CHAPTER

11

Running Offline

If you want to view a website, you need to connect to the Internet. Everybody knows that. So why a chapter about offline applications? The very notion seems so last century. After all, didn't web applications overthrow several generations of offline, desktop applications on their way to conquering the world? And there are plenty of tasks—from following the latest Kardashian sightings to ordering a new office chair—that just wouldn't be possible without a live, real-time connection. But remember, even web applications aren't meant to stay *permanently* online. Instead, they're designed to keep working during occasional periods of downtime when a computer loses its network connection. In other words, a useful offline web application can tolerate intermittent network disruptions.

This fact is particularly important for people using smartphones and tablets. To see the problem, try traveling through a long tunnel while using a web application on one of these devices. Odds are you'll get a nasty error page, and you'll have to start all over again when you get to the other side. But do the same with an *offline* web application, and you'll avoid interruption. Some of the features of the web application may become temporarily unavailable, but you won't get booted out. (Of course, some tunnels are longer than others. An ambitious offline web application can keep working through a three-hour plane flight—or a three-week trip to the Congo, if that's what you're after. There's really no limit to how long you can stay offline.)

By using HTML5's offline application feature, you can start to shift your ordinary web page into a web-based "mini-app." And if you combine the offline feature with plenty of JavaScript code, the data storage features described in Chapter 10, and the server communication features described in Chapter 12, your mini-app can be nearly as powerful and self-sufficient as the native applications designed for smartphones

and tablets. The big advantage is that your HTML5-powered mini-app can run on *any* device, whereas a native app is locked into a specific platform.

In this chapter, you'll learn how to turn any web page (or group of web pages) into an offline application. You'll also learn how to tell when a website is available and when it's offline, and react accordingly.

FREQUENTLY ASKED QUESTION

When It Makes Sense to Go Offline

Should I make my web page offline-able?

Offline web applications don't suit every sort of web page. For example, there's really no point in turning a stock quote page into an offline web application, since the whole point of its existence is to fetch updated stock data from a web server. However, the offline feature might suit a more detailed stock analysis tool that downloads a bunch of data at once and then lets you choose how to chart it or analyze it. Using a page like this, you could download some data while you're online and then tweak options and fiddle with buttons until you reach the proverbial other side of the tunnel.

The offline feature also suits web pages that are interactive and *stateful*—ones that have piles of JavaScript code maintaining lots of information in memory. These pages do more on their own, so they make sense as offline applications. And the cost of losing your connection with one of these pages is also higher, because being kicked out in the middle of a complex task is seriously annoying. So while there's no point in making a simple page of content offline-able, it's immediately obvious that a word-processor-in-a-browser tool can benefit from offline support. In fact, an offline application like this might just be able to stand in for a more fully featured desktop program.

The other consideration is your audience. The offline application feature makes great sense if your visitors include people who don't have reliable Internet connections or are likely to need mobile access (for example, if you're creating a mapping tool for tablet devices). But if not, adding offline support might not be worth the trouble.

Caching Files with a Manifest

The basic technique that makes offline applications work is *caching*—the technique of downloading a file (like a web page) and keeping a copy of it on the web surfer's computer. That way, if the computer loses its web connection, the browser can still use the cached copy of the page. Of course, caching isn't limited to pages—it works with style sheets, JavaScript code files, pictures, fonts, and any other resource your web page needs to have on hand to do its work.

To create an offline application, you need to complete three steps. Here's the high-level overview:

1. **Create a *manifest* file.**

   A manifest is a special sort of file that tells browsers which files to store, which files not to store, and which files to substitute with something else. This package of cacheable content is called an *offline application*.

2. **Modify your web page so it refers to the manifest.**

   That way, the browser knows to download the manifest file when someone requests the page.

3. **Configure the web server.**

   Most importantly, the web server needs to serve manifest files with the proper MIME type. But as you'll see, there are a few more subtle issues that can also trip up caching.

You'll tackle all these tasks in the following sections.

UP TO SPEED

Traditional Caching vs. Offline Applications

Caching is nothing new in the web world. Browsers use caching regularly to avoid repeatedly downloading the same files. After all, if you travel through several pages in a website, and each page uses the same style sheet, why download it more than once? However, the mechanism that controls this sort of caching isn't the same as the one that makes offline applications work.

Traditional caching happens when the web server sends extra information (called *cache-control headers*) along with some file that a web browser has requested. The headers tell the browser if the file should be cached and how long to keep the cached copy before asking the web server if the file has changes. Typically, caching is brief for web pages and much longer for the resources that web pages use, like style sheets, pictures, and script files.

By comparison, an offline application is controlled by a separate file (called a manifest), and it doesn't use any time limit at all. Instead, it applies the following rule: "If a web page is part of an offline application, and if the browser has a cached copy of that application, and if the definition of that application hasn't changed, then use the cached copy." You, the web developer, can add certain exceptions—for example, telling the browser not to cache certain files or to substitute one file for another. But there's no need to worry about expiration dates and other potentially messy details.

Creating a Manifest

The manifest is the heart of HTML5's offline application feature. It's a text file that lists the files you want to cache.

The manifest always starts with the words `CACHE MANIFEST` (in uppercase), like this:

```
CACHE MANIFEST
```

After that, you list the files you want to cache. Here's an example that grabs two web pages (from the personality test example described on page 289):

```
CACHE MANIFEST

PersonalityTest.html
PersonalityTest_Score.html
```

Spaces (like the blank line in the manifest shown above) are optional, so you can add them wherever you want.

NOTE Watch for typos. If you attempt to cache a file that doesn't exist, the browser will ignore not just that file, but the entire manifest.

With an offline application, the browser must cache everything your application needs. That includes web pages and the resources these web pages use (like scripts, graphics, style sheets, and embedded fonts). Here's a more complete manifest that takes these details into account:

```
CACHE MANIFEST
# pages
PersonalityTest.html
PersonalityTest_Score.html

# styles & scripts
PersonalityTest.css
PersonalityTest.js

# pictures & fonts
Images/emotional_bear.jpg
Fonts/museo_slab_500-webfont.eot
Fonts/museo_slab_500-webfont.woff
Fonts/museo_slab_500-webfont.ttf
Fonts/museo_slab_500-webfont.svg
```

Here, you'll notice two new details. First, you'll see several lines that begin with a number sign (#). These are *comments*, which you can add to remind yourself what goes where. Second, you'll see some files that are in subfolders (for example, *emotional_bear.jpg* in the *Images* folder). As long as these files are on the web server and accessible to the browser, you can bundle them up as part of your offline application package.

Complex web pages will need a lot of supporting files, which can lead to long, complex manifest files. Worst of all, a single mistyped file name will prevent the offline application feature from working *at all*.

TIP You might decide to leave out some resources that are unimportant or overly large, like ad banners or huge pictures. That's quite all right, but if you think their absence might cause some trouble (like error messages, odd blank spaces, or scrambled layouts), consider using JavaScript to tweak your pages when the user is offline, by using the connection-checking trick described on page 370.

Once you've filled out the contents of your manifest file, you can save it in your site's root folder, alongside your web pages. You can use whatever file name you want, although you should add the file extension *.appcache* (as in *PersonalityTest.appcache*). Other file extensions may work (for example, in the early days of HTML5 some web developers used *.manifest*), but the latest versions of the HTML5 specification recommend *.appcache*. The important thing is that the web server is configured

to recognize the file extension. If you're running your own web server, you can use the setup steps described on page 360. If not, you need to talk to your web hosting company and ask them what file extensions they use to support manifest files.

TROUBLESHOOTING MOMENT

Don't Cache Pages That Use the Query String

The query string is the extra bit of information that appears on the end of some URLs, separated by a question mark. Usually, you use the query string to pass information from one web page to another. For example, the original version of the personality test uses the query string to pass the personality scores from the *PersonalityTest.html* page to the *PersonalityTest_Score.html* page. If you fill out the multiple-choice questions on the first page and click Get Score, the browser redirects you using a URL like this:

```
http://prosetech.com/html5/PersonalityTest_Score.html?e=-10&a=-5&c=10&n=5&o=20
```

Here's the problem. In the eyes of the HTML5 caching system, a request for the page *PersonalityTest_Score.html* is not the same as a request for *PersonalityTest_Score.html?e=-10&a=-5&c=10&n=5&o=20*. The first page is cached, according to the manifest. But the second URL may as well point to a completely different page. Unless you add the page name and the complete query string in the manifest, it won't be cached. And because there's no way you want to add a separate manifest entry for every possible combination of personality scores, there's no way to properly cache the query-string-enabled version of the *PersonalityTest_Score.html* page.

To avoid this problem, don't use caching and query strings at the same time. For example, if you want to add caching to the personality test, use the version that puts personality scores in local storage. (That's the version used in the caching example on page 357.)

Using Your Manifest

Just creating a manifest isn't enough to get a browser to pay attention. To put your manifest into effect, you need to refer to it in your web pages. You do that by adding the `manifest` attribute to the root `<html>` element and supplying the manifest file name, like this:

```
<!DOCTYPE html>
<html lang="en" manifest="PersonalityTest.manifest">
...
```

You need to take this step for every page that's part of your offline application. In the previous example, that means you need to change two files: *PersonalityTest.html* and *PersonalityTest_Score.html*.

NOTE A website can have as many offline applications as you want, as long as each one has its own manifest.

Putting Your Manifest on a Web Server

Testing manifest files can be a tricky process. Minor problems can cause silent failures and throw off the entire caching process. Still, at some point you'll need to give it a try to make sure your offline application is as self-sufficient as you expect.

It should come as no surprise that you can't test offline applications when you're launching files from your hard drive. Instead, you need to put your application on a web server (or use a test web server that runs on your computer, like the IIS web server that's built into Windows).

To test an offline application, follow these steps:

1. **Make sure the web server is configured to use the MIME type *text/cache-manifest* when serving manifest files (typically, those are files with the extension .appcache).**

   If the web server indicates that the file is any other type, including a plain text file, the browser will ignore the manifest completely.

NOTE Every type of web server works differently. Depending on your skills, you may need the help of your web hosting company or your neighborhood webmaster to set MIME types (step 1) and change caching settings (step 2). Page 152 has more information about MIME types, and shows one example of how you might add a new MIME type through a web hosting account.

2. **Consider turning off traditional caching (page 357) for manifest files.**

   Here's the problem. Web servers may tell web browsers to cache manifest files for a short period of time, just as they tell them to cache other types of files. This behavior is reasonable enough, but it can cause king-sized testing headaches. That's because when you update the manifest file, some browsers will ignore it and carry on with the old, cached manifest file, and so they'll keep using the old, cached copies of your web pages. (Firefox has a particularly nasty habit of sticking with out-of-date manifest files.) To avoid this problem, you should configure the web server to tell browsers not to cache manifest files, ever.

   Once again, every web server software has its own configuration system, but the basic idea is to tell your server to send a `no-cache` header whenever someone requests an .appcache file.

3. **Request the page in a web browser that supports offline applications. Virtually every browser does, except old versions of Internet Explorer—you need IE 10 or better.**

   When a web browser discovers a web page that uses a manifest, it may ask for your permission before downloading the files. Mobile devices probably will, because they have limited space requirements. Desktop browsers may or may not—for example, Firefox does (see Figure 11-1), but Chrome, Internet Explorer, and Safari don't.

   If you give your browser permission (or if your browser doesn't ask for it), the caching process begins. The browser downloads the manifest and then downloads each of the files it references. This downloading process takes place in the background and doesn't freeze up the page. It's just the same as when a browser downloads a large image or video, while displaying the rest of the page.

