page can not be accessed by pages on other hosts. This provides a level of security for data, in that it is protected from scripts running on other pages. But any script running on the page that created the database has full access to the database, and of course the user has access to the data.

IndexedDB has several advantages over SQLite. First of all, its native data storage format is a Javascript object. There is no need to map a Javascript object into a SQL table

structure, which is always a poor fit and can allow for SQL Injection attacks. Injection attacks can't happen with IndexedDB, though in some cases XSS could be a problem, if a user manages to inject Javascript into the store and have it put back into a page.

The IndexedDB data store provides a set of interfaces to store Javascript objects on a local disk. Each object must have a key by which objects can be retrieved, and may also have secondary keys.

To show how to use IndexedDB I will develop a small application that uses IndexedDB to store and work with a set of data. In this case it will show a limited index of O'Reilly books. Each book will have a record in the format as in Example 5-1. In this case there will be indexes on the title and price fields so that the user can search on

*Example 5-1. Setting the transaction version*

```
{
    "title": "Real World Haskell",
    "price": 49.95,
    "price_can": 49.95,
    "authors": [
        "Bryan O'Sullivan",
        "John Goerzen",
        "Don Stewart"
    ],
    "cover_animal": "Rhinosorus Beatle",
    "cover_url": "http://....",
    "topics": ["Haskell"]
}
```

IndexedDB keeps applications and related data stores in sync as they evolve, through a built-in versioning system. Each Datastore has a version that the an application can check when it loads. If the version is not current, the application can then take appropriate actions to make it current by creating new object stores and indexes.

Versions should be changed when a new data store is being added, or an index is being added. When an Index or Store is removed then a version must also be changed.

Version should changes as the needs of an application change. In our example the initial indexes are on the Title and the price. However it is possible that after the application has been launched that someone wishes to add an index on the topics field, so users can search by that field. This will require a new version of the store to add the index. The program when loaded can check the version number and know that if it is version 1.0 that it should run the upgrade to 1.1 which has the new index.

*Example 5-2. Setting the transaction version*

```
// TODO: Add a new index on the topics field
```

Interactions with IndexedDB must be done by way of transactions. This because the IndexedDB interface is asynchronous, and IndexedDB can be accessed from a web worker or a second window running another thread (each window in Javascript runs its own thread). So even though simulaneous accesses will be rare, it is possible for more than one Javascript thread to access a given IndexedDB database at the same time, and the interface must protect against race conditions.

> The IndexedDB interface spec includes a synchronous version of the interface that can be used in Web Workers, however it has not been implemented in browsers yet. In addition using it will mean that any code which uses it can not be reused between a WebWorker and non WebWorker contexts.

The fundemental unit of data storage in IndexedDB is the *database*. A database in IndexedDB is roughly comparable to a database in a relational product like MySQL. Each IndexedDB database contains one or more Object Stores, which can be considered equivalent to tables in a SQL database. However, unlike a SQL database, an Object Store has no fixed schema.

Each record in an Object Store is a key/value pair, where the key is the primary index and the value is a Javascript object. Any Javascript object that can be serialized can be inserted into the Object Store.

> Closures and functions in general can not be stored in IndexedDB

The first step in using the IndexedDB is to create a new database. To do that, just pick a name and request that it be opened. If the Database does not exist, it will be created. If it does exist, it will be opened.

To access the database, call the IndexedDB open method:

```
indexedDB.open(db)
```

Note that in older versions of Firefox and Chrome the IndexedDB interface may be `mozIndexedDB` or `webkitIndexedDB`, so be sure to check which interfaces exist and use the approprate ones. This chapter will use `indexedDB` exclusively.

After opening a database, you need to create a transaction object. The call creating this object will take a list of object stores to be used, as well as an optional mode variable and timeout. The mode variable can be `IDBTranscation.READ_ONLY`, `IDBTransca tion.READ_WRITE`, or `IDBTranscation.VERSION_CHANGE`, the default being `IDBTransca`

tion.READ_ONLY. The timeout is specified in milliseconds, or can be set to 0 for no time out. The

In a server side database an administrator can check if the orginziational state of the database is correct when software is updated. In IndexedDB the datastore is being run on the user's computer out there on the net. As such the program itself must be able to check if the store is setup correctly and if it is not fix it.

IndexedDB does not specify a format for versions, nor does it require that they be in order. It just records the current version and then allows the programmer to work with that data. A good approach would be to have methods to change from each version to the next, and when a page loads check what version is current and roll forward to the most recent. If there is no version, start with the initial version. So in an old project a first time visitor may get a sequence of versions like this: `1.0 -> 1.1 -> 2.0 -> 3.0 -> 4.0 -> 4.1 -> 5.0`.

In this evolution each new version will introduce a change in the stores or indexes. For example it may be that in version 3.0 it was found that a specific store needed a new index, so it was added, but the feature that needed it is removed in version 5.0 so that index can then be removed.

To facilitate this IndexedDB supports a versioning operation. Each version of the soft-ware that has different needs for an object store must have a version that is set with the the `db.setVersion()` method.

the `db.setVersion(version)` method creates a `VERSION_CHANGE` transaction that is the only place where Object stores and indexes can be created, changed or deleted. Thus ensuring that the state of an Object Store is consistent with the version that has been set. Thus the program can assume that if the store says it is version 3.0 that it will have the object stores and indexes that go with version 3.0.

*Example 5-3. Setting the transaction version*

```
if (db.version !== "1.0") {
    var request = db.setVersion("1.0");
    request.onsuccess = function (event) {
        var store = db.createObjectStore("calendar", {
            keyPath: "startDate"
        });
    };
}
```

It's a good practice to include a chain of update operations. So if a user shows up with version 1.0 of the schema and the current version is 4.0, the application in his browser will run the code to upgrade from 1.0 to 2.0, then 2.0 to 3.0, and finally 3.0 to 4.0.

An IndexedDB transaction is contained in a Javascript function. When the function completes the transaction is commited. To abort a transaction call the `abort()` method. (What object does that work on?, will using throw work?).

# Adding and Updating Records

All data being added to a IndexedDB database must be done inside a transaction, as I explained earlier. This provides safety from other Javascript processes that may try to modify that same data. Even though JavaScript is single threaded, it is possible to open up the datastore from a web worker or from a second window in the same browser both would be able to modify data at the same time.

Data is added to a store with the `store.put()` or `store.add()` methods, which take a Javascript Object to be added to the data store. The difference between `store.put()` and `store.add()` is that `put` can also update existing data. An attempt to `add` an object whose key already exists will fail.

The `store.put()` method returns a transaction object, for which you can define three callbacks. The main callback you need is `onsuccess`, which will be called if the transaction is successful. Example 5-4 shows a typical sequence of events: a `store.put()` call attempts to add a new entry to a calendar, and on success, the callback redraws the calendar visible to the user.

The other two callbacks concern transaction failures. If the database refuses to handle a transaction, `onerror` will be called, while if the transaction is aborted `onabort` will be called. A transaction can be aborted with the `abort()` method of the transaction object.

*Example 5-4. Putting data*

```
calendar.indexedDB.addCalendar = function (calendarEntry) {
    var db = indexedDB.db;
    var trans = db.transaction(["calendar"], IDBTransaction.READ_WRITE, 0);
    var store = trans.objectStore("calendar");
    calendarEntry.timeStamp = new Date().getTime();
    var request = store.put(calendarEntry);

    request.onsuccess = function (e) {
        redrawCalendar();
    };

    request.onerror = function (e) {
        console.log(e.value);
    };
};
```

# Adding Indexes

Indexes can be added or removed only in a `setVersion` transaction. It is possible to check the current version of a store on a customer's computer and, if needed, bring it up to the current version. For example, version 1.0 of the software there have two stores, while in version 1.1 a third store has been added and one of the origninal tables has had a new index added. By checking the version, your code can know if it should apply any needed updates or if the user's datastore is set up correctly.

One key difference between a datastore like IndexedDB and most other data storage is that IndexedDB runs on the customer's browser, so updating it requires a little more thought than a data storage mechanism that runs on a small set of servers in a central location. The Javascript code must be able to cope with the possibility that the user may have an out-of-date store and update on the fly.

An index can be created with the data store in the `createObjectStore()` method. But it can also be added later with the `store.createIndex()` method. This method takes three parameters:

*name*
> The name of the index.

*keypath*
> The path to the data to be indexed.

*unique*
> An optional boolean that causes a unique index to be created. If a unique index is created with non-unique data it will raise a `CONSTRAIN_ERR` exception.

An index that is no longer needed can be removed with the `store.deleteIndex(name)` method. This method takes the same name that was used with `store.createIndex()` and will delete the index.

*Example 5-5. Adding an Index*

```
var request = db.setVersion("1.1");
request.onsuccess = function (event) {
    var objectStore = db.createObjectStore("customers", {
        keyPath: "with"
    });
    objectStore.createIndex("with", "with", {
        unique: false
    });
};
```

# Retrieving data

Of course, being able to add data to a store would not be of much use if it could not be pulled back out in useful ways to be displayed or otherwise processed. Again create a `store` object but unlike the `add` method here we are going to create a `IDBKeyRange` object that can specify bounds on our query. The `IDBKeyRange` object is then passed to the `store.openCursor` method which returns a cursor. The Cursor object will call its `onsuccess` method when the data has been retrieved.

The success handler will recieve an event object that will contain the first record of the result set. The success method should then process this data and call the `result.con tinue()` method which will cause indexeddb to call the success handler again with the next element in the result set.

In Example 5-5 The data is extracted using a cursor. The cursor iterates over over the lines of the result set until there are no more left. In this example each line is rendered into HTML and added to an element.

*Example 5-6. Listing Data*

Mozilla has also implemented a `getAll()` method on the objectStore which will return the entire contents of the store as an array. This is not part of the IndexedDB standard and they warn that it may go away in future, so use with caution. But if you need to get the entire contents of a store as an array this is the way to do it.

Of course there are times when you might not want every record in a store, or even most of them but just one, In this case using one of the indexes that have been defined on the data makes sense. The `index.get()` method will return one row of data as matched by an index. This is useful for a unique index.

*Example 5-7. Quering an Index*

```
var index = objectStore.index("startDate");
index.get(date).onsuccess = function (event) {
    $('.alert').html("EVENT: " + event.targt.result.what).fadeIn(500);
};
```

Other times it may be useful to get a range of items from the store. In Example 5-8 The query will find all events in the calendar that are with "Bill". However the `IDBKeyR ange` object can also specify ranges as well as upper and lower bounds for a query. To specify a lower bound of "Bill" use the method `IDBKeyRange.lowerBound("Bill")`, by adding a second element of true it is possible to make it show all items greater than "Bill". Similarly `IDBKeyRange.upperBound()` will specify all items less than its value in-

clusive or not as needed. The `IDBKeyRange.bound("Ashdod", "Tel Aviv")` method would give us a list of all items between the cities of Ashdod and Tel Aviv.

*Example 5-8. Quering and Index by Range*

```
var withBill = IDBKeyRange.only("Bill");
index.openCursor(withBill).onsuccess = function (event) {
    var cursor = event.target.result;
    if (cursor) {
        alert("Name: " + cursor.key + ", SSN: " + cursor.value.ssn + ", email: " + cursor.value.email);
        cursor.
        continue ();
    }
}
```

# Deleting data

Deleting data from an object store is also pretty easy. Each object store has a `delete()` method that can be called with the index of the items to be deleted. The store will then delete those items and call the callback.

*Example 5-9. Deleting Data*

```
var request = db.transaction([datastore], IDBTransaction.READ_WRITE).objectStore(store).delete(key);
request.onsuccess = function (evt) {
    console.log("Key Removed");
};
```

# Handling Errors

Each element in the IndexedDB interface features an optional on `onerror` handler which will be called in the case of some sort of error. These error events also bubble. So if an error is called on an insert and not handled it will then move up to the Store and then to the database object. In Example 5-10 is presented a very simple error handler that will just log errors to the firebug console.

*Example 5-10. Error*

```
db.onerror = function (evt) {
    console.error(evt);
};
```

# Files

For obvious reasons, the browser has historically had very limited ability to access the files system. HTML forms have been able to upload files, and certain HTTP headers make it possible for a user to download files from the server. But outside of those specific features, the browser has not been able to access files on the local file system. In general this is a good thing. I don't want every web page I visit to be able to look around my hard drive!

Some of the new features HTML5 give the browser limited access to the file system. Newer browsers allow Javascript to access the files via the exiting form file input. Historically a browser could upload a file from a form, but now it is possible to use the data from the file directly in Javascript. In addition, the browser now lets you drag files from your desktop to a web application. Google's gmail uses this feature to allow the user to attach files. This had been done with Flash previously, but now can be done with just JavaScript.

This does not create any new security problems, because the application already had access to this data by uploading it to the server then downloading it again into the browser.

As of this writing these features are supported in Firefox, Chrome, and Opera. For Safari and Internet Explorer the Flash plugin will work to allow file drag and drop.

## Blobs

Javascript has always been good at working with strings and numbers, but binary data has never been its strong point. But recently, it has added a blob data type and interfaces to work with blobs ("binary large objects"). JavaScript sees a blob as a big chunk of raw data. So the amount of actual manipulation that JavaScript can do on a blob is actually somewhat limited. However, blobs are very important for moving binary data around. Blobs can be read from and written to files ("Working With Files" on page 69), used as URL's, saved in the IndexedDB (Chapter 5 on page 59), passed to and from a Web Worker (Chapter 8).

You can create a new blob with the `BlobBuilder` interface. This creates a basic empty BlobBuilder, to which is it possible to append a string, an existing blob, or binary data from an `ArrayBuffer`. The `BlobBuilder.getBlob()` method returns the actual blob.

BlobBuilder is called `MozBlobBuilder` in Firefox, as of version 6, and `WebKitBlob Builder` in Safari Nightly builds and Chrome as of version 8.

In Example 6-1 raw PNG data is passed to a BlobBuilder Object to create a blob which is then turned into a URL. The URL object can be set at the src attribute of an image tag to display in the dom.

*Example 6-1. Creating a Blob URL from raw data*

```
/*global window, $, BlobBuilder, document, XMLHttpRequest */
/*jslint onevar: false, white: false */

function makePNG(pngData)
{
  var BlobBuilder = window.BlobBuilder || window.MozBlobBuilder || window.WebKitBlobBuilder;

  var blob = new BlobBuilder();
  blob.append(pngData);
  var url = blob.getBlob('image/png').createObjectURL();
  return url;
}
```

You can extract a section of the blob with the `slice()` method. Because `slice` has different parameters from the array and string methods of the same name, it has been renamed to `mozSlice()` on Firefox and `webkitSlice()` on Chrome. Whatever name the browser requires, the method returns the extracted data as a new blob. Taking a slice is the only access that the blob API does allows to the raw data of the blob.

If a blob contains data that needs to be loaded as if it were a URL it can be turned into something that can be used as a url with the `createObjectURL()` method. This returns an Object that can be assigned to HTML tag attribute that expects a URL. For example a Blob that contains image data can be assigned to the src attribute of an <img> tag to display the image.

When done with a URL, it is important to deallocate it manually, because Javascript will not be able to determine when to garbage collect this object. To do this, use the `revokeBlobURL()` method. Blob URL's have the same origin as the creating script, so They can be used pretty flexiably in places where the Browser's Same Oragin policy can be a problem. The browser will also revoke all blob URL's when the user navigates away from the page.

# Working With Files

For many years, HTML forms have been able to include a file type as a form element that allows the user to specify a file to upload to the server. The browser does not allow Javascript to set this field, for fear that it could somehow force the upload of a file it shouldn't. However, new JavaScript APIs allow you to read the contents of files from the local system that have been added to the form element by the user.

If you have a form with a file upload field, it will provide a `FileList` object. Each element of the `FileList` object is a `File` object. A file object provides the user with the file's name, MIME type, size, and Last Modified Date. The full path is not exposed in Javascript, but can be seen in Firebug.

The file pointed to by a `File` Object can be read in full by the `FileReader` Object, or in sections by using the `.slice()` method. To upload a very large file, such as a video, it may make sense to chop the file up into smaller parts and upload each part to the server instead of trying to upload a single file that may be a few hundred megabytes. It should also be noted that many servers, by default, limit file upload sizes to a few tens of megabytes.

By using the `FileReader` interface, your program can read the contents of the file. All of these methods return **void** and the data will be delivered after the browser finishes reading the file into memory, with the `onload` handler. This aynchronous operation is important because it can take some time for a very large file to be read into the browser.

The `FileReader` API has four options for reading in data:

`FileReader.readDataAsURL()`
> Transforms a file to a URL so it can be used in the page, the resulting object will contain the full data of the file.

`FileReader.readAsText()`
> Returns the data as a string. By default, the text is encoded as UTF-8, but you can specify a different format.

`FileReader.readAsBinaryString()`
> Returns the data as a binary string, with no attempt to interpret the contents.

`FileReader.readAsArrayBuffer()`
> Returns the data as an ArrayBuffer.

Example 6-2 shows an example of a Drop handler. Drag and Drop find handling will be introduced fully in . For this example the thing to understand is that the function is called by the event handler with a list of files in the form of a `FileList` object. The first file in the list is then displayed by the browser by creating a URL on the fly and adding it to an <img> tag.

*Example 6-2. Appending an image to a document*

```
var el = document.getElementById('dropzone');
el.addEventListener('drop', function (evt) {
    var file = evt.dataTransfer.file[0];
    file.readDataAsURL();
    file.onload = function (img)
    {
        $('div.images').append('<img>').attr({src: img});
    };
});
```

# Uploading Files

Being able to drag a file from the desktop to the browser is a nice trick, but if you can't somehow upload it to the server it looses a lot of its use. Newer versions of the `XMLHttpRequest` interface provide a way to do just that. Using the `FormData` interface, a program can wrap up files and send them to the server. `XHMLHttpRequest` also provides a few callbacks where you can offer feedback to the user. An `onprogress` event returns the number of bytes that have been sent and the total size of the upload, which can let you display a progress bar during large uploads. An `oncomplete???` event is called when the upload is finished.

When you upload files with `XMLHttpRequest`, the server will see the upload in the same interfaces that would be used with a standard `<input type='file' multiple >` form interface. So the server side code for this should be be the same standard code for file uploads that has been in use since the form element was introduced.

Example 6-3 shows an example of code to upload files over Ajax. It uses the `FormData` object to wrap up the files which are then sent to the server.

*Example 6-3. Uploading files with Ajax*

```
function upload(files){
var uploadBlob = function uploadBlob(files, onload) {
    var xhr = new XMLHttpRequest();
    xhr.open('POST', params.url, true);
    xhr.onload = onload;
    xhr.send(files);
};

var formData = new FormData();
files.map(function (file) {
    formData.append(file.fileName, file);
    return file;
});

uploadBlob(files, function (){
    alert("Upload Finished");
```

```
});
}
```

# Drag and Drop

HTML 5 has increased support for drag-and-drop. It is now possible to drag and drop one or more files from the desktop to the browser and give Javascript access to the contents of those files. This could be used, for example, by a file manager to upload files to a server, or by a graphics program to manipulate images. A social network site could allow a user to drag images to your browser, and then crop, scale, rotate, and preview them in the browser before uploading them to the server. This could save server resources by providing it with smaller images.

> The File APIs, for security reasons, will not let you upload a directory structure, but just a list of simple files. If you are building a file manager, you can of course have the user upload an archive file (ZIP, RAR, *tar*, etc.) and have the server expand it. There are libraries to unpack most archive formats in most of the popular server side languages.

To implement drag-and-drop, add a listener to the `drop` event of the DOM element that is the drop target. The event handler you specify will be passed an argument that will have a list of files in it that contains all of the files that have been dragged. Note that this object looks like an array, in that it has numeric keys and a length property. However it is not actually an array, and trying to call `map` or any of the other array operators on it will not work.

> Selenium can not drag a file from the desktop to the browser, so there is no way to automatically test drag-and-drop with Selneium.

# Putting it all together

Example 6-4 To put all of this together a more complete example that will designate an area of the web page as a drop zone. Then when a file is dropped it is packaged with the `FormData` object and then upload it via the `XMLHttpRequest` Object. It also shows a progress indicator to give the user feedback on what is happening.

*Example 6-4. Uploading files*

```
/*global $, FormData,alert, document, XMLHttpRequest */
/*jslint onevar: false, white: false */
```

```
(function () {
    var toArray = function toArray(files) {
        var output = [];
        for (var i = 0, f; f = files[i]; i += 1) {
            output.push(f);
        }
        return output;
    };

    var updateProgress = function (state) {
        var progress = $('progress#progress');
        progress.attr('max', 100);
        if (state === 'start') {
            return progress.fadeIn(500);
        }
        else if (state === 'stop') {
            return progress.fadeOut(500);
        }
        // use the jQuery UI Progress bars
        return progress.attr('value', state);
    };

    var uploadBlob = function uploadBlob(params) {
        var xhr = new XMLHttpRequest();
        xhr.open('POST', params.url, true);
        xhr.onload = params.success;

        // Listen to the upload progress.
        xhr.upload.onprogress = params.progress;

        xhr.send(params.blob);
        return params;
    };


    var fileupload = function fileupload() {

        var el = document.getElementById('dropzone');
        var stopEvent = function (evt) {
            evt.stopPropagation();
            evt.preventDefault();
        };
        el.addEventListener('dragover', stopEvent, false);

        el.addEventListener('drop', function (evt) {
            stopEvent(evt);

            var files = toArray(evt.dataTransfer.files);

            updateProgress('start');
            var packageFiles = function (files) {
                var formData = new FormData();
                files.map(function (file) {
                    formData.append(file.fileName, file);
                    return file;
```

```
                });

                var block = {
                    url: '/upload.php',
                    success: function () {
                        updateProgress('done');
                    },
                    progress: function (evt) {
                        updateProgress(100 * (evt.loaded / evt.total));
                    },
                    blob: formData
                };
                return block;
            };
            uploadBlob(packageFiles(files));
        }, false);
    };
    //run setup
    fileupload();
}());
```

# Filesystem

The idea of a browser allowing Javascript to access the filesystem is enough to send anyone who thinks about security into a panic. There are many things on any user's hard drive that the one would not want the browser to be able to access.

Google Chrome allows Javascript to access a sandboxed file system on the user's computer. If you want to run the `FileSystem` API from inside a web page, Chrome must be started with the `--unlimited-quota-for-files` flag. However, if you are building an app for the Google web store then you can access this API by specifying `unlimitedStorage` in the store manifest file.

While LocalStorage and IndexedDB allow a Javascript program to store objects in a database, the `FileSystem` API is useful for storing large binary objects. For example, if you're building a video app it may be useful to store working images on a filesystem. Other use cases could include Streaming video, games with lots of media assets, image editing, or audio applications. In short, this interface will be a good fit for any application that needs to store a lot of data locally, on a short-term or long-term basis.

> Because the Filesystem API is only supported in Chrome, and only when the user has loaded a trusted app, it is somewhat beyond the scope of this book. A full treatment can be found in the book Using the HTML5 Filesystem API.

# Take it Offline

The Internet may seem to be always on these days but, let's be honest, its not. There are times and places when even the most modern mobile devices are out of range of the network for one reason or another.

Chapter 4 looked at how to have data stored local to the browser so that it does not require network access to use. However if the web page on which the application is hosted is not available, having the data handy will be of no use.

With more and more of the modern application infrastructure moving into the browser, being able access this software at any time has become of critical importance. The problem is that the standard web application assumes that many componenets including JavaScript Sources, HTML, Images, CSS and so forth will be loaded with the web page. In order to be able to use those resources when the user does not have access to the internet requires that copies of those files be stored locally, and used by the browser when needed. HTML 5 provides the ability for a programmer give the browser a manifest of files that should be loaded and saved. The browser will be able to access these files even when there is no network connection to the server.

The files listed in the manifest will also be loaded from the local disk even if the browser is online. Thus giving the end user the experince of the ultimate Content Delivery Network.

As long as the browser is online when a page is loaded it will check the manifest file with the server. If the manifest file has changed the browser will attempt to redownload all the files listed for download in the manifest. Once all the files in the manifest have been downloaded the browser will update the file cache to show the new files.

## Introduction to the Manifest File

The ability to access files while offline was one of the features introduced by Google in Gears. The user provided a manifest as a JSON file, which then directed the browser to load other required files offline. When the browser next visited that page, the files

would be loaded from the local disk instead of from the network. When the version field of the manifest file was updated, Gears would check all the files in the manifest for updates.