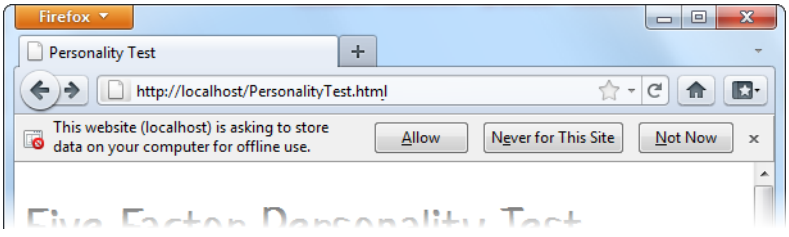


FIGURE 11-1

Firefox shows this message when it loads a web page that uses a manifest. Click Allow to grant permission to download and cache all the files that are listed in the manifest. On subsequent visits, when Firefox detects a changed manifest, it will download the new files without asking for permission again.

4. **Go offline.**

   If you're testing on a remote server, just disconnect your network connection. If you're testing on a local web server (one that's running on your computer), shut your website down (Figure 11-2).

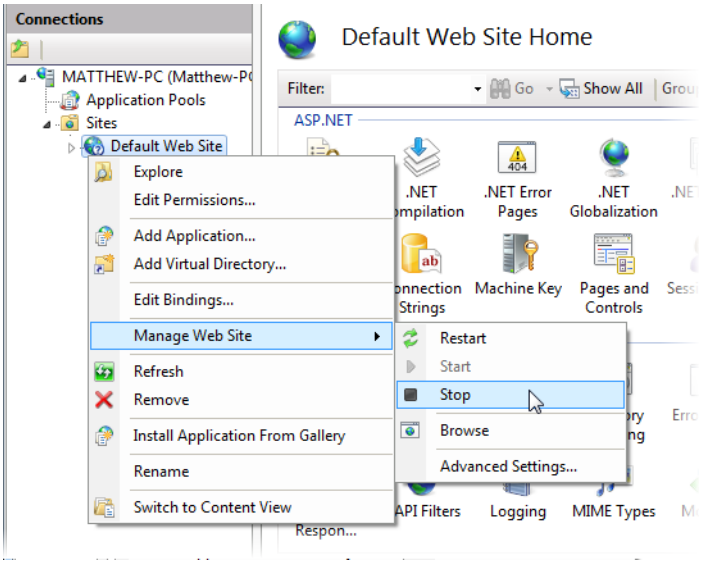


FIGURE 11-2

The exact way you shut down your test website depends on the type of web server software you're using. In the Windows-based IIS software (shown here), a simple right-click gets you started.

5. **Browse to one of the pages in your offline application, and refresh it.**

   Ordinarily, when you click the Refresh or Reload button, your browser always tries to make contact with the web server. If you're requesting an ordinary page and you've lost your Internet connection, this request will fail. But if you're requesting a page from an offline application, the browser seamlessly substitutes the cached copy, without even informing you of the switch. You can even click links to jump from one page to another, but if you navigate to a page that's not part of the offline application, you'll get the familiar "no response" error.

TROUBLESHOOTING MOMENT

My Offline Application Doesn't Work Offline

The offline application feature is fragile and a bit quirky. A minor mistake can throw it all off. If you follow the steps described above, but you get a "no response" error when you attempt to access your offline pages, check for these common problems:

- **Problems downloading the manifest.** If the manifest isn't there, or isn't accessible to the browser, you'll have an obvious problem. But equally important is serving the manifest with the right MIME type (page 152).
- **Problems downloading the files that are listed in the manifest.** For example, imagine that your manifest includes a picture that no longer exists. Or it asks for a web font file, and that font file uses a file type that your web server doesn't allow. Either way, if the browser fails to download even a single file, it will give up completely (and throw away any cached information it already has).

  To avoid this problem, start simple, with a manifest that lists just a single web page and no resources. Or, in more complex examples, look at the web server logs to find out exactly what resources the browser has requested (which may tell you the point at which it met an error and gave up).
- **An old manifest is still cached.** Browsers can cache the manifest file (according to the traditional caching rules of the Web) and ignore the fact that you've changed it. One of the signs that you've stumbled into this problem is when some pages are cached but other, more recently added pages are not. To solve this problem, consider manually clearing the browser cache (see the box on page 364).

Updating the Manifest File

Getting an application to work offline is the first challenge. The next is updating it with new content.

For example, consider the previous example (page 358), which caches two web pages. If you update *PersonalityTest.html*, fire up your browser, and reload the page, you'll still see the original, cached version of the page—regardless of whether your computer is currently online. The problem is that once a browser has a cached copy of an application, it uses that. The browser ignores the online versions of the associated web pages and doesn't bother to check whether they've changed. And because offline applications never expire, it doesn't matter how long you wait: Even months later, the browser will stubbornly ignore changed pages.

However, the browser *will* check for a new manifest file. So you can save a new copy of that, put it on the web server, and you've solved the problem, right?

Not necessarily. To trigger an update for a cached web application, you need to meet three criteria:

- **The manifest file can't be cached in the browser.** If the browser has a locally cached copy of the manifest file, it won't bother to check the web server at all. Browsers differ on how they handle manifest file caching, with some (like Chrome) always checking with the web server for new manifests. But Firefox follows the traditional rules of HTTP caching and holds onto its cached copy for some time. So if you want to save yourself development headaches, make sure your web server explicitly tells clients that they shouldn't cache manifests (page 360).
- **The manifest file needs a new date.** When a browser checks the server, the first thing it does is ask whether the last-updated timestamp has changed. If it hasn't, the web browser doesn't bother to download the manifest file.
- **The manifest file needs new content.** If a browser downloads a newly updated manifest file but discovers that the content hasn't changed, it stops the update process and keeps using the previously cached copy. This potentially frustrating step actually serves a valuable purpose. Re-downloading a cached application takes time and uses up network bandwidth, so browsers don't want to do it if it's really not necessary.

If you've been following along carefully, you'll notice a potential problem here. What if there's no reason to change the manifest file (because you haven't added any files), but you do need to force browsers to update their application cache (because some of the existing files have changed)? In this situation, you need to make a trivial change to the manifest file, so it appears to be new when it isn't. The best way to do so is with a comment, like this:

```
CACHE MANIFEST
# version 1.00.001
# pages
PersonalityTest.html
PersonalityTest_Score.html

# styles & scripts
PersonalityTest.css
PersonalityTest.js

# pictures & fonts
Images/emotional_bear.jpg
Fonts/museo_slab_500-webfont.eot
Fonts/museo_slab_500-webfont.woff
Fonts/museo_slab_500-webfont.ttf
Fonts/museo_slab_500-webfont.svg
```

The next time you need browsers to update their caches, simply change the version number in this example to 1.00.002, and so on. Presto—you now have a way to force updates and keep track of how many updates you've uploaded.

Updates aren't instantaneous. When the browser discovers a new manifest file, it quietly downloads all the files and uses them to replace the old cache content. The next time the user visits the page (or refreshes the page), the new content will appear. If you want to switch over to the newly downloaded application right away, you can use the JavaScript technique described on page 372.

NOTE There is no incremental way to update an offline application. When the application has changed, the browser tosses out the old and downloads every file again, even if some files haven't changed.

GEM IN THE ROUGH

Clearing the Browser's Cache

When testing an offline application, it's often helpful to manually clear the cache. That way, you can test new updates without changing the manifest.

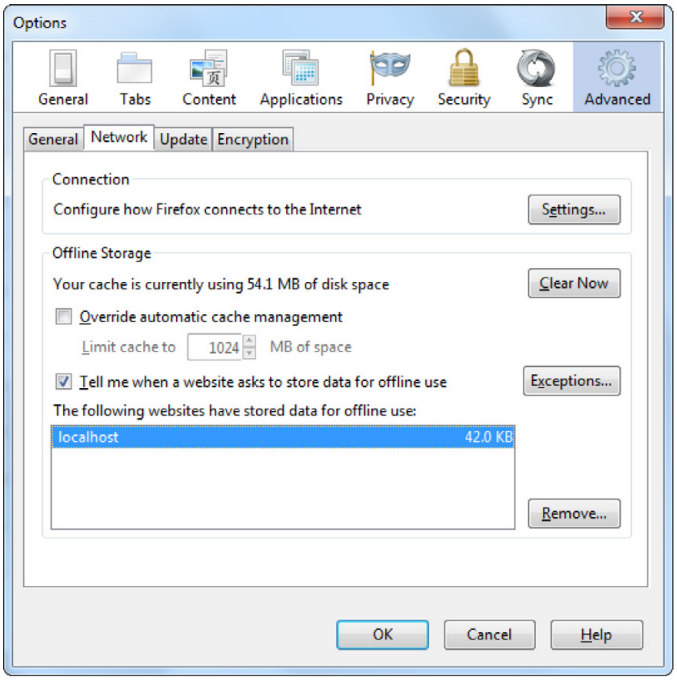
Every browser has a way to clear the cache, but every browser tucks it away somewhere different. The most useful browsers keep track of how much space each offline application uses (see Figure 11-3). This information lets you determine when caching has failed—for example, the application's website isn't listed or the cached size isn't as big as it should be. It also lets you remove the cached files for a single site without disturbing the others.

Browser Support for Offline Applications

By now, you've probably realized that all major browsers support offline applications, aside from the notable HTML5 laggard, Internet Explorer. Support stretches back several versions, which all but ensures that Firefox, Chrome, and Safari users will be able to run your applications offline. But Internet Explorer didn't get around to adding support until version IE 10, which means there's no caching in the still-popular IE 9 and IE 8.

However, the way that different browsers support offline applications isn't completely consistent. The most important difference is the amount of space they allow offline applications to fill. This variation is significant, because it sets the difference between websites that will be cached for offline access and ones that won't (see the box on page 366).

There's no worthwhile way to get offline application support on browsers that don't include it as a feature (like IE 9). However, this shouldn't stop you from using the offline application feature. After all, offline applications are really just a giant frill. Web browsers that don't support them will still work: They just require a live web connection. And people that need offline support—for example, frequent travelers—will discover the value of having a non-IE browser on hand for their disconnected times of need.



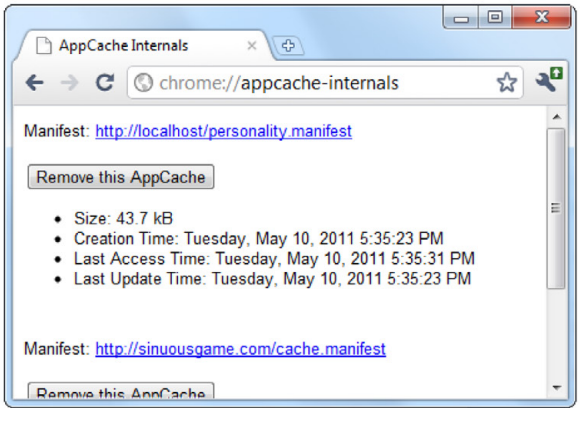


FIGURE 11-3

Top: In the Firefox menu, click Options, choose the Advanced icon, and then choose the Network tab to end up here. You can review the space usage of every website, or clear the cached resources of any one, by selecting it and clicking Remove. In this example, there's just one cached website, on the domain localhost (which represents a test server on the current computer).

Bottom: To get a similar display in Chrome, type chrome://appcache-internals *into the address bar.*

FREQUENTLY ASKED QUESTION

How Much Can You Cache?

Are there limits on how much information can be cached?

Different browsers impose wildly different size restrictions on offline applications.

Mobile browsers are the most obvious example. Because they run on space-limited devices, they tend to be stingy with their caching. Desktop browsers are more generous, but they're equally unpredictable. Browsers may assign a fixed space limit to each website domain, or they may calculate a suitable allotment based on the current amount of free space on your computer, along with other factors. Often, space is shared between several HTML5 features—for example, if you're using the File API or IndexedDB feature (see Chapter 10), your browser may use a single pool of space with these features and the application cache.

Unfortunately, the lack of consistency among browsers is a problem. If you create an offline application that attempts to stuff the cache beyond its limit, the browser quietly gives up and throws away all the downloaded data. Not only will you waste time and bytes, but your website users won't get any offline benefits. They'll be forced to use your application online.