The HTML 5 manifest is similar in idea but somewhat different in implementation. One nice thing about it is that you can implement a manifest in an application without using any JavaScript code, which gears did require. To do so, add the `manifest` attribute containing the name of the manifest file to the document's `<html>` tag (Example 7-1). The manifest file must be served with the mime type `text/cache-manifest`. This can be done via the web server configuration files. For the Apache web server add the following line to the config file. For other web servers, consult the server's documentation. The file name does not matter as long as the file has the correct mime type, but `cache.man ifest` seems to be a good default choice.

*Example 7-1. HTML manifest declaration*

```
<!DOCTYPE HTML>
<html manifest="/cache.manifest">
<body>
       ...
</body>
</html>
```

To setup the mime type with the Apache web server add a line like this to the Apache config files. For other web servers see the documentation for the relivant server.

```
    AddType text/cache-manifest  .manifest
```

## Structure of the Manifest file

The format of the manifest file is in fact pretty simple. The first line must be just the words `CACHE MANIFEST`. After that comes a list of files, one per line, to include in the manifest (Example 7-2). Comments can marked with the pound (#) character.

The manifest will cache HTTP GET requests, while POST, PUT, and DELETE will still go to the network. If the page has an active manifest file all GET requests will be directed to local storage. But for some files, offline access does not make sense. These can include various server resources such as Ajax calls, or collection of documents that could get so large as to overflow the cache area. These files can be included in a `NETWORK` section of the manifest. Any URLs in the `NETWORK` section will bypass the cache and load directly from the server. The HTML5 manifest requires that any non included files be explictly opted out of the manifest.

In other cases, you may wish to provide different content depending on whether the user is offline or online. The manifest provides a `FALLBACK` section for such resources. The user will be shown different content, depending on whether the browser has a

connection to the internet or not. On each line of the `FALLBACK` section, the first file is loaded from the server when a connection is available, and the second file is loaded locally when the connection is not available.

Both the `NETWORK` and `FALLBACK` sections list file paterns, not specific files. So it is possible to list entire directories or URL paths here, as well as file types such as images (for instance, `*.jpg`).

*Example 7-2. Manifest File*

```
CACHE MANIFEST
# 11 October 2010
/index.php
/js/jquery.js
/css/style.css
/images/logo.png

NETWORK:
/request.php

FALLBACK:
/about.html /offline-about.html
```

## Updates to the manifest

The browser will update the files in the manifest whenever the manifest file itself changes. There are several ways to handle this. It is possible to add a version number in a comment in the file. If the project is making use of a version controll system like Subversion, you can use the version number tag for this.

The problem with using a version number from a version control system is that it requires a programmer to remember to update that file every time any file in the system changes. It would be much better to create an automated system that updates the manifest file whenever a file listed in it changes, and run that script as part of a deployment procedure.

For instance, you could write a script that checks all the files in the manifest for changes and then change the manifest file itself when one of the files changes. A simple way to do this is to write a script that loops over all the files in the manifest, then does an MD5 checksum on each one, then puts a final checksum into the manifest file. This will ensure that any changes will cause the manifest file to update.

This script is probably too slow to run from the web server, as running it hundreds of times a second would be overkill. However, it can be efficiently run in the development environment, one option would be to have it run from an editor when a file is saved. Or run it as part of the check-in process for a version control system.

In Example 7-3 we parse the manifest file and do a few things with it. The program uses the Symfony Yaml Library to load a list of files to use as a manifest. As a bonus, the program first checks that no file has been included more than once. It also checks that every file exists, because missing files will break the manifest. By adding each file's MD5 as a comment after the filename, the script makes sure that any updated file will cause a manifest change so that the browser will update its content. It takes a datafile in the format of Example 7-5. Example 7-3 will output a manifest file with the MD5 hash as a comment in the file, as in Example 7-4.

*Example 7-3. Automatically updating a manifest file*

```php
<?php

header('Content-Type: text/cache-manifest');
echo ("CACHE MANIFEST\n");
$files = sfYaml::load('manifest.yml');
$hashes = '';
$files = unique($files);

foreach($files->cache as $file)
  {
    if(file_exists($file))
      {
    echo $file."\n";
    $hashes .=md5_file($file);
      }
  }
echo "\nNETWORK:\n"
foreach ($files->network as $file)
{
  echo $file. "\n";
}

echo "\nFALLBACK:\n"
foreach ($files->fallback as $file)
{
  echo $file. "\n";
}

echo "# HASH: ". md5($hashes) . "\n";
```

*Example 7-4. Manfiest with MD5 Hash*

```
CACHE MANIFEST
index.html
css/style.js
js/jquery.js
js/myscript.js

NETWORK:
network/file

FALLBACK:
```
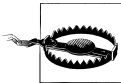
```
/avatars/ /offline-avatars/offline.png

#HASH: 090c7e8fe42c16777fba844f835e839b
```

*Example 7-5. The data for Example 7-3*

```
files:
  - index.html
  - css/style.css
  - js/jquery.js
  - js/myscript.js

network:
  - network/file

faillback:
  - /avatars/ /offline-avatars/offline.png
```

> The manifest is not always very good about updating when you think it should. Even with a new version of a manifest, it can often take some time to update the content in the browser. Unless you set the cache control headers, the browser will not download the manifest again until several hours after it was last downloaded. Make sure that the cache control headers don't cause the browser to only download the file, say, every five years, or use the ETag header. Or better yet have the server set a no cache header. Be sure to test well.

# Events

When the browser loads a page with a manifest file, it will fire a `checking` event on the `window.applicationCache` object. This event will fire whether or not the page has been visited before.

If the cache has not been seen before, the browser will fire a `downloading` event, and start to download the files. This event will also fire if the manifest file has changed. If the manifest has not changed the browser will fire a `noupdate` event.

As the browser downloads the files, it fires a series of `progress` events. These can be used if you wish to provide some form of feedback to the user to let her know that software is downloading.

Once all the files have downloaded the `cached` event is fired.

If anything goes wrong, the browser will fire the `error` event. This can be the result of the HTML page having a problem, the manifest being defective or failing to download, or any resource listed in the manifest failing to download. If a single file is missing from the manifest, it won't download any of the files in the manifest. In the case of a manifest

that is being changes if there is a bad link the old versions of the file will be retained. In the case of a new manifest the browser may just ignore the manifest. However it is possible that not all browsers or browser versions will be conistent on this point. Having an automatic test to validate all the URLS in a manifest is a good idea. This can be a very hard error to cache as there may be very little visible evidence of what went wrong. Catching the error object and presenting it to the user would be a good idea, as would some form of automatic testing for bad links.

In Google Chrome, the developer tools can show a list of files in the manifest (Figure 7-1). Under the Storage tab, the Application Cache item will show the status of various items.
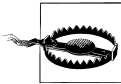
> It is a good idea during development to turn off the manifest file, and enable it only once the project is ready to go live. Using the cache can make it very hard to develop the application if changes don't appear quickly.

# Debugging Manifest Files

Manifest files provide a particular debugging challenge. It can be the source of several special classes of bugs.

The first and most obvious bug is to include missing files in the manifest. If a file is included in the page and it is not in the manifest, it will not be loaded by the page, in the same way a missing file on the server will not be downloaded.

Many Selenium tests will not explicity test for correct styles and the presense of images, so it is quite possible an application missing a CSS file or image will still work to the extent that it is normally tested in Selenium. In an application that includes resources from outside web servers, those must also be whitelisted in the manifest file.

A further complication comes in some browsers, including Firefox, that make the manifest an opt-in feature. So a Selenium test may not opt into it, which would make the entire test moot. In order to test this in Firefox, it will be nessary to set up a Firefox profile in which the application cache is on by default. To do this:

- Quit Firefox completely.
- Start up Firefox from a command line with the `-profileManager` switch. This will give a dialog like Figure 7-2. Save the custom profile.
- Restart Firefox. Go to the Firefox options menu, select the Advanced tab and under that the Network tab (Figure 7-3), and turn off the "Tell me when a website asks to store data for offline use" option.

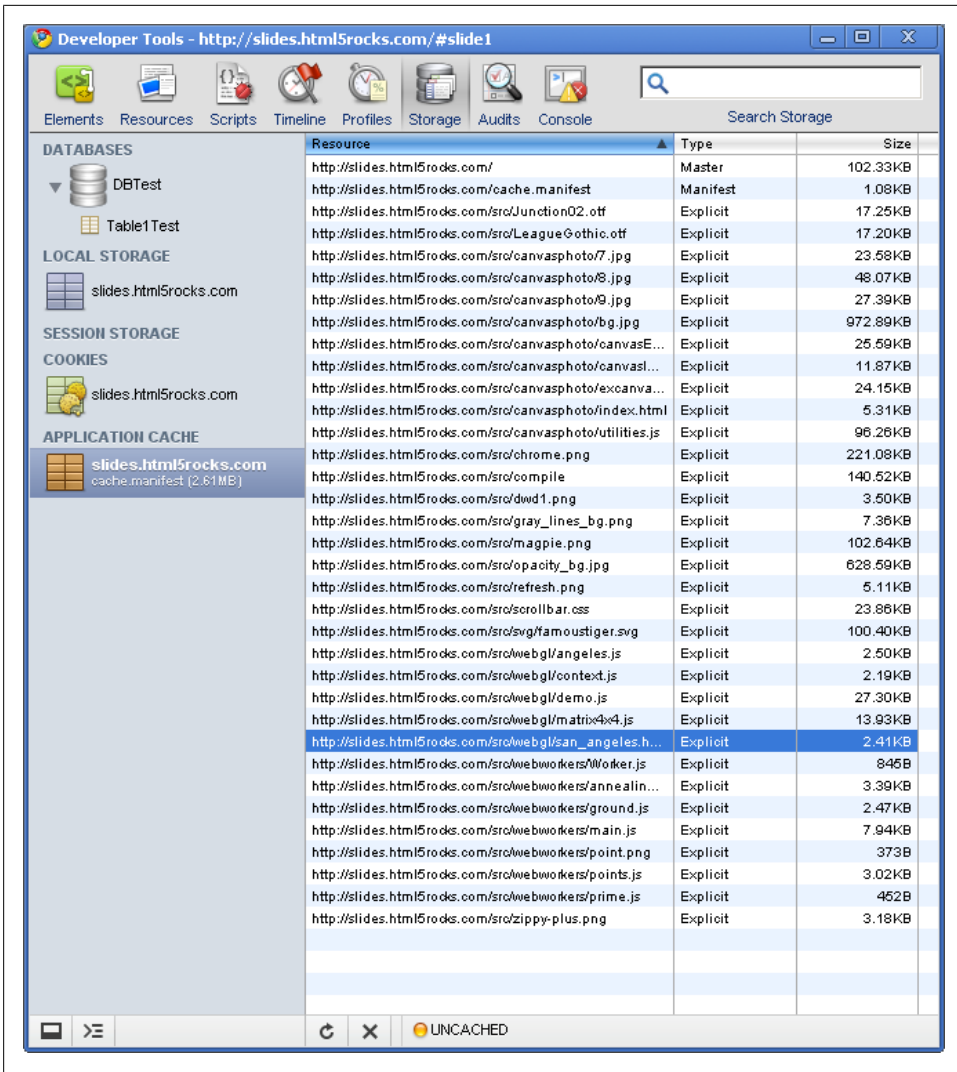Now, when starting up the selenium RC server, use an option like this:

*Figure 7-1. Chrome storage viewer*

```
java -jar selenium-server.jar -firefoxProfileTemplate
```

For full details on Firefox profiles, see *http://support.mozilla.com/en-US/kb/Managing +profiles*.