The best rule of thumb is to assume you'll be limited to 50 MB on Apple devices (like the iPad and iPhone) and closer to 85 MB on Android devices. All mobile browsers will ask the user for permission before allowing a website to use the cache. On desktop browsers, you'll probably get a starting allotment of 250 or 350 MB. If your cache swells, some desktop browsers will offer to ratchet up the available space beyond the starting allotment, but there's no guarantee.

Practical Caching Techniques

So far, you've seen how to package up a group of pages and resources as an offline application. Along the way, you learned to write a manifest file, update it, and make sure browsers don't ignore your hard work. This knowledge is enough to put simple applications offline. However, more complex websites sometimes need more. For example, you may want to keep some content online, substitute different pages when offline, or determine (in code) whether the computer has a live Internet connection. In the following sections, you'll learn how to accomplish all these tasks with smarter manifest files and a dash of JavaScript.

Accessing Uncached Files

Earlier, you learned that once a page is cached, the web browser uses that cached copy and doesn't bother talking to any web servers. But what you may not realize is that the browser's reluctance to go online applies to *all* the resources an offline web page uses, whether they're cached or not.

For example, imagine you have a page that uses two pictures, using this markup:

```
<img src="Images/logo.png" alt="Personality quiz">
<img src="Images/emotional_bear.jpg" alt="Sad stuffed bear">
```

However, the manifest caches just one of the pictures:

```
CACHE MANIFEST
PersonalityTest.html
PersonalityTest_Score.html

PersonalityTest.css
PersonalityTest.js

Images/emotional_bear.jpg
```

You might assume that a browser will grab the *emotional_bear.png* picture from its cache, while requesting *logo.png* from the web server (as long as the computer is online). After all, that's the way it works in your browser when you step from a cached web page to an uncached page. But here, the reality is different. The browser grabs *emotional_bear.jpg* from the cache but ignores the uncached *logo.png* graphic, displaying a broken-image icon or just a blank space on the page, depending on the browser.

To solve this problem, you need to add a new section to your manifest. You title this section with a `NETWORK:` title, followed by a list of the pages that live online:

```
CACHE MANIFEST
PersonalityTest.html
PersonalityTest_Score.html

PersonalityTest.css
PersonalityTest.js
Images/emotional_bear.jpg

NETWORK:
Images/logo.png
```

Now the browser will attempt to get the *logo.png* file from the web server when the computer is online, but not do anything when it's offline.

At this point, you're probably wondering why you would bother to explicitly list files you don't want to cache. It could be for space considerations—for example, maybe you're leaving out large files to make sure your application can be cached on browsers that allow only small amounts of cache space (page 366).

But a more likely situation is that you have content that should be available when requested but never cached—for example, tracking scripts or dynamically generated ads. In this case, the easiest solution is to add an asterisk (*) in your network section. That's a wildcard character that tells the browser to go online to get every resource you haven't explicitly cached:

```
NETWORK:
*
```

You can also use the asterisk to target files of a specific type (for example, `*.jpg` refers to all JPEG images) or all the files on a specific server (for example, `http://www.google-analytics.com/*` refers to all the resources on the Google Analytics web domain).

NOTE It may occur to you that you could simplify your manifest by using the asterisk wildcard in the list of cached files. That way, you could cache bunches of files at once, rather than list each one individually. Unfortunately, the asterisk isn't supported for picking cached files, because the creators of HTML5 were concerned that careless web developers might try to cache entire mammoth websites.

Adding Fallbacks

Using a manifest, you tell the browser which files to cache and, using the network section, which files to always get from the Web and *never* cache. Manifests also support one more trick: a fallback section that lets you swap one file for another, depending on whether the computer is online or offline.

To create a fallback section, start with the `FALLBACK:` title, which you can place anywhere in your manifest. Then, list files in pairs on a single line. The first file name is the file to use when online; the second file name is the offline fallback:

```
FALLBACK:
PersonalityScore.html PersonalityScore_offline.html
```

The web browser will download the fallback file (in this case, that's *PersonalityScore_offline.html*) and add it to the cache. However, the browser won't use the fallback file unless the computer is offline. While it's online, the browser will request the other file (in this case, *PersonalityScore.html*) directly from the web server.

NOTE Remember, you don't have to be disconnected from the Web to be "offline" with respect to a web application. The important detail is whether the web domain is accessible—if it doesn't respond, for any reason, that web application is considered to be offline.

There are plenty of reasons to use a fallback. For example, you might want to substitute a simpler page when offline, a page that doesn't use the same scripts, or smaller resources. You can put the fallback section wherever you want, so long as it's preceded by the section title:

```
CACHE MANIFEST
PersonalityTest.html
PersonalityTest_Score.html

PersonalityTest.css
```

```
FALLBACK:
PersonalityScore.html PersonalityScore_offline.html
Images/emotional_bear.jpg Images/emotional_bear_small.jpg
PersonalityTest.js PersonalityTest_offline.js

NETWORK:
*
```

NOTE Incidentally, the files that you want to cache are part of the CACHE : section in the manifest. You can add the section title if you want, but you don't need it unless you want to list files after one of the other sections.

The fallback section also supports wildcard matching. This feature lets you create a built-in error page, like this:

```
FALLBACK:
/ offline.html
```

This line tells the browser to use the fallback for *any* file that isn't in the cache.

Now imagine someone attempts to request a page that's in the same website as the offline application, but isn't in the cache. If the computer is online, the web browser tries to contact the web server and get the real page. But if the computer is offline, or if the website is unreachable, or if the requested page quite simply doesn't exist, the web browser shows the cached *offline.html* page instead (Figure 11-4).

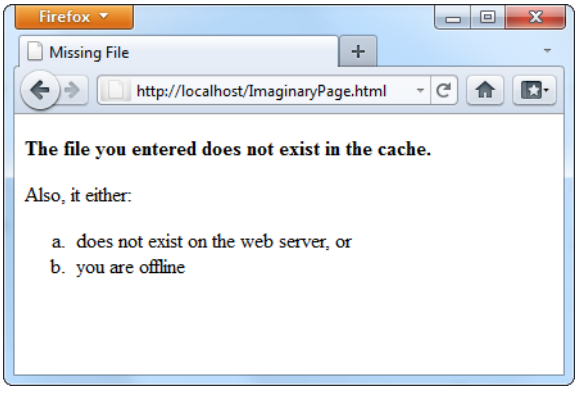


FIGURE 11-4

Here the page ImaginaryPage.html *doesn't exist. Interestingly, the browser doesn't update the address bar, so the visitor has no way of knowing the exact name of the error page.*

The previous example used the somewhat arbitrary convention of using a single forward slash character (/) to represent any page. That might strike you as a bit odd, considering that the network section uses the asterisk wildcard character for much the same purpose. (And some browsers, like Firefox, do allow you to substitute an asterisk for the slash.) You're simply looking at one more harmless HTML5 quirk.

Incidentally, you can write more targeted fallbacks that still use the slash to match all the files in a specific subfolder, like this:

```
FALLBACK:
/paint_app/ offline.html
```

GEM IN THE ROUGH

How to Bypass the Cache When You're Online

When you load up a cached page, the browser expects to find everything in the cache. It doesn't matter whether you're online or offline. The browser prefers the cache and expects to use it for everything, aside from the files you've explicitly identified in the NETWORK: section.

This behavior is straightforward, but it's also frustratingly inflexible. The chief problem happens in situations where you would *prefer* to use the online version of a page, but you still want to have the cached copy ready if you can't connect to the network. For example, think of the front page of a news site. If you're online, it makes sense to grab the latest copy of the front page every time you visit it. But if you're offline, the most recently cached page would still be helpful. The standard caching system doesn't allow for this scenario, because it forces you to choose between caching *always* and caching *never*.

In the eleventh hour of the HTML5 standardization process, a new idea slipped in that offers the solution. The trick is to add a new section, called SETTINGS:, with the following information:

```
SETTINGS:
prefer-online
```

This tells the browser to try and get resources from the network if possible, but to use the cached version if that request fails.

Although this quick fix seems ideal, it raises several problems of its own. It's an all-or-nothing setting that applies to *every* page and resource (not just the files you choose). It guarantees that the cache won't be as fast in offline mode, because the browser will waste at least some time trying to contact the web server. But the biggest drawback is that, at the time of this writing, only Firefox respects the `prefer-online` setting. Other browsers simply ignore it and carry on using the cache in the normal way.

Checking the Connection

The fallback section is the secret to a handy JavaScript trick that lets you determine whether the browser is currently online. If you're an old-hand JavaScript developer, you probably know about the `navigator.onLine` property, which provides a slightly unreliable way to check whether the browser is currently online. The problem is that the `onLine` property really reflects the state of the browser's "work offline" setting, not the actual presence of an Internet connection. And even if the `onLine` property were a more reliable indicator of connectivity, it still wouldn't tell you whether the browser failed to contact the web server or whether it failed to download the web page for some other reason.

The solution is to use a fallback that loads different versions of the same JavaScript function, depending on whether the application is online or offline. Here's how you write the fallback section:

```
FALLBACK:
online.js offline.js
```

The original version of the web page refers to the *online.js* JavaScript file:

```
<!DOCTYPE html>
<html lang="en" manifest="personality.manifest">
<head>
  <meta charset="utf-8">
  <title>...</title>
  <script src="online.js"></script>
  ...
```

It contains this very simple function:

```
function isSiteOnline() {
  return true;

}
```

But if the *online.js* file can't be accessed, the browser substitutes the *offline.js* file, which contains a method with the same name but a different result:

```
function isSiteOnline() {
  return false;

}
```

In your original page, whenever you need to know the status of your application, check with the `isSiteOnline()` function:

```
var displayStatus = document.getElementById("displayStatus");
if (isSiteOnline()) {
  // (It's safe to run tasks that require network connectivity, like
  // contacting the web server through XMLHttpRequest.)
  displayStatus.innerHTML = "You are connected and the web server is online.";
}
else {
  // (The application is running offline. You may want to hide or
  // programmatically change some content, or disable certain features.)
  displayStatus.innerHTML = "You are running this application offline.";
}
```

Pointing Out Updates with JavaScript

You can interact with the offline application feature using a relatively limited JavaScript interface. It all revolves around an object called `applicationCache`.

The `applicationCache` object provides a status property that indicates whether the browser is checking for an updated manifest, downloading new files, or doing something else. This property changes frequently and is nearly as useful as the complementary events (listed in Table 11-2), which fire as the `applicationCache` switches from one status to another.

TABLE 11-1 *Caching events*

EVENT	DESCRIPTION
onChecking	When the browser spots the manifest attribute in a web page, it fires this event and checks the web server for the corresponding manifest file.
onNoUpdate	If the browser has already downloaded the manifest, and the manifest hasn't changed, it fires this event and doesn't do anything further.
onDownloading	Before the browser begins downloading a manifest (and the pages it references), it fires this event. This occurs the first time it downloads the manifest files and during updates.
onProgress	During a download, the browser fires this event to periodically report its progress.
onCached	Signals the end of a first-time download for a new offline application. No further events occur after that happens.
onUpdateReady	Signals the end of a download that retrieved updated content. At this point, the new content is ready to use, but it won't appear in the browser window until the page is reloaded. No further events occur after that happens.
onError	Something went wrong somewhere along the process. The web server might not be reachable (in which case the page will have switched into offline mode), the manifest might have invalid syntax, or a cached resource might not be available. If this event occurs, no more follow.
onObsolete	While checking for an update, the browser discovered that the manifest no longer exists. It then clears the cache. The next time the page is loaded, the browser will get the live, latest version from the web server.

Figure 11-5 shows how these events unfold when you request a cached page.