A second class of problems can occur when the manifest is updated and the browser does not reflect the update. Normally, it will take a minute or two after loading a page for the browser to update the file cache, and the browser will not check the cache until the page is loaded. So if the server is updated, the browser will not have the new version
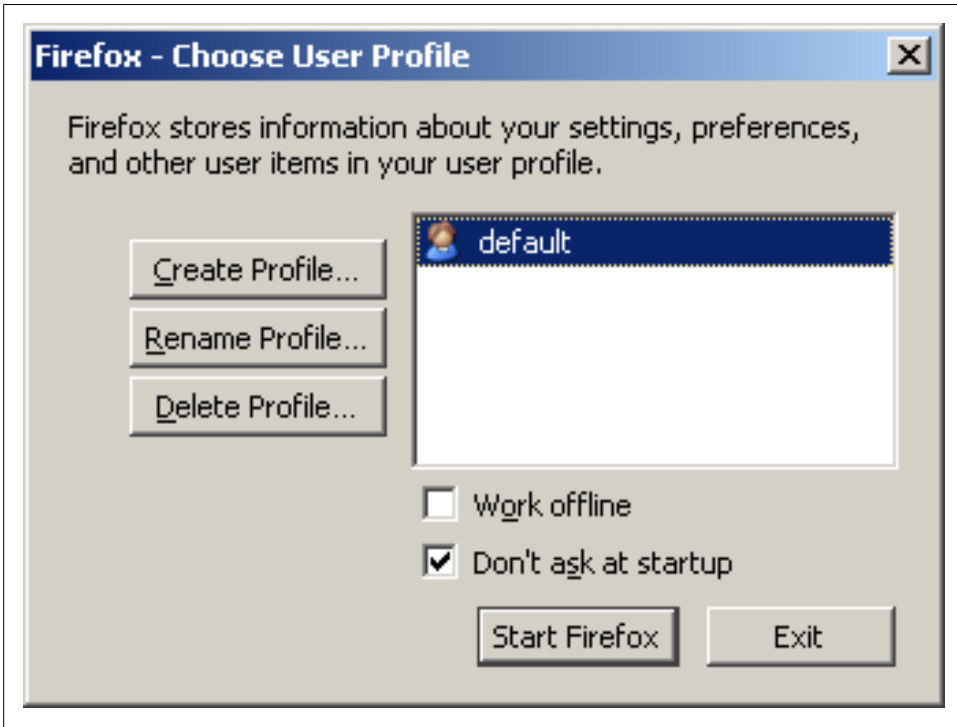
*Figure 7-2. Firefox Custom Profile Dialog*

until the user visits the page. This can cause problems if there has been an update on the server that will cause the application in the browser to fail, such as a change in how the Ajax protocall.

When the user visits the page (assuming of course that the browser is online), the browser will fetch the manifest file from the server. However, if the manifest file has a cache control header set on it, the browser may not check for a new version of the manifest. For example, if the file has a header that says that the browser should check for updates only once a year (as is sometimes common on web servers), the browser will not reload the manifest file. So it is very important to ensure that the manifest file itself is not cached by the browser, or if it is cached it is done only via an ETag.

The browser can always prevent cacheing of the manifest file by giving the URL with a query string attached as in `cache.manifest?load=1`. If the manifest file is a static text file the query string will be ignored but the browser will not know that and will force the server to send a fresh copy.

Different web browsers, and even different versions of a single browser, may update the manifest file somewhat differently. So it is very important to test any application using a manifest file very carefully across different browsers and browser versions.
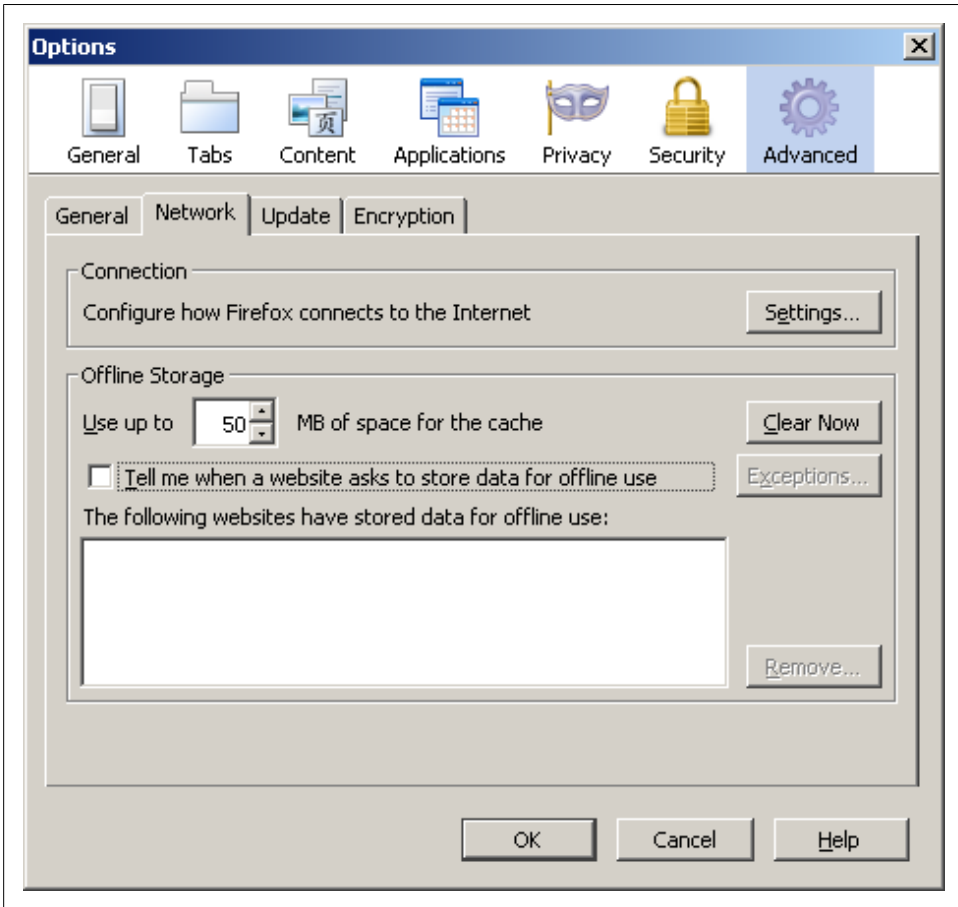
*Figure 7-3. Firefox options*

# CHAPTER 8

# Splitting Up Work Through Web Workers

Javascript has, since its inception, run in a single thread. With small applications, this was practical but now with larger and larger applications it has limits. As more and more Javascript is run, the application will start to block waiting for code to finish.

Javascript runs code from an event loop that takes events off a queue of all the events that have happened in the browser. Whenever the Javascript runtime is idle, it will take the first event off the queue and run the handler that goes with that event (Figure 8-1). As long as those handlers run quickly, this makes for a responsive user experience.

In the past few years, the competition between browsers has in part revolved around speed of Javascript. In Chrome and Firefox, Javascript can now run go as much as one hundred times faster as it did back in the days of IE 6. Because of this, it is possible to squeeze more into the Event loop.

Thankfully, most things that Javascript has to do are fast. They tend to be on the order of manipulating some data and passing it into the DOM or making an Ajax call. So the model in Figure 8-1 works pretty well. For things that would take longer than a fraction of a second to compute, there are a number of tricks. The main one is to break the computation up into small steps and run each one as an independent job on the queue. Each step ends with a call to the next step after a short delay, say 1/100 of a second. This prevents the task from locking up the event queue. But it still fundamentally unsatisfactory, as puts the work of the task scheduler onto the programmer.

If the time steps are too small, computation can still clog up the event queue and cause other task to lag behind. So things will still happen, but the user will feel the lag as the system fails respond right away to clicks and other user-visible activities. On the other hand, if the steps between actions are too large, the computation will take a very long time to complete causing the user to wait for her results.
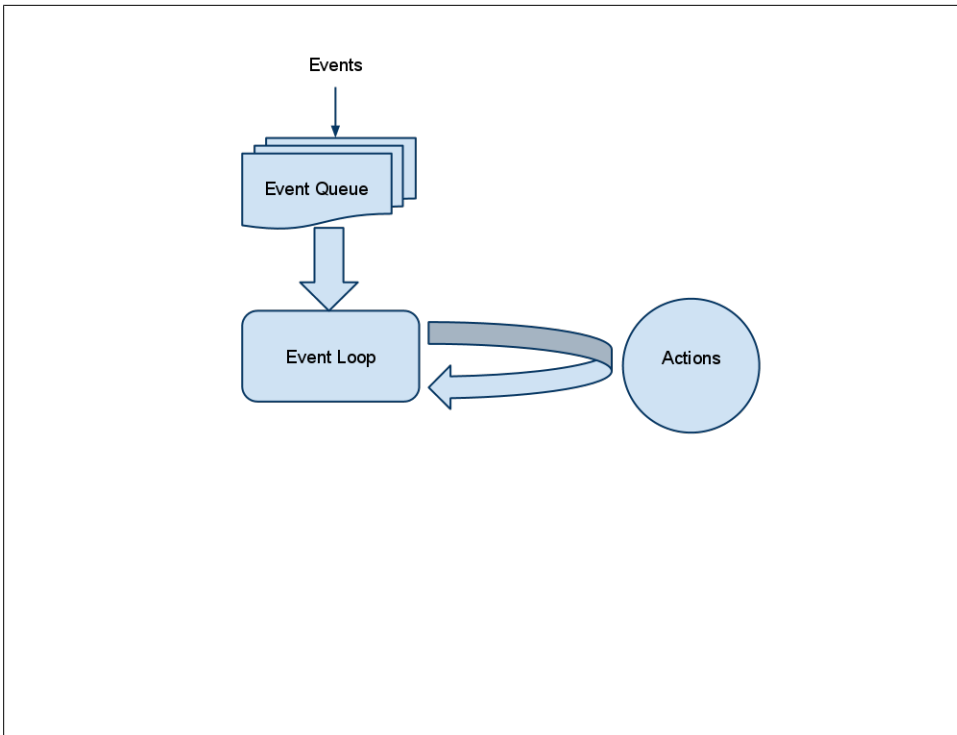
*Figure 8-1. Event Loop*

Google Gears created the idea of the "Worker Pool," which has turned into the HTML5 Web Worker. The interfaces are somewhat different, but the basic ideas are the same. A worker is a separate Javascript process that can perform computations and pass messages back and forth with the main process and other workers. A Web Worker differs from a thread in Java or Python in one key aspect of design: *there is no shared state*. The workers and the main Javascript instance can communicate only by passing messages.

That one difference leads to a number of key programming practices, mostly simpler than thread programming. Web Workers have no need for mutexes, locks, or synchronization. Deadlocks and race conditions can't occur. This also means you can use the huge number of Javascript packages out there without worry whether they are thread safe. The only changes to the browser's Javascript environment are a few new methods and events.

Each worker (including the main window) maintains an independent event loop. Whenever there is no code running, Javascript runtime returns to this event loop. At this point the runtime will take the first message out of the queue. If there are no events in the queue, it will wait until an event arrives and then handle it. If some piece of code is running for a long time, no events will be handled until that piece of code is finished.

In the main window, this will result in the browser user interface locking up. (Some browsers will offer to let you stop Javascript at this point.) In a worker, a long task will keep the worker from accepting any new events. However the main window, and any other workers, will continue to be responsive.

This design choice does, however, place some restrictions on the worker processes themselves. First, workers do not have access to the DOM. This also means that a worker can't use the Firebug console interface, as Firebug communicates with Javascript by way of the DOM. Finally, Javascript debuggers cannot access workers, so there is no way to step through code or do any of the other things that would normally be done in the debugger.

# Web Worker Use Cases

The types applications traditionally run on the Web, and the limitations of the web browser environment, limited the computational needs that would call for a Web Worker. Until recently, most web applications manipulated small amounts of data that mostly consisted of text and numbers. In these cases, a Web Worker type of construct is of limited use. Now Javascript is asked to do a lot more, and many common situations can benefit from spawning new tasks.

## Graphics

The HTML5 `<svg>` and `<canvas>` tags allow Javascript to manipulate images, a potentially heavy task computationally Although web browsers have been able to display images since the release of the Mosaic browser around 1993, the browsers couldn't manipulate those images. If a web programmer wanted to take an image and distort it, overlay it transparently, and so forth, it could not be done in the browser. In the `<img>` tag, all the browser could do is substitute a different image by changing the `src` attribute, or change the displayed size of the image. However the browser had no way of knowing what the image was or accessing the raw data that made up the image.

The recently added `<canvas>` tag makes it possible to import an existing image into a canvas and export the raw data back into Javascript for processing, as long as the image was loaded from the same server as the page it is on. It is also possible to export a frame from a video in the HTML5 `<video>` tag. [*]

Once the data has been extracted from a graphic, you can pass it to a worker for post processing. This could be useful for doing anything from cleaning up an image to doing a Fourier transform on a scintific data set. Canvas makes it possible to build complex image editing through various filters written in Javascript, which should often use Web Workers for better performance.