The most useful event is onUpdateReady, which signals that a new version of the application has been downloaded. Even though the new version is ready for use, the old version of the page has already been loaded into the browser. You may want to inform the visitor of the change, in much the same way that a desktop application does when it downloads a new update:

```
<script>
window.onload = function() {

  // Attach the function that handles the onUpdateReady event.
  applicationCache.onupdateready = function() {
    var displayStatus = document.getElementById("displayStatus");
    displayStatus.innerHTML = "There is a new version of this application. " +
      "To load it, refresh the page.";
  }
}
</script>
```

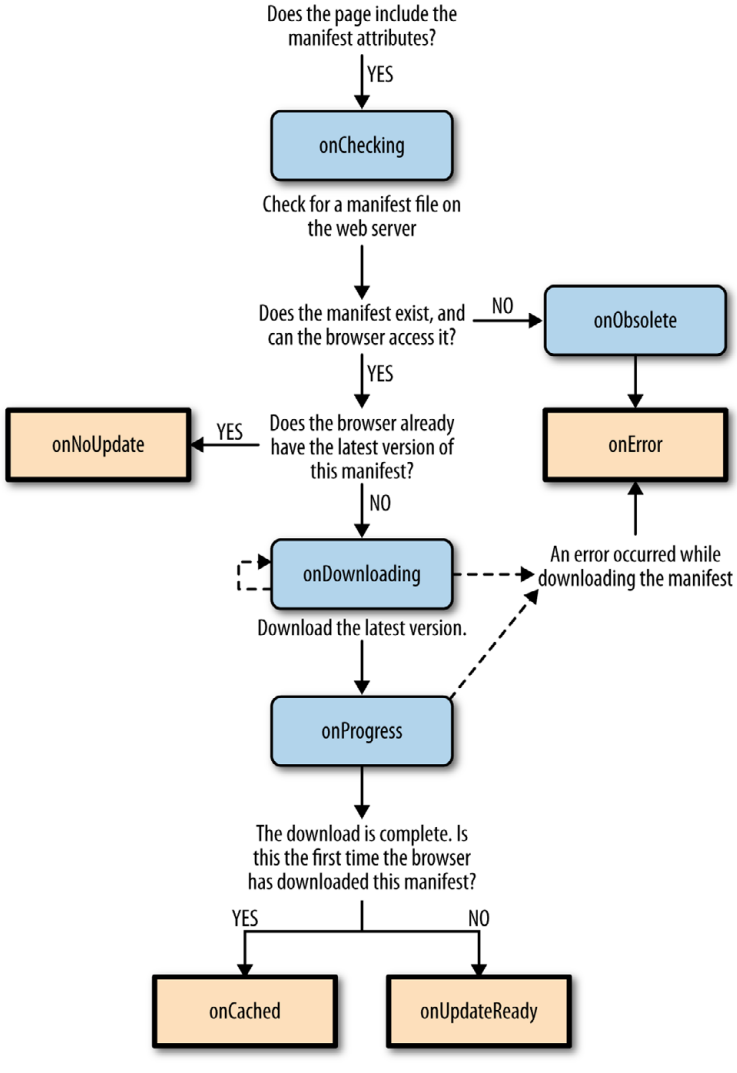


FIGURE 11-5

If you're a micromanager at heart, you can take control of the caching process. But first you need to understand how the caching event unfolds, as depicted in this diagram. The process stops when the browser reaches one of the square boxes (with a bold border) shown here.

Or, you can offer to reload the page *for* the visitor using the `window.location.reload()` method:

```
<script>
window.onload = function() {

  applicationCache.onupdateready = function() {
```

```
    if (confirm(
     "A new version of this application is available. Reload now?")) {
        window.location.reload();
      }
    }
  }
  </script>
```

Figure 11-6 shows this code at work.

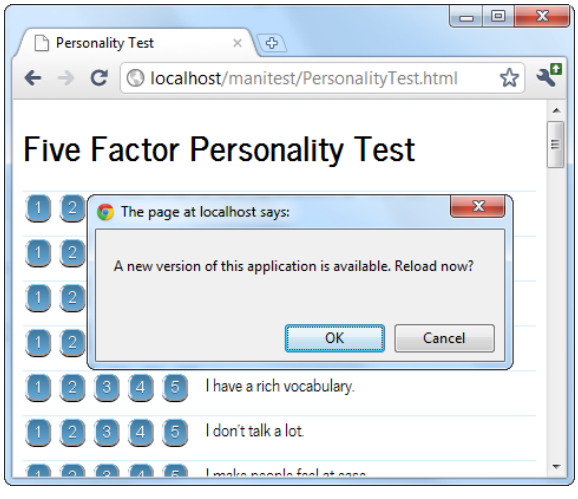


FIGURE 11-6

Here, the application offers to reload itself with the newly downloaded content, if the visitor clicks OK. (Otherwise, the new content will appear the next time the page is loaded or refreshed.)

The `applicationCache` also provides two methods for more specialized scenarios. First is the confusingly named `update()` method, which simply checks for a new manifest. If one exists, it starts the download process in the background. Otherwise, it does nothing more.

Although browsers check for updates automatically, you can call `update()` if you think the manifest has changed since the user first loaded the page. This method might be important in a very long-lived web application—for example, one that users leave open on the same page, all day.

The second method is `swapCache()`, which tells the browser to start using the newly cached content, if it has just downloaded an update. However, `swapCache()` doesn't change the page that's currently on display—for that, you need a reload. So what good is `swapCache()`? Well, by changing over to the new cache, anything that you load from that point on—say, a dynamically loaded image—comes from the new cache, not the old one. If managed carefully, `swapCache()` could give your page a way to get access to new content without forcing a complete page reload (and potentially resetting the current application to its initial state). But in most applications, `swapCache()` is more trouble than it's worth, and it can cause subtle bugs by mixing old and new bits of the cache.

CHAPTER

12

Communicating with the Web Server

When you started your journey with HTML5, you began with its markup-based features, like semantic elements, web forms, and video. But as you progressed through this book, you slowly shifted your focus to web page *programming* and the JavaScript-powered parts of HTML5. Now you're about to dip your toe into a few HTML5 features that take web page programming to the next level. They not only require JavaScript code but also some *server-side code*—code that runs on the web server, in whatever server-side programming language you want to use.

Adding a server-side language to the mix poses a bit of a problem. On one hand, it doesn't matter what server-side programming language you pick, as long as it can work with pure HTML5 pages (and they all can). On the other hand, there's not much point in getting knee-deep learning a technology that you don't plan to use or that your web host doesn't support. And there's no shortage of good choices for server-side programming, including PHP, ASP.NET, Ruby, Java, Python, and many more.

This chapter tackles the problem by using a small amount of very simple server-side code. It's just enough to round out each example and to let you test the HTML5 part of the equation (that's the JavaScript code in the web page). In your websites, you'll need to change and extend this server-side code, depending on what you're trying to accomplish and which server-side language you prefer.

So what are these features that require server-side interaction? HTML5 provides two new ways for your web pages to talk with a web server. The first feature is *server-sent events*, which lets the web server call up your page and pass it information at periodic intervals. The second feature is the much more ambitious *web sockets* framework, which lets browsers and web servers carry out a freewheeling back-and-

forth conversation. But before you explore either of these, you'll start with a review of the current-day tool for web server communication: the `XMLHttpRequest` object.

NOTE Server-sent events and web sockets seem deceptively simple. It's easy enough to learn how they work and to create a very basic example (as in this chapter). But building on that to create something that will work securely and reliably on a professional website, and provide the sort of features you want, is an entirely different matter. The bottom line is this: To implement these features in your website, you'll probably need to get the help of someone with serious server-side programming experience.

Sending Messages to the Web Server

Before you can understand the new server communication features in HTML5, you need to understand the situation that web pages were in before. That means exploring `XMLHttpRequest`—the indispensable JavaScript object that lets a web page talk to its web server. If you already know about `XMLHttpRequest` (and are using it in your own pages), feel free to skip over this section. But if your web page design career so far consists of more traditional web pages, keep reading to get the essentials.

UP TO SPEED

The History of Web Server Involvement

In the early days of the Web, communicating with a web server was a straightforward and unexciting affair. A browser would simply ask for a web page, and the web server would send it back. That was that.

A little bit later, software companies began to get clever. They devised web server tools that could get in between the first step (requesting the page) and the second step (sending the page), by running some sort of code on the web server. The idea was to change the page dynamically—for example, by inserting a block of markup in the middle of it. Sometimes code would even create a new page from scratch—for example, by reading a record in a database and generating a tailor-made HTML page with product details.

And then web developers got even more ambitious, and wanted to build pages that were more interactive. The server-side programming tools could handle this, with a bit of juggling, as long as the web browser was willing to refresh the page. For example, if you wanted to add a product to your ecommerce shopping cart, you would click a button that would submit the current page (using web forms; see Chapter 4) and ask for a new one. The web server could then send back the same page or a different page (for example, one that shows the contents of the shopping cart). This strategy was wildly successful, and just a bit clunky.

Then web developers got ambitious again. They wanted a way to build slick web applications (like email programs) without constantly posting back the page and regenerating everything from scratch. The solution was a set of techniques that are sometimes called Ajax, but almost always revolve around a special JavaScript object called `XMLHttpRequest`. This object lets web pages contact the web server, send some data, and get a response, without posting or refreshing anything. That clears the way for JavaScript to handle every aspect of the page experience, including updating content. It also makes web pages seem slicker and more responsive.

The XMLHttpRequest Object

The chief tool that lets a web page speak to a web server is the `XMLHttpRequest` object. The `XMLHttpRequest` object was originally created by Microsoft to improve the web version of its Outlook email program, but it steadily spread to every modern browser. Today, it's a fundamental part of most modern web applications.

The basic idea behind `XMLHttpRequest` is that it lets your JavaScript code make a web request on its own, whenever you need some more data. This web request takes place *asynchronously*, which means the web page stays responsive even while the request is under way. In fact, the visitor never knows that there's a web request taking place behind the scenes (unless you add a message or some sort of progress indicator to your page).

The `XMLHttpRequest` object is the perfect tool when you need to get some sort of data from the web server. Here are some examples:

- **Data that's stored on the web server.** This information might be in a file or, more commonly, a database. For example, you might want a product or customer record.
- **Data that only the web server can calculate.** For example, you might have a piece of server-side code that performs a complex calculation. You could try to perform the same calculation in JavaScript, but that might not be appropriate—for example, JavaScript might not have the mathematical smarts you need, or it might not have access to some of the data the calculation involves. Or your code might be super-sensitive, meaning you need to hide it from prying eyes or potential tamperers. Or the calculation might be so intensive that it's unlikely a desktop computer could calculate it as quickly as a high-powered web server. In all these cases, it makes sense to do the calculation on the web server.
- **Data that's on someone else's web server.** Your web page can't access someone else's web server directly. However, you can call a program on your web server (using `XMLHttpRequest`), and that program can then call the other web server, get the data, and return it to you.

The best way to really understand `XMLHttpRequest` is to start playing with it. In the following sections, you'll see two straightforward examples.

Asking the Web Server a Question

Figure 12-1 shows a web page that asks the web server to perform a straightforward mathematical calculation. The message is sent through the `XMLHttpRequest` object.

Before you can create this page, you need some sort of server-side script that will run, process the information you send it (in this case, that's the two typed-in numbers), and then send back a result. This trick is possible in every server-side programming language ever created (after all, sending back a single scrap of text is easier than sending a complete HTML document). This example uses a PHP script, largely because PHP is relatively simple and supported by almost all web hosting companies.