---

[*] See *HTML5 Canvas* by Steve Fulton and Jeff Fulton (O'Reilly) for more information on the graphics in HTML5.

## Maps

In addition to graphics, Javascript has APIs now for handing map data. Being able to import a map from the Internet and find out the user's current location via geolocation allows a wide range of web application services.

Suppose you build a route finder into a mobile browser. It would be very nice to be able to take your phone and tell it you wish to go to "#14 King George St, Tel Aviv" and have the browser figure out where you are, direct you to the nearest bus stop, and tell you that you should take the number 82 bus to get there from the diamond district in Ramat Gan.

An even more complex version of that software might check traffic to tell you that a different bus might take a more round-about route and leave you a block from your destination, but probably run faster by missing a major traffic snarl.

# Using Web Workers

To start up a web worker, create a new `Worker` object and pass, as the parameter to the call, the file that contains the code (Example 8-1). This will create a worker from the source file.

*Example 8-1. Worker Example*

```
$(document).ready(function (){
    var worker = new Worker('worker.js');
    worker.onmessage = function (event){
        console.info(event);
    };
    worker.postMessage("World" );
});
```

The browser will load the worker, run any code that is not in an event handler, and then launch the event loop to wait for events. As in a normal window environment any code that is not inside an event handler will be run at this point. The main event to be concerned with is the `message` event, which is how you send data to the worker. The data from the main thread is held in the `event.data` field. The worker should process this data and then send it back with the `self.postMessage()` method.

## The Worker Environment

Web workers run in a pretty minimal environment. Many of the familiar objects and interfaces of Javascript in the browser are missing, including the DOM, the `document` object, and the `window` object.

In addition to the standard ECMA script objects like String, Array, and Date, the following objects and interfaces are available to the web worker:

- The `navigator` object, which contains four properties: `appName`, `appVersion`, `user Agent`. and `platform`
- The `location` object, with all properties read-only.
- The `self` object, which is the worker object.
- The `importScripts()` method
- The XMLHttpRequest interface for doing Ajax methods
- `setTimeout()` and `setInterval()`
- The `close()` method, which ends the worker process

ECMAscript 5 JSON interfaces can also be used as they are part of the langauge not the Browser enviroment. Furthermore, the worker can import library scripts from the server with the `importScripts()` method. This method takes a list of one or more files, which are then loaded. This has the same effect as using a <script> tag in the main user interface thread. Unlike most methods in Javascript, `importScripts` is blocking. The function will not return until all the listed scripts have been loaded. `importScripts` will execute the loaded files in the order in which they were specified to the command.

While localStorage and sessionStorage are not accessible from the web worker, Index-edDB databases are.Chapter 5 on page 59 In addition the IndexedDB Spec says that the blocking forms of calls can be used in a web worker (but not in the main window). So in the case of a worker using IndexedDB to manipulate data it would make sense to load the new data into the database and then send an "updated" message to the main window or other workers to let them know to take any needed actions.

## Worker Communication

The main event that concerns a worker is the `message` event, which is sent to the worker from the `postMessage` method in the main Javascript context to pass information. In Firefox, it is possible to pass complex Javascript objects. However, Chrome and Safari support only simple data, such as strings, Booleans, and numbers. It is good practice to encode all data into JSON before sending it to a web worker.

The worker can send data back to the main thread via the same `postMessage` method. Receive it back in the main thread via the `worker.onmessage` handler.

The model for worker communication is that the main task creates the worker, after which they pass messages back and forth as shown in Figure 8-2.

# Web Worker Fractal Example

Example 8-1 is the "Hello World" of Web Workers. A more complex example is called for. Figure 8-3 shows a visual representation of a Mandelbrot set computed in a web worker. Here the worker and the main thread split up the work to draw the fractal. The
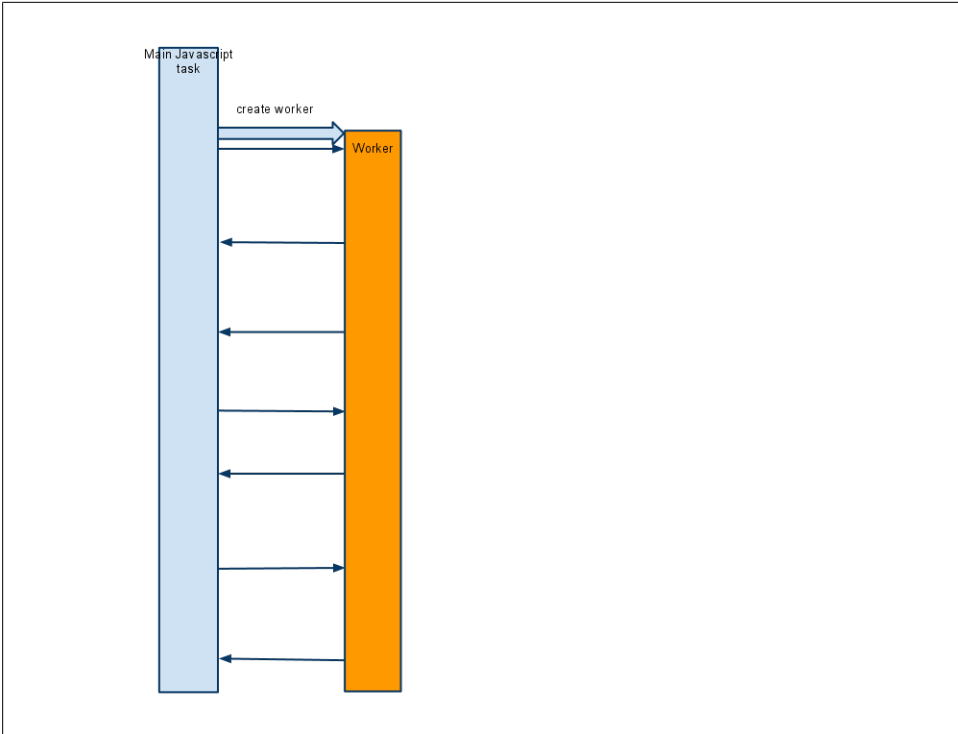
*Figure 8-2. Worker Communication*
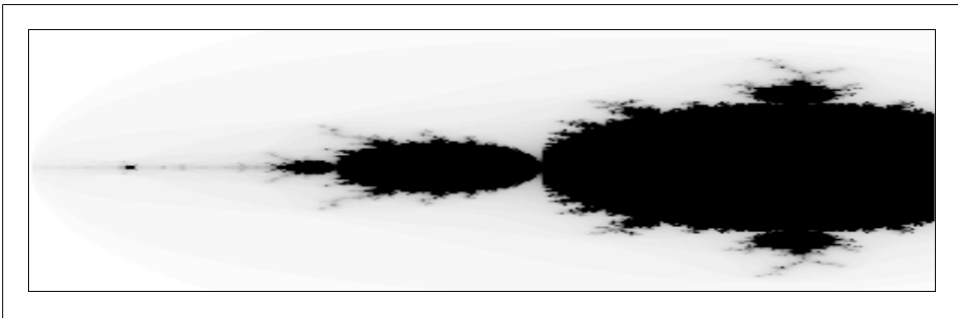


*Figure 8-3. Mandelbrot Example*

worker does the actual work of computing the Mandelbrot set, while the front-end script takes that raw data and displays it in the canvas.

The front-end script (Example 8-2) sets up the canvas element and scales it to fit in the page. Then it creates an object to wrap the worker interface. The wrapper object creates the worker in the wrapper's `run()` method, passing to the worker a parameter block that tells it what chunk of the Mandelbrot set to compute.

The `draw` method takes the data, scales it to fit onto the canvas, sets a color, and then draws the pixel.

> The HTML Canvas does not have a draw pixel command, so to draw a pixel we must draw a square of size 1 and offset it by half a pixel from the spot where we want it to show up. So to draw a pixel at 20,20 it should be at 19.5,19.5 to 20.5, 20.5. The locations on the canvas grid are not the pixels on the screen but the points between them.

The `onmessage` handler then waits for events to be sent from the worker. If the event type is `draw`, the handler calls the method to draw the new data into the canvas. If the event is `log`, it is logged to the Javascript console via `console.info()`. This provides a very simple method to log status information from a worker.

The `startWorker` method aliases the `this` to a local variable named `that`. This is because `this` is not lexically scoped like other JavaScript variables. To allow the inner function to have access to that object, which it will need to draw a pixel, it is nessary to alias it to a lexically scoped variable. By convention that variable is often called `that`.

*Example 8-2. Mandelbrot Front End*

```
var drawMandelSet = function drawMandelSet(){

    var mandelPanel = $('body');

    var width = mandelPanel.innerWidth();
    var height = mandelPanel.innerHeight();

    var range = [{
        x: -2,
        y: -1.4
    }, {
        x: 5,
        y: 1.4
    }];

    $('canvas#fractal').height(height + 100);
    $('canvas#fractal').width(width  - 50);
    var left = 0;
    var top = 0;

    var canvas = $("canvas#fractal")[0];
    var ctx = canvas.getContext("2d");
    var params = {
        range: range,
        startx: 0.0,
        starty: 0.0,
        width: width,
        height: height
```

```
    };
    var y_array = [];

    var worker = {
        params: params,

        draw: function draw(data){
            data.forEach(function d(point){
                if (this.axis.x[point.drawLoc.x] === undefined) {
                    this.axis.x[point.drawLoc.x] = point.point.x;
                }
                if (this.axis.y[height - point.drawLoc.y] === undefined) {
                    this.axis.y[height - point.drawLoc.y] = point.point.y;
                }

                ctx.fillStyle = pickColor(point.escapeValue);
                ctx.fillRect(point.drawLoc.x + 0.5, height - point.drawLoc.y + 0.5, 1, 1);
            }, this);
        },

        axis: {
            x: [],
            y: [],
            find: function(x, y){
                return new Complex(this.x[x], this.y[y]);
            },

            reset: function(){
                this.x = [], this.y = [];
            }
        },
        myWorker: false,

        run: function startWorker(params){
            this.myWorker = new Worker("js/worker.js");

            var that = this;
            this.myWorker.postMessage(JSON.stringify(params));

            this.myWorker.onmessage = function(event){

                var data = JSON.parse(event.data);
                if (data.type === 'draw') {
                    that.draw(JSON.parse(data.data));
                }
                else
                    if (event.data.type === 'log') {
                        console.info(event);
                    }
            };
        }
    };

    worker.run(params);
    return worker;
```

```
};

$(document).ready(drawMandelSet);

Function.prototype.createDelegate = function createDelegate(scope){
    var fn = this;
    return function(){
        fn.call(scope, arguments);
    };
};

function pickColor(escapeValue){
    if (escapeValue === Complex.prototype.max_iteration) {
        return "black";
    }

    var tone = 255 - escapeValue * 10;
    var colorCss = "rgb({r},{g},{b})".populate({
        r: tone,
        g: tone,
        b: tone
    });
    return colorCss;
}

String.prototype.populate = function populate(params) {
    var str = this.replace(/\{\w+\}/g, function stringFormatInner(word) {
        return params[word.substr(1, word.length - 2)];
    });
    return str;
};
```

The actual worker itself is very simple Example 8-3 It just loads up a few other files and then waits for a message to be sent from the User Interface. When it gets one, it starts the computation.

*Example 8-3. Mandelbrot Computation*

```
importScripts('function.js','json2.js', 'complex.js','computeMandelbrot.js', 'buildMaster.js');

onmessage = function(event){
    var data = typeof event.data === 'string'? JSON.parse(event.data) :  event.data;
    buildMaster(data);
};
```

The build function (Example 8-4) loops over the grid of points for the Mandelbrot set, computing the escape value for each point(Example 8-5. After every 200 points, the build function sends the results of its computation back to the main thread for drawing, and then zeroes out its internal buffer of computed points. This way, instead of waiting for the entire grid to be drawn at once, the user sees the image build progressively.