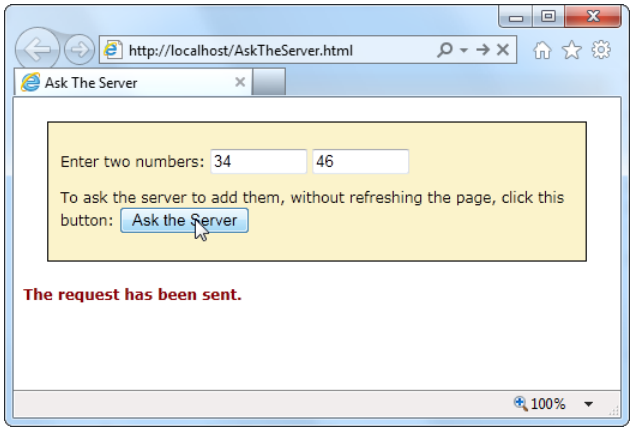


FIGURE 12-1

Click the "Ask the Server" button, and this web page creates an `XMLHttpRequest` *object and sends the two numbers to the web server. The web server runs a simple script to calculate the answer and then returns the result (shown later, in Figure 12-2).*

CREATING THE SCRIPT

To create a PHP script, you first create a new text file. Inside that text file, you start by adding the funny-looking codes shown here, which delineate the beginning and end of your script:

```
<?php
  // (Code goes here.)
?>
```

In this example the code is straightforward. Here's the complete script:

```
<?php
  $num1 = $_GET['number1'];
  $num2 = $_GET['number2'];
  $sum = $num1 + $num2
  echo($sum);
?>
```

Even if you aren't a PHP expert, you probably have a pretty good idea of what this code does just by looking over it. The first order of business is to retrieve the two numbers that the web page will have sent:

```
  $num1 = $_GET['number1'];
  $num2 = $_GET['number2'];
```

The $ symbol indicates a variable, so this code creates a variable named `$num1` and another named `$num2`. To set the variables, the code pulls a piece of information out of a built-in collection named `$_GET`. The `$_GET` collection holds all the information from the URL that was used to request the script.

For example, if you place the PHP script in a file named *WebCalculator.php*, a web request might look like this:

```
http://www.magicXMLHttpRequest.com/WebCalculator.php?number1=34&number2=46
```

Here, the URL holds two pieces of information tacked onto the end, in the URL section known as the *query string*. (The code is easier to interpret with a bit of extra spacing for illustrative purposes, as shown here. In real life, browsers don't let you use URLs with spaces.)

```
http://www.magicXMLHttpRequest.com/WebCalculator.php ? number1=34 & number2=46
```

First, the query string has a value named `number1`, which is set to 34. Next is a value named `number2`, which is set to 46. The question mark (?) denotes the start of the query string, and the ampersand symbol (&) separates each value from the one that precedes it (unless you have just a single value in your query string, in which case you don't need it). When the PHP engine fires up, it retrieves the query string values and stuffs them into the `$_GET` collection so your code can access them. (Most server-side programming platforms support a model like this. For example, in Microsoft's ASP.NET framework, you can access the same information through the `Request.QueryString` collection.)

NOTE HTML veterans know that there are two ways to send data to a web server—through the query string, or by posting it in the body of the request message. With either technique, the data is encoded in the same way, and it's accessed in the web server in a similar way as well. For example, PHP has a `$_POST` collection that provides access to any posted data.

Once the PHP script has the two numbers in hand, it simply needs to add them together and store the result in a new variable. Here, that new variable is named `$sum`:

```
$sum = $num1 + $num2
```

The last step is to send the result back to the web page that's making the request. You could package the result in a scrap of HTML markup or even some data-friendly XML. But that's overkill in this example, since plain text does just fine. But no matter what you choose, sending the data back is simply a matter of using PHP's `echo` command:

```
echo($sum);
```

Altogether, this script contains a mere four lines of PHP code. However, it's enough to establish the basic pattern: The web page asks a question, and the web server returns a result.

NOTE Could you write this page entirely in JavaScript, so that it performs its work in the browser, with no web server request? Of course. But the actual calculation isn't important. The PHP script shown here is an example that stands in for *any* server task you want to perform. You could make the PHP script as complex as you like, but the basic exchange of data will stay the same.

CALLING THE WEB SERVER

The second step is to build the page that uses the PHP script, with the help of XMLHttpRequest. It all starts out simply enough. At the beginning of the script code, an XMLHttpRequest object is created, so it will be available in all your functions:

```
var req = new XMLHttpRequest();
```

When the user clicks the "Ask the Server" button, it calls a function named askServer():

```
<div>
  <p>Enter two numbers:
    <input id="number1" type="number">
    <input id="number2" type="number">
  </p>
  <p>To ask the server to add them, without refreshing the page, click
  this button:<button onclick="askServer()">Ask the Server</button>
  </p>
</div>
<p id="result"></p>
```

Here's where the real work takes place. The askServer() function uses the XMLHttpRequest object to make its behind-the-scenes request. First, it gathers the data it needs—two numbers, which are in the text boxes in the form:

```
function askServer() {
  var number1 = document.getElementById("number1").value;
  var number2 = document.getElementById("number2").value;
```

Then it uses this data to build a proper query string, using the format you saw earlier:

```
  var dataToSend = "?number1=" + number1 + "&number2=" + number2;
```

Now it's ready to prepare the request. The open() method of the XMLHttpRequest starts you out. It takes the type of HTTP operation (GET or POST), the URL you're using to make your request, and a true or false value that tells the browser whether to do its work asynchronously:

```
  req.open("GET", "WebCalculator.php" + dataToSend, true);
```

NOTE Web experts agree unanimously—the final argument of the open() method should *always* be true, which enables asynchronous use. That's because no website is completely reliable, and a synchronous request (a request that forces your code to stop and wait) could crash your whole web page while it's waiting for a response.

Before actually sending the request, you need to make sure you have a function wired up to the event of the XMLHttpRequest object's onReadyStateChange event. This event is triggered when the server sends back any information, including the final

response when its work is done. Here, the code links the event to another function in the page, named `handleServerResponse()`:

```
    req.onreadystatechange = handleServerResponse;
```

Now you can start the process with the `XMLHttpRequest` object's `send()` method. Just remember, your code carries on without any delay. The only way to read the response is through the `onReadyStateChange` event, which may be triggered later on:

```
    req.send();

    document.getElementById("result").innerHTML = "The request has been sent.";
}
```

When the `onReadyStateChange` event occurs and you receive a response, you need to immediately check two `XMLHttpRequest` properties. First, you need to look at `readyState`, a number that travels from 0 to 4 as the request is initialized (1), sent (2), partially received (3), and then complete (4). Unless `readyState` is 4, there's no point continuing. Next, you need to look at `status`, which provides the HTTP status code. Here, you're looking for a result of 200, which indicates that everything is OK. You'll get a different value if you attempt to request a page that's not allowed (401), wasn't found (404), has moved (302), or is too busy (503), among many others. (See *www.addedbytes.com/for-beginners/http-status-codes* for a full list.)

Here's how the current example checks for these two details:

```
function handleServerResponse() {
  if ((req.readyState == 4) && (req.status == 200)) {
```

If those criteria are met, you can retrieve the result from the `responseText` property. In this case, the response is the new sum. The code then displays the answer on the page (Figure 12-2):

```
    var result = req.responseText;
    document.getElementById("result").innerHTML = "The answer is: " +
     result + ".";
  }
}
```

The `XMLHttpRequest` object doesn't make any assumptions about the type of data you're requesting. The name of the object has XML in it because it was originally designed with XML data in mind, simply because XML is a convenient, logical package for parceling up structured information. However, `XMLHttpRequest` is also used with requests for simple text (as in this example), JSON data (page 329), HTML (as in the next example), and XML. In fact, non-XML uses are now *more* common than XML uses, so don't let the object name fool you.

TIP You need to put your web pages on a test web server before you can use any server-side code, including PHP scripts. To run the example pages in this chapter without the frustration, visit the try-out site at *http://prosetech.com/html5*.

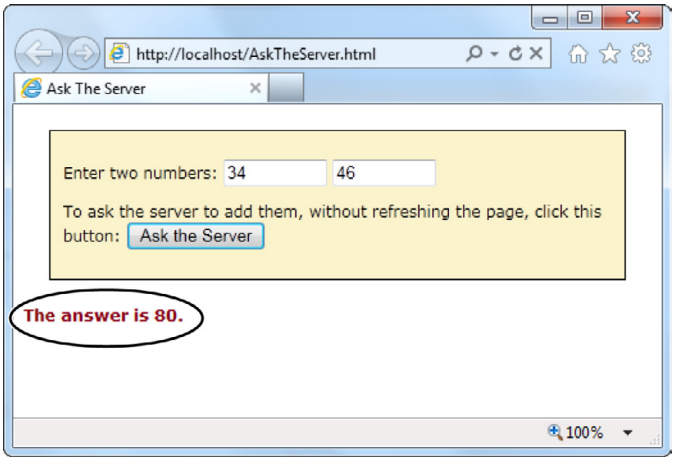


FIGURE 12-2

The web server has returned a response, triggering a JavaScript function, which changed the page.

Getting New Content

Another scenario where `XMLHttpRequest` comes in handy is loading new HTML content into a page. For example, a news article might contain multiple pictures but show just one at a time. You click a button, and some JavaScript fetches the content for the next picture and inserts it in the page. Or a page might use the same technique to show the slides in a top-five or top-10 list. Figure 12-3 shows a slideshow example that shows a series of captioned pictures that accompany an article.

There are a number of reasons to use a design like the one shown in Figure 12-3. Done skillfully, this technique can be a great way to tame huge amounts of content, so it's readily available but not immediately overwhelming. (In less capable hands, it's just a desperate attempt to boost page views by forcing readers to make multiple requests to get all the content they want.)

The best way to design this sort of page is with the `XMLHttpRequest` object. That way, the page can request new content and update the page without triggering a full refresh. And full refreshes are bad, because they download extra data, cause the page to flicker, and scroll the user back to the top. All of these details seem minor, but they make the difference between a website that feels slick and seamless, and one that seems hopelessly clunky and out of date.

FIGURE 12-3

This page splits its content into separate slides. Click the Previous or Next link to load in a new slide, with different text content and a new picture. To make this work, the page uses the `XMLHttpRequest` *object to request the new content as it's needed.*

To build the example in Figure 12-3, you first need to carve out a spot for the dynamic content. Here it's a element that creates a golden box and has two links underneath:

```
<div id="slide">Click Next to start the show.</div>
<a onclick="return previousSlide()" href="#">&lt; Previous</a> 
<a onclick="return nextSlide()" href="#">Next &gt;</a>
```

The links call previousSlide() or nextSlide(), depending on whether the visitor is traveling forward or backward in the list of sites. Both functions increment a counter that starts at 0, moves up to 5, and then loops back to 1. Here's the code for the nextSlide() function:

```
var slideNumber = 0;

function nextSlide() {
  // Move the slide index ahead.
  if (slideNumber == 5) {
    slideNumber = 1;
  } else {
    slideNumber += 1;
  }

  // Call another function that shows the slide.
  goToNewSlide();

  // Make sure the link doesn't actually do anything (like attempt
  // to navigate to a new page).
  return false;
}
And here's the very similar code for previousSlide():
function previousSlide() {
  if (slideNumber == 1) {
    slideNumber = 5;
  } else {
    slideNumber -= 1;
  }

  goToNewSlide();
  return false;
}
```

Both functions rely on another function, goToNewSlide(), which does the real work. It uses XMLHttpRequest to contact the web server and ask for the new chunk of data.

The real question: Where does the *ChinaSites.html* page get its data from? Sophisticated examples might call some sort of web service or PHP script. The new content could be generated on the fly or pulled out of a database. But this example uses

a low-tech solution that works on any web server—it looks for a file with a specific name. For example, the file with the first slide is named *ChinaSites1_slide.html*, the file with the second is *ChinaSites2_slide.html*, and so on. Each file contains a scrap of HTML markup (not an entire page). For example, here's the content in *ChinaSites5_slide.html*:

```
<figure>
  <h2>Wishing Tree</h2>
  <figcaption>Make a wish and toss a red ribbon up into the branches
  of this tree. If it sticks, good fortune may await.</figcaption>
  <img src="wishing_tree.jpg">
</figure>
```

Now that you know where the data is stored, it's easy enough to create an `XMLHttpRequest` that grabs the right file. A simple line of code can generate the right file name using the current counter value. Here's the `goToNewSlide()` function that does it:

```
var req = new XMLHttpRequest();

function goToNewSlide() {
  if (req != null) {
    // Prepare a request for the file with the slide data.
    req.open("GET", "ChinaSites" + slideNumber + "_slide" + ".html", true);
    // Set the function that will handle the slide data.
    req.onreadystatechange = newSlideReceived;

    // Send the request.
    req.send();
  }
}
```

The last step is to copy the retrieved data in the `<div>` that represents the current slide:

```
function newSlideReceived() {
  if ((req.readyState == 4) && (req.status == 200)) {
    document.getElementById("slide").innerHTML = req.responseText;
  }
}
```

TIP To give this example a bit more pizzazz, you could create a transition effect. For example, the new picture could fade into view while the old one fades out of sight. All you need to do is alter the `opacity` property, either with a JavaScript timer (page 301) or a CSS3 transition (page 199). This is one of the advantages of dynamic pages that use `XMLHttpRequest`—they can control exactly how new content is presented.