*Example 8-4. Mandelbrot Build*

```
var chunkSize = 200;
function buildMaster(data){

    var range = data.range;
    var width = data.width;
    var height = data.height;
    var startx = data.startx;
    var starty = data.starty;
    var dx = (range[1].x - range[0].x) / width;
    var dy = (range[1].y - range[0].y) / height;


    function send(line){
        var lineData = JSON.stringify(line.map(function makeReturnData(point){
            return {
                drawLoc: point.drawLoc,
                point: point.point,
                escapeValue: point.point.mandelbrot()
            };
        }));

        var json = JSON.stringify({
            type: 'draw',
            data: lineData
        });
        postMessage(json);
    };


    function xIter(x, maxX, drawX){
        var line = [];
        var drawY = starty;
        var y = range[0].y;
        var maxY = range[1].y;

        while (y < maxY) {
            if (line.length % chunkSize === chunkSize - 1) {
                send(line);
                line = [];

            }
            var pt = {
                point: new Complex(x, y),
                drawLoc: {
                    x: drawX,
                    y: drawY
                }
            };
            line.push(pt);
            y += dy;
            drawY += 1;
        }
        send(line);
        if (x < maxX && drawX < width) {
```

```
        xIter.defer(1, this, [x + dx, maxX, drawX + 1]);
        }
    }

    xIter(range[0].x, range[1].x, startx);

}
```

*Example 8-5. Mandelbrot Computation*

```
Complex.prototype.max_iteration = 255 * 2;
Complex.prototype.mandelbrot = function(){

    var x0 = this.x;
    var y0 = this.y;
    var x = x0;
    var y = y0;
    var count;
    var x_, y_;
    var max_iteration = this.max_iteration;
    function inSet(x, y){
        return x * x + y * y < 4;
    }
    count = 0;
    while (count < max_iteration && inSet(x, y)) {
        x_ = x * x - y * y + x0;
        y_ = 2 * x * y + y0;
        count += 1;
        x = x_;
        y = y_;
    }

    return count;
};
```

The final part of this application is the actual mathematical computation of the Man‐delbrot set shown in Example 8-5. This function is done as a while loop instead of a pure function as in "Functional Programming" on page 12, because Javascript does not support tail recursion. Doing this as a recursive function would be more elegant, but would risk causing a stack overflow.

While the worker is doing the calculation of the Mandelbrot set, its main event look is blocked. So It is not possible for the UI process to send it a new computation task, or more correctly stated, the worker will not accept the new task until the current task is finished.

To interrupt or change a worker's behavior—for instance, to let the user in the user interface thread select which area of the Mandelbrot set to draw and then request that the worker draw that area—there are a few ways to go about it.

The simplest method would be to kill the worker and create a new one. This has the advantage that the new worker starts off in a clean state and there can be nothing left over from the prior runs. On the other hand, it also means that the worker has to load all the scripts and data from scratch. So if the worker has a long start-up time, this is probably not the best approach.

The second method is a little more complex: manage the task queue manually through your program. Have a data structure in the main thread or a worker that keeps a list of blocks of data to compute. When a worker needs a task, it can send a message to that queue object and have a task sent to it. This creates more complexity but has several advantages. First, the worker does not need to be restarted when the application needs it to do something different. Secondly it allows the use of multiple workers. Each worker can query the queue manager when it needs the next part of the problem.

You could also have the master task send a large number of events to the worker in sequence, However, this has the problem that there is no way from Javascript to clear the event queue. So having a job queue that can be managed seems to be the best approach. We'll explore this solution in the following section.

## Using Multiple Web Workers

There is no requirement that an application restrict itself to one Web Worker. Javascript is quite happy to let you start up a reasonable number of workers. Of course, this makes sense only if the problem can be easily partitioned into several workers, but many problems can be.

Each worker is an independent construction, so it is possible to create several workers from the same source code, or to create several workers that work independently.

Workers are a reasonably heavy construct in Javascript, so it is probably a bad idea to create more than, say, ten workers on a given task. However, the optimal number is probably dependent on the user's browser and hardware as well as the task to be performed.

## Testing and Debugging Web Workers AKA SHOOT ME NOW!

Over the last ten years the tools for Javascript debugging have gotten quite good. Firebug and Chrome Developer Tools both are first-rate debugging tools that can be used for testing Javascript applications. Unfortunately, neither can access code running in a Web Worker. So you can't set break points or step through your code in a worker. Nor do workers show up in the list of loaded scripts that appear in the respective script tags of Firebug and Chrome. Nor can Selenium or QUnit tests directly test worker code running in a web worker.

Errors in a worker are reported back to the console in Firefox and Chrome. Of course, in many cases knowing which line and file the error occurred at does not help all that much, as the actual bug was somewhere else.

Chrome does provide the programmer a method for debugging web workers. The Chrome Developer Tools script panel contains a Web Workers check box. This option causes Chrome to simulate a worker using an iframe.

# A pattern for degredation

Being able to use web workers to pull complex tasks out of the user's browser task offers great power for the programmer. Firefox has supported wokers since version 3.5 and Chrome has supported wokers since version 4.0. Similarly Safari and Opera have supported workers for some time. However as of now Microsoft Internet Exporer does not support web workers, (though it may appear in IE version 10, nor does Safari on iOS so it is not possible to support workers on the iPad/iPod/iPhone platform.

What would be ideal would be a library that would enable a programmer to abstract out the code to be run into a function or module and a runner that would run that code in a web worker if avalable and via a `setTimeout` method. Furthermore it would be able to provide a common set of interfaces that could be used for the various interactions such as posting a message back to the main application.

Such a library should always use feature detection rather than browser detection to figure out which version of the code to run. While right now a given browser may or may not support web workers in the future that will change and a library needs to be able to work with those changes.

The actual function to do the work in this pattern will be called repeatedly with the run state as a parameter, it should do whatever processing it needs to do and return a modified state parameter which will be used to call it again until it finishes its job and calls the `stop()` method or is otherwise interupted. The run function ( Example 8-6) should be treated as a pure function, it should only process its inputs and return a value but not effect any change in global state as depending on if is running as a worker or not it will have a different set of interfaces avalable to it.

*Example 8-6. Run*

```
var run = function run(state)
{

    // do something to modify the state
    return state;
}
```

When running in a worker (Example 8-7) the run function can be run from inside a standard loop. The system is setup via a `postMessage` call with some initial parameters

which are then passed as the inital state to the run method. That method will be re-peatedly called by the while loop until it calls the stop function, at which point the state will be posted back to the main message.

*Example 8-7. Run*

```
//TODO setup code here
var runner = {
    stop: false,
    postMessage: function (message){
        // fill this in
    },

    stop: function (){
    this.stop = true;
    },
    error: function(error){
      this.stop = true;
    },

    setup: function(run)
    {
        this.run  = run;
        this.execute();
    },

    execute: function ()
    {
        var state;
    while(!this.stop)
        {
        state = this.run.apply(this, [state]);
        }
    this.postMessage(state);
    }
};
```

When Web Workers are not avalable then the run function must be presented with the same interface. (Example 8-8) Once again it will call the run function with a `state` parameter which should be returned. However as this is not a web worker it will then call the `window.setTimeout()` method to deplay the next itteration by some amount of time and call the function again.

Communications between the simulated web worker and the main body of the code is also somewhat different. As there is no postMessage method with a callback then the runner must simulate it by presenting a mechanism to register a callback that can take the same parameters as the WebWorker's `onmessage` handler.

*Example 8-8. Run*

```
var runner = {
    stop: false,
    // over ride this function
    onmessage: function(){},
    postMessage: function (message){
      this.onmessage(message);
    },

    stop: function (){
    this.stop = true;
    },
    error: function(error){
      this.stop = true;
    },

    setup: function(run, state)
    {
      this.run   = run;
      this.state = state;
      this.execute();
    },

    execute: function ()
    {
    var that = this;

    setTimeout(function runItterator(){
        that.state = that.run.apply(that, [that.state]);
        if(that.stop)
        {
            that.postMessage(that.state);
        }
        else
        {
            that.execute();
        }
    }, 250);
    }
};
```

This model of how to make code portable between a web worker and reguarl JavaScript is presented as a model and not a full solution. It is missing some features such as loading code. It is also missing a way to call an asycronous method such as an ajax call and resume processing when done. This would be nessary as while in general Web-Workers are designed for processor intensive work there will be times when access to an Ajax Call or indexedb makes sense.

# Libraries for Web Workers

When programming Javascript in the main thread, programmers use a library such as jQuery to improve the API and to hide differences between browsers. For use with Web Workers, there is a jQuery extension called jQuery Hive that provides much of this

functionality. Hive includes the PollenJS library in the main Javascript thread. The library includes interfaces to create workers.

Hive will also encode and decode messages between the main thread and worker if needed. In some browsers (notably Firefox), complex data can be sent over the `post Message` interface. However, in some versions of Chrome and Safari, `postMessage` will handle only a string or other simple data.

Hive also includes a subset of the jQuery API in the worker itself. The most important methods in the Hive API are `$.get()` and `$.post()`, which mirror the APIs in jQuery. If a worker needs to access the server via Ajax, for instance, using Hive will make your life much easier.

Hive also includes access to a persistent storage interface via `$.storage`. To set a value, use `$.storage(name, value)`. Calling `$.storage(name)` without the second value paramater will return the existing value, if set.

Also included in Hive are `$.decode()` and `$.encode()`, which can be used to encode or decode JSON messages.

# Web Sockets

HTTP is a request and response protocal. It was designed to request files and still operates around the ideas of file requests. For the type of application that has to load data and then save it later this works pretty well.

However for an application that needs real time data from the server this works quite poorly. Many classes of applications require real time, or semi-real time access to the server. Applications such as chat, or those that share data in real time like many of the Google Office applications really need a way for the server to push data to the browser when things happen on the server. There are a few ways to do this with HTTP, but none of them really work well.

Some applications, such as gmail, simply make a large sequence of http requests, more than one a second Figure 9-1. This has a lot of overhead and is not a particularly efficiant way to poll the server. It can also create a huge amount of server load as for each request there is a setup and tear down that may need to happen on the server. Plus there is the network overhead of HTTP headers, as well as user authentication. The HTTP headers can add a few hundred bytes to each request. In a busy server this can add a significant amount of load to the servers and network.

A second method is to open up a HTTP request to the server, and let it hang open, When the server needs to send some data it will send it to the client and then close the HTTP request. At this point the browser will open up a new connection and repeat. Depending on the specific server technology employed this can still cause a significant load on the server as a large pool of threads and connections are kept running, even if in a waiting state, though this would be less of an issue using a non blocking server such as Node.JS. There is a further complication in that the browser may only allow a limited number of Ajax requests to a given server at a time, so by holding a request or two open it may cause other things to block, making this less than an optimal way to do things.

HTML5 introduces the idea of Web Sockets, which greatly resemble TCP/IP sockets in how they work. A socket is opened by the browser back to the server from which it was loaded and can be kept open until it is no longer needed and explicitly closed. A
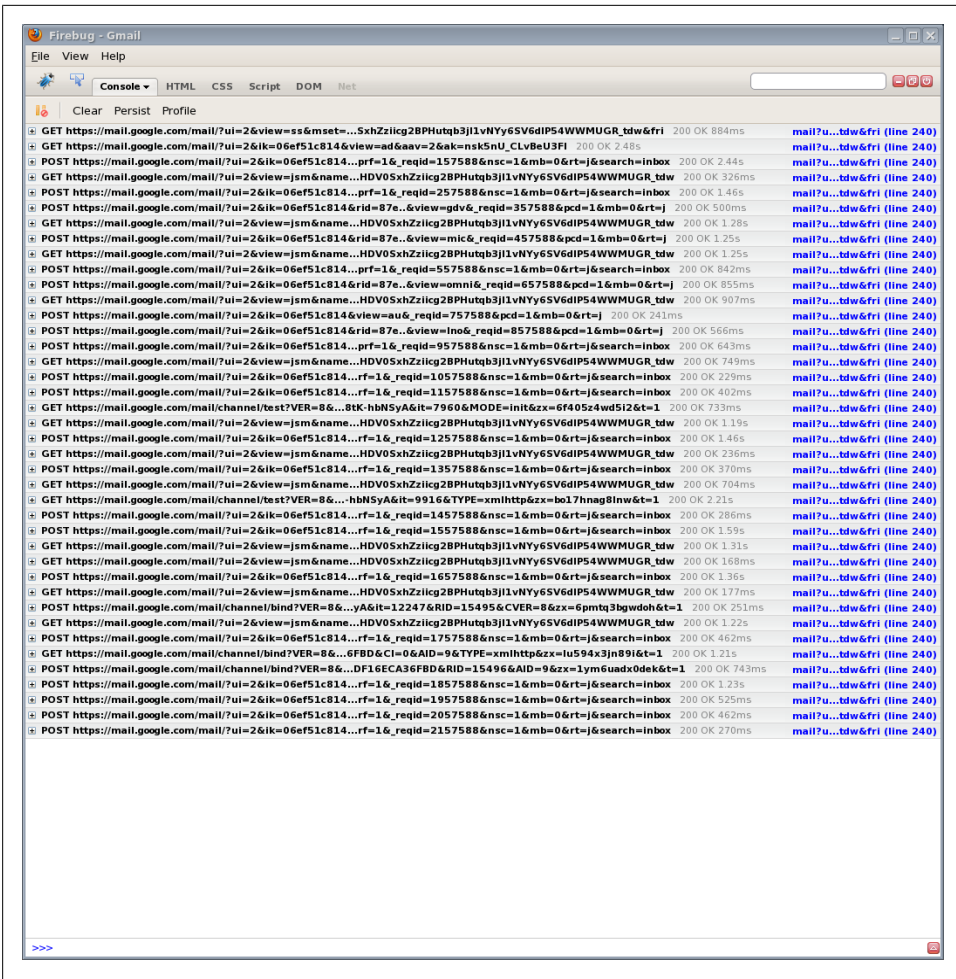
*Figure 9-1. Firebug in gmail*

socket is a bi directional real time data channel, while a HTTP request is a simple polling system. If you were to send each keystroke to the server over HTTP with Ajax then there would probably be an overhead of 300-400 bytes at a minumum, maybe as much as a kilobyte or two with cookies for each keystroke. With a socket there is no HTTP headers so much of that overhead will just go away. The overhead would be reduced to just a few bytes.