This isn't the last you'll see of this example. In Chapter 13 (page 428), you'll use HTML5's history management to manage the web page's URL so that the URL

changes to match the currently displayed slide. But for now, it's time to move on to two new ways to communicate with the web server.

NOTE If you're using the popular jQuery JavaScript toolkit, you probably won't use `XMLHttpRequest` directly. Instead, you'll use jQuery methods, like `jQuery.ajax()`, which use `XMLHttpRequest` behind the scenes. The underlying technology remains the same, but jQuery streamlines the process.

Server-Sent Events

The `XMLHttpRequest` object lets your web page ask the web server a question and get an immediate answer. It's a one-for-one exchange—once the web server provides its response, the interaction is over. There's no way for the web server to wait a few minutes and send another message with an update.

However, there are some types of web pages that could use a longer-term web server relationship. For example, think of a stock quote on Google Finance (*www.google.com/finance*). When you leave that page open on your desktop, you'll see regular price updates appear automatically. Another example is a news ticker like the one at *www.bbc.co.uk/news*. Linger here, and you'll find that the list of headlines is updated throughout the day. You'll find similar magic bringing new messages into the inbox of any web-based mail program, like Microsoft Outlook.com (*www.outlook.com*).

In all these examples, the web page is using a technique called *polling*. Periodically (say, every few minutes), the page checks the web server for new data. To implement this sort of design, you use JavaScript's `setInterval()` or `setTimeout()` functions (see page 301), which trigger your code after a set amount of time.

Polling is a reasonable solution, but it's sometimes inefficient. In many cases, it means calling the web server and setting up a new connection, only to find out that there's no new data. Multiply this by hundreds or thousands of people using your website at once, and it can add up to an unnecessary strain on your web server.

One possible solution is *server-sent events*, which let a web page hold an open connection to the web server. The web server can send new messages at any time, and there's no need to continually disconnect, reconnect, and run the same server script from scratch. (Unless you want to, because server-sent events *also* support polling.) Best of all, the server-sent event system is simple to use, works on most web hosts, and is sturdily reliable. However, it's relatively new, as Table 12-1 attests, with no support in current versions of Internet Explorer.

NOTE If you're looking for a polyfill that can fake server-sent event support using polling, there are several candidates worth checking out at *http://tinyurl.com/polyfills*.

TABLE 12-1 *Browser support for server-sent events*

	IE	FIREFOX	CHROME	SAFARI	OPERA	SAFARI IOS	CHROME FOR ANDROID
Minimum version	-	6	6	5	11	4	29

In the following sections, you'll put together a simple example that demonstrates server-sent events.

The Message Format

Unlike `XMLHttpRequest`, the server-sent events standard doesn't let you send just any data. Instead, you need to follow a simple but specific format. Every message must start with the text `data:` followed by the actual message text and the *new line* character sequence, which is represented as `\n\n` in many programming languages, including PHP.

Here's an example of what a line of message text looks like as it travels over the Internet:

```
data: The web server has sent you this message.\n\n
```

It's also acceptable to split a message over multiple lines. You use the end-of-line character sequence, which is often represented as a single `\n`. This makes it easier to send complex data:

```
data: The web server has sent you this message.\n
data: Hope you enjoy it.\n\n
```

You'll notice that you still need to start each line with `data:` and you still need to end the entire message with `\n\n`.

You could even use this technique to send JSON-encoded data (page 329), which would allow the web page to convert the text into an object in a single step:

```
data: {\n
data: "messageType": "statusUpdate",\n
data: "messageData": "Work in Progress"\n
data: }\n\n
```

Along with the message data, the web server can send a unique ID value (using the prefix `id:`) and a connection timeout (using `retry:`):

```
id: 495\n
retry: 15000\n
data: The web server has sent you this message.\n\n
```

Your web page pays attention to the message data, but it doesn't deal with the ID and connection timeout information. Instead, the *browser* uses these details. For example, after reading the above message, the browser knows that if it loses its connection to the web server, it should attempt to reconnect after 15,000 milliseconds (15 seconds). It should also send the ID number 495 to help the web server identify it.

NOTE A web page can lose its connection to the web server for a variety of reasons, including a brief network outage or a proxy server that times out while waiting for data. If possible, the browser will attempt to reopen the connection automatically after waiting a default 3 seconds.

Sending Messages with a Server Script

Now that you know the message format, it's trivially easy to create some server-side code that spits it out. Once again, it makes sense to turn to PHP to build a straightforward example that virtually all web hosts will support. Figure 12-4 shows a page that gets regular messages from the server. In this case, the messages are simple—they contain the current time on the web server.

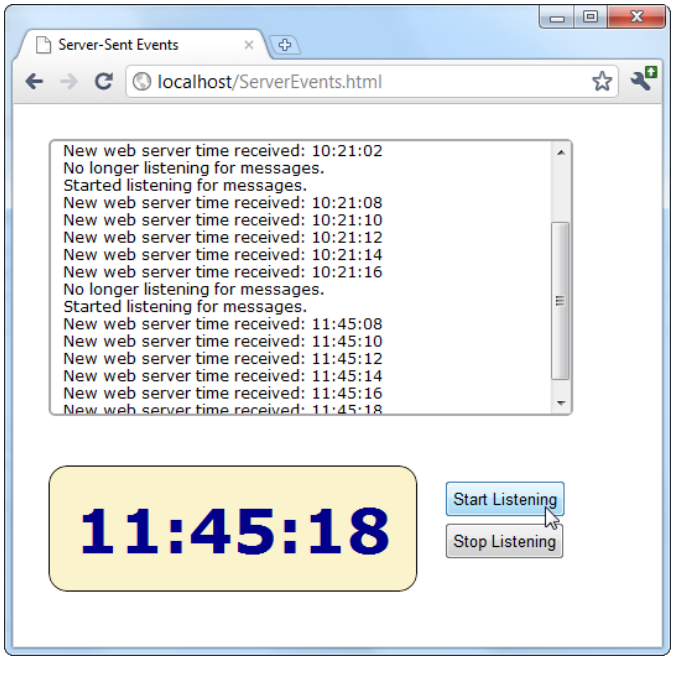


FIGURE 12-4

When this page is listening, it receives a steady stream of messages from the web server—approximately one every 2 seconds. Each message is added to the scrolling list at the top, and the time display at the bottom shows the time received from the most recent message.

NOTE The web server time is a single piece of information that's continuously updated. That makes it a good candidate for creating a simple server-side event demonstration like this one. However, in a real example, you're more likely to send something more valuable, like the most up-to-date headlines for a news ticker.

The server-side part of this example simply reports the time, in regular intervals. Here's the complete script, at a glance:

```
<?php
  header('Content-Type: text/event-stream');
  header('Cache-Control: no-cache');
  ob_end_clean();

  // Start a loop that goes forever.
  do {
    // Get the current time.
    $currentTime = date('h:i:s', time());

    // Send it in a message.
    echo 'data: ' . $currentTime . PHP_EOL;
    echo PHP_EOL;

    flush();

    // Wait 2 seconds before creating a new message.
    sleep(2);
  } while(true);
?>
```

The beginning of this script sets two important headers. First, it sets the MIME type to `text/event-stream`, which is required by the server-side event standard:

```
header('Content-Type: text/event-stream');
```

Then, it tells the web server (and any proxy servers) to turn off web caching. Otherwise, it's possible that some of the time messages will arrive in uneven batches:

```
header('Cache-Control: no-cache');
```

There's one more step needed to turn off PHP's built-in buffering system. This way, the data your PHP script returns is delivered to the browser immediately.

```
ob_end_clean();
```

The rest of the code is wrapped in a loop that continues indefinitely (or at least until the client disappears). Each time the loop runs, it uses the built-in `time()` function to grab the current time (in the format *hours:minutes:seconds*), and it stuffs that into a variable:

```
$currentTime = date('h:i:s', time());
```

Next, the loop uses this information to build a message in the proper format, which it sends using PHP's trusty `echo` command. In this example, the message is single line, starting with `data:` and followed by the time. It ends with the constant `PHP_EOL`

(for PHP *end of line*), which is a shorthand way of referring to the \n character sequence described earlier:

```
echo 'data: ' . $currentTime . PHP_EOL;
echo PHP_EOL;
```

NOTE If this looks funny, it's probably because PHP uses the dot operator (.) to join strings. It works in the same way as the + operator with text in JavaScript, only there's no way to accidentally confuse it with numeric addition.

The `flush()` function makes sure the data is sent right away, rather than buffered until the PHP code is complete. Finally, the code uses the `sleep()` function to stall itself for 2 seconds before continuing with a new pass through the loop.

TIP If you wait a long time between messages, your connection might be cut off by a *proxy server* (a server that sits between your web server and the client's computer, directing traffic). To avoid this behavior, you can send a comment message every 15 seconds or so, which is simply a colon (`:`) with no text.

Processing Messages in a Web Page

The web page that listens to these messages is even simpler. Here's all the markup from the `<body>` section, which divides the pages into three `<div>` sections—one for the message list, one for the big time display, and one for the clickable buttons that start the whole process:

```
<div id="messageLog"></div>
<div id="timeDisplay"></div>
<div id="controls">
  <button onclick="startListening()">Start Listening</button><br>
  <button onclick="stopListening()">Stop Listening</button>
</div>
```

When the page loads, it looks up these `messageLog` and `timeDisplay` elements and stores them in global variables so they'll be easy to access in all your functions:

```
var messageLog;
var timeDisplay;

window.onload = function() {
  messageLog = document.getElementById("messageLog");
  timeDisplay = document.getElementById("timeDisplay");
};
```

The magic happens when someone clicks the Start Listening button. At this point, the code creates a new `EventSource` object, supplying the URL of the server-side resource that's going to send the messages. (In this example, that's a PHP script named *TimeEvents.php*.) It then attaches a function to the `onMessage` event, which fires whenever the page receives a message.

```
var source;

function startListening() {
  source = new EventSource("TimeEvents.php");
  source.onmessage = receiveMessage;
  messageLog.innerHTML += "<br>" + "Started listening for messages.";
}
```

TIP To check for server-side event support, you can test if the `window.EventSource` property exists. If it doesn't, you'll need to use your own fallback approach. For example, you could use the `XMLHttpRequest` object to make periodic calls to the web server to get data.

When `receiveMessage` is triggered, you can get the message from the `data` property of the event object. In this example, the data adds a new message in the message list and updates the large clock:

```
function receiveMessage(e) {
  messageLog.innerHTML += "<br>" + "New web server time received: " + e.data;
  timeDisplay.innerHTML = e.data;
}
```

You'll notice that once the message is delivered to your page, the pesky `data:` and `/n/n` details are stripped out, leaving you with just the content you want.

Finally, a page can choose to stop listening for server events at any time by calling the `close()` method of the `EventSource` object. It works like this:

```
function stopListening() {
  source.close();
  messageLog.innerHTML += "<br>" + "No longer listening for messages.";
}
```

Polling with Server-Side Events

The previous example used server-side events in the simplest way. The page makes a request, the connection stays open, and the server sends information periodically. The web browser may need to reconnect (which it will do automatically), but only if there's a problem with the connection or if it decides to temporarily stop the communication for other reasons (for example, low battery in a mobile device).

But what happens if the server script ends and the web server closes the connection? Interestingly, even though no accident occurred, and even though the server deliberately broke off communication, the web page still automatically reopens the connection (after waiting the default 3 seconds) and requests the script again, starting it over from scratch.

You can use this behavior to your advantage. For example, imagine that you create a relatively short server script that sends just one message. Now your web page acts like it's using polling (page 386), by periodically reestablishing the connection. The

only difference is that the web server tells the browser how often it should check for new information. In a page that uses traditional polling, this detail is built into your JavaScript code.

The following script uses a hybrid approach. It stays connected (and sends periodic messages) for 1 minute. Then it recommends that the browser try again in 2 minutes and closes the connection:

```
<?php
  header('Content-Type: text/event-stream');
  header('Cache-Control: no-cache');

  ob_end_clean();

  // Tell the browser to wait 2 minutes before reconnecting,
  // when the connection is closed.
  echo 'retry: 120000' . PHP_EOL;

  // Store the start time.
  $startTime = time();

  do {
    // Send a message.
    $currentTime = date('h:i:s', time());
    echo 'data: ' . $currentTime . PHP_EOL;
    echo PHP_EOL;
    flush();

    // If a minute has passed, end this script.
    if ((time() - $startTime) > 60) {
      die();
    }

    // Wait 5 seconds, and send a new message.
    sleep(5);
  } while(true);
?>
```

Now when you run the page, you'll get a minute's worth of regular updates, followed by a 2-minute pause (Figure 12-5). In a more sophisticated example, you might have the web server send a special message to the web browser that tells it there's no reason to wait for updated data (say, the stock markets have closed for the day). At this point, the web page could call the `close()` method of the `EventSource` object.

NOTE With complex server scripts, the web browser's automatic reconnection feature may not work out so well. For example, the connection may have been cut off while the web server was in the middle of a specific task. In this case, your web server code can send each client an ID (as described on page 387), which will be sent back to the server when the browser reconnects. However, it's up to your server-side code to generate a suitable ID, keep track of what task each ID is doing (for example, by storing some information in a database), and then attempt to pick up where you left off. All of these steps can be highly challenging if you lack super-black-belt coding skills.

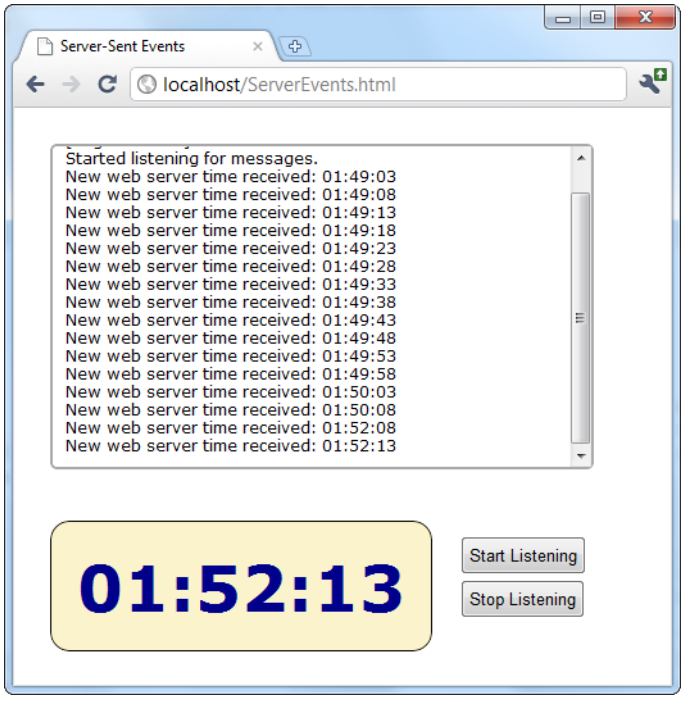


FIGURE 12-5

This page uses a combination of streaming (for a batch of messages, over the course of a minute), followed by polling (for the next 2 minutes). This sort of design might make sense as a way to minimize web server traffic, depending on how often your data is updated and how important it is to have up-to-the-minute data.

Web Sockets

Server-sent events are a perfect tool if you want to receive a series of messages from the web server. But the communication is completely one-sided. There's no way for the browser to respond, or to enter into a more complex dialogue.

If you're creating a web application where the browser and the web server need to have a serious conversation, your best bet (without adopting Flash) is to use the `XMLHttpRequest` object. Depending on the sort of application you're building, this approach may work fine. However, there are stumbling blocks aplenty. First, the `XMLHttpRequest` object doesn't work well if you need to send multiple messages

back and forth very quickly (like you might in a chat application, for example). Second, there's no way to associate one call with the next, so every time the web page makes a request, the web server needs to sort out who you are all over again. If your web page needs to make a chain of related requests, your web server code can become frightfully complicated.

There's a solution to these problems, but it's not for the faint of heart. That solution is *web sockets*, a standard that lets a browser hold open a connection to a web server and exchange back-and-forth messages for as long as it wants. The web sockets feature has generated plenty of excitement, and has gained reasonably good browser support (see Table 12-2), although Internet Explorer users need IE 10 or later to use it.

TABLE 12-2 *Browser support for web sockets*

	IE	FIREFOX	CHROME	SAFARI	OPERA	SAFARI IOS	CHROME FOR ANDROID
Minimum version	10	11	14	6	12.1	6	29

TIP Web sockets are a bit finicky. For example, you could run a browser that supports them but still run into trouble because of restrictive computer network settings, a firewall, or the antivirus software on your computer. If you're in doubt about whether your computer can use web sockets, go to *http://websocketstest.com*. This site attempts to connect to a test server and provides a handy single-page report that tells you whether web sockets are working.

Before you use web sockets, you need to understand two important points. First, web sockets are a specialized tool. They make great sense in a chat application, a massive multiplayer game, or a peer-to-peer collaboration tool. They allow new types of applications, but they probably don't make sense in most of today's JavaScript-powered web applications (like ecommerce websites).

Second, web socket solutions can be fiendishly complex. The web page JavaScript is simple enough. But to build the server-side application, you'll need mad programming skills, including a good grasp of multithreading and networking concepts, which are beyond the scope of this book. However, if some other company, service, or hotshot programmer has already created a web socket server for you to use, you won't have too much trouble talking to it using a bit of HTML5-enhanced JavaScript. You'll learn how in the following sections.

The Web Socket Server

In order to use web sockets, you need to run a program (called a *web socket server*) on the web server for your website. This program has the responsibility of coordinating everyone's communication, and once it's launched it keeps running indefinitely.

NOTE Many web hosts won't allow long-running programs, unless you pay for a *dedicated server* (a web server that's allocated to your website, and no one else's). If you're using ordinary shared hosting, you probably can't create web pages that use the web socket feature. Even if you can manage to launch a web socket server that keeps running, your web host will probably detect it and shut it down.

To give you an idea of the scope of a web socket server, consider some of the tasks a web socket server needs to manage:

- Set the message "vocabulary"—in other words, decide what types of messages are valid and what they mean.
- Keep a list of all the currently connected clients.
- Detect errors sending messages to clients, and stop trying to contact them if they don't seem to be there anymore.
- Deal with any in-memory data—that is, data that all web clients might access—safely. Subtle issues abound—for example, consider what happens if one client is trying to join the party while another is leaving, and the connection details for both are stored in the same in-memory object.

Most developers will never create a server-side program that uses sockets; it's simply not worth the considerable effort. The easiest approach is to install someone else's socket server and design custom web pages that use it. Because the JavaScript part of the web socket standard is easy to use, this method won't pose a problem. Another option is to pick up someone else's socket server code and then customize it to get the exact behavior you want. Right now, plenty of projects are developing usable web socket servers (many of them free and open source) for a variety of tasks, in a variety of server-side programming languages. You'll get the details on page 399.

A Simple Web Socket Client

From the web page's point of view, the web socket feature is easy to understand and use. The first step is to create the `WebSocket` object. When you do, you supply a URL that looks something like this:

```
var socket = new WebSocket("ws://localhost/socketServer.php");
```

The URL starts with *ws://*, which is the new system for identifying web socket connections. However, the URL still leads to a web application on a server (in this case, the script named `socketServer.php`). The web socket standard also supports URLs that start with wss://, which indicates that you want to use a secure, encrypted connection (just as you do when requesting a web page that starts with https:// instead of http://).

NOTE Web sockets aren't limited to contacting their home web server. A web page can open a socket connection to a web socket server that's running on another web server, without any extra work.

Simply creating a new WebSocket object causes your page to attempt to connect to the server. You deal with what happens next using one of the WebSocket's four events: onOpen (when the connection is first established), onError (when there's a problem), onClose (when the connection is closed), and onMessage (when the page receives a message from the server):

```
socket.onopen = connectionOpen;
socket.onmessage = messageReceived;
socket.onerror = errorOccurred;
socket.onopen = connectionClosed;
```

For example, if the connection has succeeded, it makes sense to send a confirmation message. To deliver a message, you use the WebSocket object's send() method, which takes ordinary text. Here's a function that handles the onOpen event and sends a message:

```
function connectionOpen() {
  socket.send("UserName:jerryCradivo23@gmail.com");
}
```

Presumably, the web server will receive this and then send a new message back.

You can use the onError and onClose events to notify the web page user. However, the most important event (by far) is the onMessage event that fires every time the web server delivers new data. Once again, the JavaScript that's involved is perfectly understandable—you simply grab the text of the message from the data property:

```
function messageReceived(e) {
  alert("You received a message: " + e.data);
}
```

If the web page decides its work is done, it can easily close the connection with the close() method:

```
socket.close();
```

However, once the socket is closed, you can't send any more messages unless you recreate the socket object. Recreating the socket object is the same as creating it for the first time—you use the new keyword, supply the URL, and attach all your event handlers. If you plan to be connecting and disconnecting frequently, you'll want to move this code into separate functions so you can call on them when needed.

As you can see, the WebSocket object is surprisingly simple. In fact, you've now covered all the methods and events it offers. Based on this overview, you can see that using someone else's socket server is a breeze—you just need to know what messages to send and what messages to expect.

NOTE A lot of behind-the-scenes work takes place to make a web socket connection work. First, the web page makes contact using the well-worn HTTP standard. Then it needs to "upgrade" its connection to a web socket connection that allows unfettered back-and-forth communication. At this point, you could run into a problem if a proxy server sits between your computer and the web server (for example, on a typical company network). The proxy server may refuse to go along with the plan and drop the connection. You can deal with this problem by detecting the failed connection (through the `WebSocket`'s `onError` event) and falling back on one of the socket polyfills described on GitHub at *http://tinyurl.com/polyfills*. They use tricks like polling to simulate a web socket connection as well as possible.

Web Socket Examples on the Web

Curious to try out web sockets for yourself? There are plenty of places on the web where you can fire up an example.

For starters, try *www.websocket.org/echo.html*, which features the most basic web socket server imaginable: You send a message, and it echoes the same message back to you (Figure 12-6). While this isn't very glamorous, it lets you exercise all the features of the WebSocket class. In fact, you can create your own pages that talk to the echo server, which is a good way to practice your web socket skills. As long as you use the correct socket server URL (in this case, that's *ws://echo.websocket.org*), the code works perfectly well. It doesn't matter whether the web page is on a different web domain or it's stored on your computer's hard drive.