As of this writting (August 2011) Web sockets are supported by Chrome version 8 and later and Safari version 5. As of Firefox version 6 WebSockets are avalable, but the constructor is `MozWebSockets`. Opera has implemented the web sockets spec but leave it turned off by default pending security issues being worked out. For browsers that do not support WebSockets there are fallbacks using classic HTTP or Flash that can work.

There are also some libraries such as socket.io that will provide a constant interface for WebSockets and the fallback to older style HTTP communications for browsers that may not support WebSockets. It is also possible to emulate web sockets via Flash for browsers that do support flash but not web sockets.

The Web Sockets specification document also appears to be a work in progress. While web sockets have been deployed in several browsers there is still very little documentation on how to implement them. There have also been several earlier versions of of the web sockets standard that are not always compatable.

# The Web Sockets Interface

To open up a web socket create a web socket by creating a Web Socket object. As a paramater pass a Web socket URL. Unline a HTTP method a web socket URL will start with `"ws"` or `"wss"` for a secure web socket that will use SSL similar to HTTPS under Ajax.

```
var socket = new WebSocket("ws://example.com/socket");
```

Once a socket connection is opened the socket's `socket.onopen()` callback will be called to let the program know that all is ready. When the socket closes the `socket.onclose()` method will be called. If the browser wishes to close the socket it should call `socket.close()`.

To send data over the socket use the `socket.send("data")` method. This will send the data to the server over the wire. Data is limited to strings so encoding the data to JSON, XML or other data interchange format will be needed if the data is more complex than simple strings. In addition sockets are text only so if binary data must be sent it should be encoded into text in some method.

# The WebSocket Protocal

A web socket connection starts out much like a HTTP connection. It opens a connection on port 80 (ws) or 443 (wss) to the server, however in addition to the standard HTTP headers it also includes some new headers that tell the server that this is a Web socket connection and not a HTTP connection. It also includes some handshaking bytes which are used to provide some security. Since the Web Socket protcall uses ports 80 and 443 most proxies and firewalls should deal with it correctly. Web sockets can also specify a different port in the same way that a HTTP protocall can, but like an Ajax call the Web Socket must be on the same port as the web server that spawned it.

Once a connection is setup both ends can send data over it. A Web socket datagram starts with a null byte (0x00) and then ends with a 0xFF byte. In between these two markers can be any valid UTF-8 string. Binary is not supported at this time. To send binary some form of encoding would be needed. The Web Socket does not care what kind of text is sent over it. It is up to the Server and the Client to agree on data formats.

Normally data will probably be JSON or XML, but there is no reason that some other format could not be used if desired.

# Web Socket Example

To illistrate how to use websockets consider a simple example in Example 9-1. Here a very simple Javascript function opens up a socket to a server which serves up stock prices. It sends a stock ticker symbol which it is interested in (IBM). The server then will find a price for that stock and send it back to the client as a JSON. The server could be setup to poll every 5 seconds for a new price and send it to the client when it changes. The client will just refresh the element every time the price changes.

The browser code for working with Web Sockets Example 9-1 should look pretty familiar to any programmer who has worked with AJAX. A Web Socket object is created with the appropriatate URL. Once the Socket is opened (be sure to wait for it to open). Data can be sent via the `socket.send` event to the server. When the server sends data back to the browser the `socket.onmessage` event is called with the string in the data field of the event object. In this case the data is a JSON so it can be parsed with the standard browser JSON parsing methods and then displayed in the browser.

*Example 9-1. Socket Client Example*

```
$(function ()
{
  var socket = new WebSocket('ws://localhost/stockprice');

  // wait for socket to open
  socket.onopen = function ()
  {
    socket.send(JSON.stringify(
    {
      ticker: "ibm"
    }));
  };

  socket.onmessage = function (msg)
  {
    var prices = $.parseJSON(msg.data);
    var html = "IBM: " + prices.ibm;
    $('div.prices').html(html);
  }
});
```

The WebSocket client does not make much sense without a server to go with it. In general WebSockets lend themselves well to event driven data such as a shared document, Stock Ticker or chat service. While PHP has often been the standby of web server

development in this case having a language with a programming model setup for long running processes and events makes more sense.

There are several good choices here. Node.js works well, and has the advantage of being Javascript which the web programmer will already be familar with. Other possibilities include Erlang and Yaws which has a web socket interface and has a multi processor model that could be idea for this kind of programming. There are also a number of options for Java and the other languages of the JVM including Scala, Clojure and so on. There are also implementations for Ruby and probably most of the .net/CLR languages. In truth most languages that are used for web server programming will be able to use Web Sockets.

In this example, done in node.js, a server is setup using the `websocket-server` package which can be found via NPM or on github. The server waits for a connection on port 8080 and when one comes in it calls the callback. That connection callback waits for a message to arive via the connection object. In this example it then sets calls a higher order function `tickerUpdate` which somehow finds stock prices and when the relivant stock symbol has changed it calls the callback which sends the new price back to the client. For a fuller guide to programming Node see "Node: Up and Running" by Tom Hughes-Croucher.

*Example 9-2. Socket Server Example*

```
        var ws = require("websocket-server");

var server = ws.createServer();

server.addListener("connection", function (connection)
{
  connection.addListener("message", function (msg)
  {
    var tickerSymbol = msg.ticker;
    tickerUpdate(tickerSymbol, function (price)
    {
      var msg =
      {
      };
      msg[tickerSymbol] = price;
      server.send(connection.id, JSON.stringify(msg));
    });
  });
});

server.listen(8080);
```

# Web Socket Protocal

While most of the time the low level details of web sockets will not of great concern to the programmer, The interfaces in the browser and on the server will take care of the details and just provide an API which can send data.

That being said sometimes it may be useful to know the low level details of how things to work, to understand why something is not working, or to implent a Web Socket Client in some other enviroment. Web Sockets carry data between browser and server using a TCP socket instead of a HTTP envelope. However it is important to understand how a socket is setup. When the browser tries to open a socket it sends what looks like a HTTP GET request but with a few extra headers Example 9-3.

After the connection has been setup then frames of data are sent back and forth. Each frame is started with a null byte `0x00` and ends with a `0xFF` byte. Inside the envelope is data in UTF-8 format.

*Example 9-3. Socket Headers*

```
        GET /socket HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Oragin: http://www.test.com
Host: www.test.com
Content-Length: 0
```

There are server side implementations for Web Sockets that will work with Python, Ruby, Erlang, Node.JS, Java as well as other languages. Example 9-4 shows a very trivial hello world type websocket interface in Ruby. The state of libaries for WebSockets is advancing and there are packages in various states of developmet for pretty much all the major languages used in web development. In general the server side of the web sockets equation will be dictated by the other needs of a project. As such it make sense to find the web sockets package for the enviroment that is being used by a given project.

Ruby's Event Machine also provides an ideal platform for working with web sockets, as the programmer is given an event based interface. Here is a very simple implementation written in Ruby to show how to use a WebSocket interface.

*Example 9-4. Ruby Event Machine Web Socket handler*

```
        require 'em-websocket'

EventMachine::WebSocket.start(:host => "0.0.0.0", :port => 8080) do |ws|
  ws.onopen    { ws.send "Hello Client!"}
  ws.onmessage { |msg| ws.send "Pong: #{msg}" }
  ws.onclose   { puts "WebSocket closed" }
end
```

# New Tags

In addition to a lot of new interfaces for working with data, HTML5 also introduces a number of new HTML tags that can be used in a web page to enhance the application developer's ability to put out a quality application.

## Tags for Applications

A common task in any application is to give the user feedback on how far along a long-running task is. This lets the user know that her task is moving and that something didn't freeze up. It is possible to show a progress meter by using a few `<div>` elements and some custom CSS, but HTML5 standardizes the procedure and look through a new `<progress>` tag. As of this writing, the tag is supported in Firefox and Chrome. It offers two attributes to make it easy to visualize progress to the user: `value` to show the current value of the progress bar and `max` show the maximum value.

In order to show a progress indication to users of legacy browsers, it is recomended that you include some form of text inside a `<span>` element inside the progress bar.

Example 10-1 shows the code for a progress bar at 20% completion. Example 6-4 in Chapter 6 shows how JavaScript can update the attributes as events in the program indicate progress. The progress bar can also be styled with CSS like any other HTML element.

*Example 10-1. Progress indicator*

```
<!DOCTYPE html>
<html>
  <head>
    <title>Progress</title>
  </head>
  <body>
    <progress value="20" max="100">
      <span>running</span>
    </progress>
  </body>
```

```
</html>
```

If the `<progress>` element can be used to show a running event, the `<meter>` element can be used to show a static value such as how full a disk is or how much money has been raised for a fundraising goal.

The meter tag can take a few parameters, including `min`, `max`, `low`, `high`, `optimum`, and `value`. All of these should be set to numeric values. The `min` and `max` values show the ends of the range of values, whereas the `value` attribute shows the current value. The `high`, `low`, and `optimum` allow the tag to sub segment the range. Through CSS, you can style values different parts of the ranges with different looks.

Like the `<progress>` tag, the `<meter>` tag should be wrapped around a `<span>` that can show the data to browsers that do not support this tag. Currently, Chrome supports this tag, as does Opera. Other browsers may follow at some point.

*Example 10-2. Meter indicator*

```
<!DOCTYPE html>
<html>
  <head>
    <title>Meter</title>
  </head>
  <body>
    <meter min="0" value="20" max="100">
      <span>20%</span>
    </meter>
  </body>
</html>
```

# Accessibility Through WAI-ARIA

For those that may have physical disabilities, using a web application (or any other GUI application) can prove to be a great challange. So HTML5 defines Accessible Rich Internet Applications. It is directed particularly at the sight-impaired (which may include people who for some reason till use a non-graphic browser). By adding attributes to tags in an application, you may help these users access its contents.

This section is not a full guide to making an accessible application, which could fill a book of its own. The basic idea of WAI-ARIA is that it adds symantac meening to elements on the page that can be read by a screen reader to let a user who is visually impaired have context for what is happening on a page. This sort of text has been supported for some time by the `alt` attribute on `<img>` tags and the `title` attribute on `<hr>` tags.

The most common attribute in WAI-ARIA is the `role` attribute. This provides context for elements. In HTML, tags such as <span> and <li> are used for many different things, from navigation to actual lists. By adding a `role` attribute, you can help the screen reader make sense of all of this.

*Example 10-3. WAI-ARIA*

```
<!DOCTYPE html>
<html>
  <head>
    <title>Meter</title>
  </head>
  <body>
   <ul id="tree1"
      role="tree"
      tabindex="0"
      aria-labelledby="label_1">
  <li role="treeitem" tabindex="-1" aria-expanded="true">Fruits</li>
  <li role="group">
    <ul>
      <li role="treeitem" tabindex="-1">Oranges</li>
      <li role="treeitem" tabindex="-1">Pineapples</li>
      ...
    </ul>
  </li>
</ul>
</body>
</html>
```

http://stackoverflow.com/questions/4176844/can-someone-explain-the-html5-aria-attribute

# Microdata

Sometimes it is useful to add machine-readable data to a set of HTML tags. For example, you can use this procedure in a template to encode data into a page that can later be read by Javascript. To enable such procedures in a standardized way, HTML5 created the concept of *microdata*, which can be added to HTML5 tags.

Traditionally, HTML tags give information about how information should be formatted on screen, but not about the information itself. A program can look at a <li> tag and know it is an item in a list, but not what kind of list. Is it a list of books for sale, or people who attended an event? By adding microdata it is possible to give context to that data, which can later be used programaticly.

In general, microdata tags will probably be used in web pages more than in applications, but applications that run as plugins to a page, or administer a page in other ways, may be called upon to use microdata. One can imagine an HTML5 application that serves

as an administrative panel for a company's web site and offers spaces where a sales manager could put in microdata tags that give items for sale a context for for Google Search or for Javascript programs from an external vendor that may be added to a web page.

In truth, microdata is pretty simple: just a few extra attibutes with standardized names attached to HTML tags, and some interfaces to work with that data attribute.

To designate a section of a page that will use microdata, give the enclosing element an `itemscope`. The vocabulary for the microdata is defined by using the `itemtype` attibute with a URL that defines the vocabulary. Within that, data is tagged by marking the enclosing HTML tag with the `itemprop` attribute.

A number of predefined vocabularies can be found at *http://www.data-vocabulary .org*. These include specs for Events, Organizations, People, Products, Revies, Review-Aggregates, Breadcrumbs, Offers, and Offer-Aggregates. Google will understand these syntaxes to enhance search results.

In theory there should be interfaces in the DOM to parse microdata, but as of now they are not ready yet and implementation can be spotty. Thankfully, microdata is just HTML attributes, so it can be easily parsed and processed with CSS selectors via the DOM interface or with jQuery.

# New Form Types

HTML 5 enhances the classic form that has been in HTML since the early 1990s with a bunch of new form types. Most of these are variations on the classic `<input type="text">` tag that has been in use since the beginning. These new form types provide some much needed flexability in the types of input a form can take, and the interface it provides.

In many cases, on a mobile device, changing the input type will also cause the device to put up a custom keyboard to enable the user to enter the right kind of data. For instance, if `type` is set to `number`, the device can put up a numeric keypad. For a type of `tel`, the device can put up a numeric keypad that looks a little different but is optimized for entering phone numbers. For a type of `email`, the keyboard will be a standard QWERTY keyboard but modified for the entry of email addresses.

One input type that is especialy useful for smartphone applications is the Speech input type: `<input type="text" x-webkit-speech/>`. The speech tag will take what the user said and translate it into text. My Android phone, for instance, has a Google Search widget that can search by voice. The speech tag still allows the user to type text normally as well. When the user speaks, the input will fire the `webkitspeechchange` event, which can be used to interact with the user.

Be warned that for users who speak English with foreign or other non-standard accents, this tag may prove very difficult to use. Many of my Israeli and Russian coworkers find that these inputs are not very useful. It may also have limited support for languages other than English. So if your application's users speak, for example, Polish or Hebrew, the tag may or may not be useful.

HTML5 adds the ability for a form element to be required. If the `required` attribute is set and the element is blank then it can be styled in css with the `:invalid` selector.

To display a slider that can be used to allow the user to pick from a range of values, use the `range` type. Specify a `min` and `max` values as well as a starting value.

Other input types are pretty simple and mostly allow the programmer to specify what kind of data in expected, and have the browser mark a field as invalid if it is not correct. Here is a table showing some of the possible options.

*Table 10-1. Form Inputs*

| Type | Use | Notes |
| --- | --- | --- |
| email | Email Addresses | |
| date | Dates | min and max can specify a range |
| time | Time of Day | min and max can specify a range |
| tel | Telephone numbers | pattern to specify format via Regular Expression |
| color | Colors | Format like #BBBBBB |
| number | Numbers | Will show up and down arrows |
| search | Search | min and max can specify a range |

# Audio and Video

HTML5 also provides new support for audio and video via the <audio> and <video> tags. These tags should be familar to anyone who has used the html <img> tag in that the audio source or video source is set with the `src` attribute. Both can be set to display controlls with the `controls` attributes. Audio and video can also be controlled with JavaScript and styled with CSS.

A full description of how to script HTML5 media is beyond the scope of this book, but can be found in the book "HTML5 Media" by Shelly Powers.

# Canvas & SVG

In addition to providing sound and video support HTML5 provides support for building graphics in the browser with both Canvas and SVG. Classic HTML can display images but Canvas and SVG can do much more.

SVG is an XML standard for Scalable Vector Graphics which is to say that images created in SVG can be scaled and rotated and manipulated. In addition each element in a SVG image is an element in the DOM. SO it is possible to say create a circle in SVG that can have standard JavaScript event handlers attached to it. Elements in SVG can also be manipulated as parts of the DOM, elements can be addede, removed, or changed directly with the DOM or with jQuery. SVG elements can also be styled with CSS like any other HTML element. SVG is covered in depth in other books including "HTML5 Graphics with SVG & CSS3" by Kurt Cagle.

Canvas was initially created by Apple for use with OS X and moved into Safari, it has from Safari moved into most of the other browsers. Canvas provides a 2D drawing surface which on which the code can render images. This was used in "Web Worker Fractal Example" on page 89 to draw a mandelbrot set, but its full power is much greater than that. For a full treatment see "HTML5 Canvas" by Steve Fulton and Jeff Fulton.

In addition to the 2D canvas a 3D canvas based on WebGL has stared bo be implemented in a number of browsers. For demos and a tutorial look at the HTML5 Rocks tutorial. *http://www.html5rocks.com/en/tutorials/three/intro/*.

# Geolocation

The World Wide Web is by definition world wide, but there are many things on it that are local. If I am searching for a pizza place I probably want one that is near where I am located. Geolocation lets the browser determine the user's location by one of several mechanisms. If GPS is avalable (as it is on many smart phones) it will use that to get a location that is probably accurate to a few meters. If there is no GPS access then the browser can try to use information from Cell Towers or WiFi hubs, these methods may not be as accurate but often are good enough. If the goal is to find a local pizza place knowing the location to a few blocks is probably good enough. In most cases the Geolocation API will require the user to approve the request.

To get the user's position call the `getCurrentPosition()` method with two callbacks, one for a successful location and one for an error. In the case that the browser was able to find the user's location it will return with the Latitude and Longitude, altidue if it can figure it out as well as an accuracy parameter. If there is an error it will call with one of several error conditions.

```
navigator.geolocation.getCurrentPosition(userLocationCallback, errorCallback);
```

# New CSS

In additon to new JavaScript interfaces and new HTML tags HTML5 also adds a bunch of new CSS selectors, including `:nth-child()` and `:first-child` as well as the CSS neg-

ation operator `:not()` which can be used like `:not(.box)`. These should allow developers to have more controll over presentation of their applications.

In addition to new selectors HTML5 includes support for Web Fonts in CSS. It is now posible to define a new font in CSS and includ a True Type Font file in the css to get a specific look.

HTML5 CSS also includes some other pretty cool features, it is possible to be more specific in Text Overflow, to set Opacity of an object in the DOM, setup Text Strokes, specify color by HSL and a lot more. Of course like everything else in HTML5 not aqll of these are supported in all browsers.

# JavaScript Tools You Should Know

JavaScript is a young language in many ways. Although, it has been around for about 15 years, only recently has it has been used for large-scale projects. So the tools to help programmers do robust, debuggable programming are still being developed. Here are a few you should strongly consider using.

*JSLint*
> This is Douglas Crockford's JavaScript syntax checker. JSLint can test JavaScript code for all manner of possible problems. You can run JSLint from the web site (*http://jslint.com*) or locally. There is a Drag and Drop widget in the Yahoo widget set, and you can run it from the command line with Rhino (discussed later). A simple Bash script will make this easy to run (Example A-1). You can even hook JSLint into your editor.
>
> If you use one tool for JavaScript, it should be this one. To understand why, read Crockford's *JavaScript: The Good Parts* (O'Reilly). That book will describe in detail why JSLint makes the choices it does. It will catch many, very common JavaScript bugs and should be considered a key part of every JavaScript programmers tool chain.
>
> To configure JSLint on a per-file basis, you can put comments at the top of the file. These can turn options on or off in the file as well as tell JSLint about global variables that are defined else where.

*Example A-1. JSLint shell wrapper*

```
    #!/bin/bash

java -jar ~/bin/rhino/js.jar ~/bin/jslint.js $1
```

*JSMin*

It's a fact of life for the JavaScript programmer that our programs are downloaded in source form to the user's web browser. This can be sped up by minification. JSMin performs a number of actions on a file, such as removing comments and white space. After running JSMin files generally are about 30% of their original size. This means that many fewer bytes have to be transmitted to the client.



JSMin is a one way process, so make sure you have a copy of your files around. It also should not be used in development, as it makes debugging very difficult. And before running JSMin, make sure to run JSLint.

*JSBeautifier*

If your JavaScript code tends to get messy, JSBeautifier is a good tool to know about. It takes a JavaScript file and reformats it according to some basic rules. You can run this from the web site or using Rhino from a command line on your desktop. All the JavaScript code in this book has been formatted with this tool.

JSBeautifier can take a number of command-line options to specify indentation style and bracket style. Indentation can be by tabs or spaces. The `-i` option controls indentation level.

JSBeautifier can also format JSON strings. Since it is written in Javascript, you can embed it in other JavaScript programs. If you need to display a JSON structure at some point to a user, you can use this library to pretty-print it (Example A-2).

*Example A-2. JavaScript Pretty Printer*

```
        #!/bin/bash

cp $1 $1.bak
export WORKING_DIR=`pwd`
cd    ~/bin/js-buautify

java -jar ~/bin/rhino/js.jar  beautify-cl.js -d ~/bin/js-buautify/  -i 1 -b -p -n $WORKING_DIR/$1 > /tmp

mv /tmp/$1 $WORKING_DIR/$1
```

*Emacs JS2 mode*

Emacs JS2 mode is a very nice framework for JavaScript editing. For those who already live in Emacs, this is very helpful.

*Aptana*

For those who wish to have a full IDE to develop JavaScript in, Aptana is a good choice. Aptana is a verion of Eclipse customized for JavaScript. It has a lot of options that can be customized.

Aptana will reformat code, but it sometimes does so in strange ways —not bad, so much as just a little different.

*YSlow*

In a large web application, it is not unusual for loading to go slowly. If this is happening, you can use this tool along with Firebug to find out where the bottlenecks are. The tool goes along with the book *High Performance Web Sites*, which was written by the creator of YSlow. It covers much more than JavaScript, because it shows when each file is transmitted as part of the web page.

*FireRainbow*

This is a Firebug plugin that colorizes JavaScript in the Firebug script tab. It's a very nice way to make code a little easier to read in the debuger.

*Speed Tracer*

This is a Google Chrome plugin that lets you know what the browser is spending time on. Before you spend a few days optimizing JavaScript, find out if it will actually help. If CSS is the real bottleneck, this will tell you!

*CoffeeScript*

CoffeeScript is a cool new language that uses JavaScript as a compile target. If you like functional programming, this should interest you. It has been getting a solid following of late. There are several CoffeeScript books out or in the works. CoffeeScript claims to produce code that will always pass JSLint validation.

*ClojureScript*

If you like Lisp, check out ClojureScript, which is a compiler that will compile the Clojure dielect of Lisp into Javascript. It comes from Rich Hickey, who created Clojure.

*Rhino*

Rhino is a Java-based JavaScript implementation. If you want to build tools to run on a command line in JavaScript, this is the tool for the job. A number of JavaScript programs, such as JSLint, will run under Rhino as well as the browser. In adddition, Rhino can be used to allow JavaScript to script Java objects, which can be pretty useful.

*Node.js*

Node.js is a new server-side platform being developed. It uses the JavaScript event loop to create a nonblocking server that can handle a great number of requests very efficently. Watch this project for cool things in the future.