It's easy to understand the JavaScript that powers a page like this. The first task is to create the socket when the page first loads and wire up all its events:

```
var socket;

window.onload = function () {
  connect();
}

function connect() {
  socket = new WebSocket("ws://echo.websocket.org")

  // Listen to all the web socket events.
  socket.onopen = connectionOpen;
  socket.onmessage = messageReceived;
  socket.onerror = errorOccurred;
  socket.onopen = connectionClosed;
}
```

Here, the socket-connection code is moved into its own dedicated function, named `connect()`, so you can call it whenever you need it. That way, you can connect and disconnect as you please. You've already seen the disconnection code:

```
function connect() {
  socket.close();
}
```

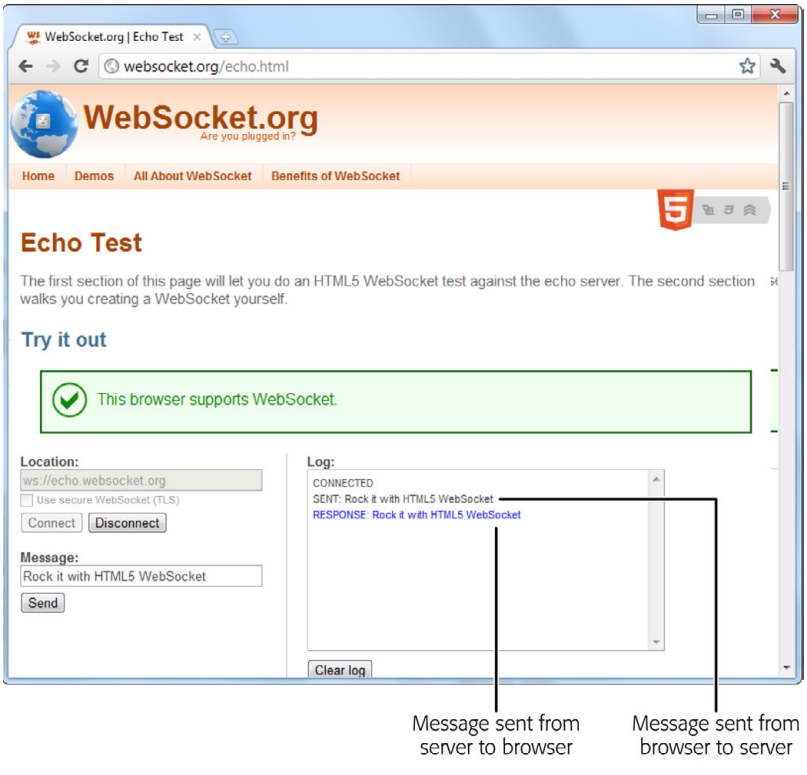


FIGURE 12-6

A web socket server that simply repeats what you say won't win any prizes in software innovation. However, it does let you see how easy it is to talk to a ready-made web socket server.

Incidentally, you can check the `readyState` property of your socket to determine whether it's open and ready to go (in which case `readyState` is 1), not yet established (0), in the process of closing (2), or closed (3).

The echo server has no real vocabulary to speak of. It treats all messages the same, as bits of text that it simply sends back to the page. Because of this simplicity, it's a simple job to send the current contents of a text box when the user clicks a button:

```
function sendMessage() {
  // Get the message data.
  var message = messageBox.value;

  // Send the message through the socket.
  socket.send(message);

  // Let the user know what just happened.
  messageLog.innerHTML += "<br>" + "Sent: " + message;
}
```

And it's just as easy to take the messages you receive and insert them into the page:

```
function messageReceived(e) {
  messageLog.innerHTML += "<br>" + "Message received: " + e.data;
}
```

NOTE If you want to look at a slightly more exciting example, check out the chat free-for-all at *http://html5demos.com/web-socket*. Log in to this simple web socket server, send a message, and everyone gets it immediately.

POWER USERS' CLINIC

Web Socket Servers

To actually run a practical example of your own, you need a web socket server that your web page can talk to. And although the server code that's required—which is several dozen lines at least—is beyond the scope of this chapter, there are plenty of places where you can find a test server. Here are some of your many options:

- **PHP.** This simple and slightly crude code project gives you a good starting point for building a web socket server with PHP. Get it at *http://code.google.com/p/phpwebsocket*.
- **Ruby.** There's more than one sample web socket server for Ruby, but this one that uses the Event-Machine model is popular. See *http://github.com/igrigorik/em-websocket*.
- **Python.** This Apache extension adds a socket server using Python. Get it at *http://code.google.com/p/pywebsocket*.
- **.NET.** Simple, it isn't. But this comprehensive project contains a complete web socket server that uses Microsoft's .NET platform and the C# language. Download it from *http://superwebsocket.codeplex.com*.
- **Java.** Similar in scope to the .NET project described above, this web socket server is pure Java. See *http://jwebsocket.org*.
- **node.JS.** Depending on who you ask, node.JS—a web server that runs JavaScript code—is either the next big thing in the making or an overgrown testing tool. Either way, you can get a web socket server that runs on it from *http://github.com/miksago/node-websocket-server*.
- **Kaazing.** Unlike the other items in this list, Kaazing doesn't provide the code for a web socket server. Instead, it provides a mature web socket server that you can license for your website. Adventurous developers who want to go it alone won't be interested. But it may make sense for less ambitious websites, especially considering the built-in fallback support in its client libraries (which try the HTML5 web socket standard first, then attempt to use Flash, or finally fall back to pure JavaScript polling). Learn more at *http://kaazing.com/products/html5-edition.html*.

CHAPTER

13

Geolocation, Web Workers, and History Management

By now, you know all about the key themes of HTML5. You've used it to write more meaningful and better-structured markup. You've seen its rich graphical features, like video and dynamic drawing. And you've used it to create self-sufficient, JavaScript-powered pages that can work even without a web connection.

In this chapter, you'll tackle three features that have escaped your attention so far. As with much of what you've already learned, these features extend the capabilities of what a web page can do—once you add a sprinkling of JavaScript code. Here's what awaits:

- **Geolocation.** Although it's often discussed as part of HTML5, geolocation is actually a separate standard that's never been in the hands of the WHATWG (page 5). Using geolocation, you can grab hold of a single piece of information: the geographic coordinates that pinpoint a web visitor's current location.
- **Web workers.** As web developers make smarter pages that run more JavaScript, it becomes more important to run certain tasks in the background, quietly, unobtrusively, and over long periods of time. You *could* use timers and other tricks. But the web workers feature provides a far more convenient solution for performing background work.
- **Session history.** In the old days of the Web, a page did one thing only: display stuff. As a result, people spent plenty of time clicking links to get from one document to another. But today, a JavaScript-fueled page can load content from another page without triggering a full page refresh. In this way, JavaScript creates a more seamless viewing experience. However, it also introduces a few

wrinkles, like the challenge of keeping the browser URL synchronized with the current content. Web developers use plenty of advanced techniques to keep things in order, and now HTML5 adds a session history tool that can help.

NOTE As you explore these last three features, you'll get a still better idea of the scope of what is now called HTML5. What started as a few good ideas wedged into an overly ambitious standard has grown to encompass a grab bag of new features that tackle a range of different problems, with just a few core concepts (like semantics, JavaScript, and CSS3) to hold it all together.

Geolocation

Geolocation is a feature that lets you find out where in the world your visitors are. And that doesn't just mean what country or city a person's in. The geolocation feature can often narrow someone's position down to a city block, or even determine the exact coordinates of someone who's strolling around with a smartphone.

NOTE Most of the new JavaScript features you've seen in this book were part of the original HTML5 specification and were split off when it was handed over to the W3C. But geolocation isn't like that—it was never part of HTML5. Instead, it simply reached maturity around the same time. However, almost everyone now lumps them together as part of the wave of future web technologies.

How Geolocation Works

Geolocation raises quite a lot of questions in people who ordinarily aren't paranoid. Like, how does a piece of software know I'm hanging out at the local coffee shop? Is there some hidden code that's tracking my every move? And who's in that white van parked outside?

Fortunately, geolocation is a lot less Big Brotherish than it seems. That's because even if a browser can figure out your position, it won't tell a website unless you give it explicit permission (see Figure 13-1).

To figure out a person's location, the browser enlists the help of a *location provider*—for example, on Firefox that's Google Location Services. This location provider has the tough job of finding the location, and it can use several different strategies to do it.

For a desktop computer with a fixed (not wireless) Internet connection, the science is simple but imprecise. When someone goes online, her traffic is funneled from her computer or local network through a cable, telephone wire, or (horrors) dial-up connection, until it reaches a high-powered piece of network hardware that brings it onto the Internet. That piece of hardware has a unique *IP address*, a numeric code that establishes its public identity to other computers. It also has a postal address in the real world.

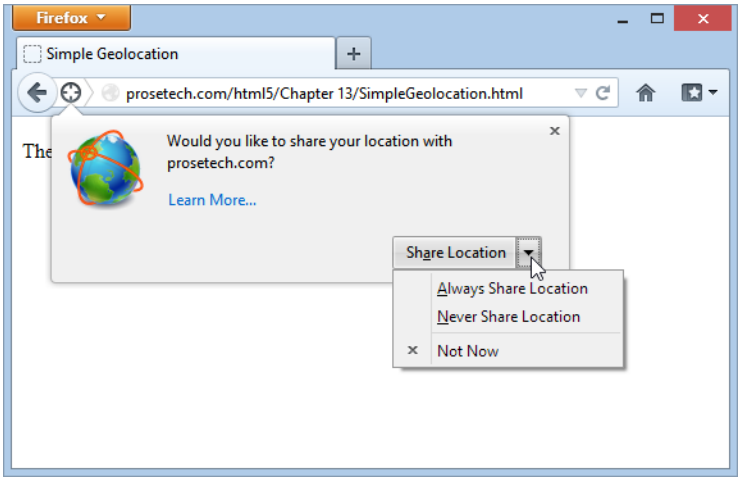


FIGURE 13-1

Here a web page wants location data, and Firefox asks whether you want to allow it just this once (click Share Location), to allow it every time (Always Share), or never to allow it (Never Share). This behavior isn't just Firefox being polite; the geolocation standard makes it an official rule to get user permission for every website that wants location data.

NOTE If you have some networking experience, you already know that your computer has its own IP address, like every computer on a network. However, this IP address is your own private one whose purpose is to separate your computer from any other devices that are sharing your connection (like the netbook in your kitchen or the tablet in your knapsack). Geolocation doesn't use that IP address.

The location provider combines these two features. First, it figures out the IP address you're connecting through, and then it pinpoints the home of the router that uses it. Because of this indirection, geolocation won't be spot-on when you're using a desktop computer. For example, if you surf from a computer on the west side of Chicago, you might find that your traffic is being funneled through a router that's several miles closer to downtown. Still, even an imprecise result like this is often useful. For example, if you're looking for nearby pizza stores in a mapping tool, you can quickly skip over to the area you're really interested in—your home neighborhood—even if you start a certain distance away.

NOTE The IP address technique is the roughest form of geolocation. If there's a better source of location data, the location provider will use that instead.

If you're using a laptop or a mobile device with a wireless connection, a location provider can look for nearby wireless access points. Ideally, the location provider consults a giant database to figure out the exact location of these access points and then uses that information to triangulate your location.

If you're using a web-enabled phone, the location provider provides a similar triangulation process, but it uses the signals from different cellphone towers. This quick, relatively effective procedure usually gets your position down to less than a kilometer. (More industrialized areas—like downtown city cores—have more cellphone towers, which results in more precise geolocation.)

Finally, many mobile devices also have dedicated GPS hardware, which uses satellite signals to pin your location down to just a few meters. The drawback is that GPS is a bit slower and draws more battery power. It also doesn't work as well in built-up cities, where tall buildings can obscure the signals. As you'll see, it's up to you whether you want to request a high-precision location using GPS, if it's available (page 409).

And of course, other techniques are possible. Nothing stops a location provider from relying on different information, like an RFID chip, nearby Bluetooth devices, a cookie set by a mapping website like Google Maps, and so on.

TIP You may also be able to change your starting position by using another tool. For example, Chrome fans can use a browser plug-in named Manual Geolocation (*http://tinyurl.com/manual-geo*) to set the position that Chrome should report when you browse a website that uses geolocation. You can even use this technique to fake your address—for example, to pretend your computer in Iowa is actually surfing in from the Netherlands. This trick isn't for espionage only—it can also be a useful debugging trick when you're testing a location-aware web app.

The takeaway is this: No matter how you connect to the Internet—even if you're sitting at a desktop computer—geolocation can get somewhere near you. And if you're using a device that gets a cellphone signal or has a GPS chip, the geolocation coordinates will be scarily accurate.

UP TO SPEED

How You Can Use Geolocation

Once you've answered the big question—how does geolocation work?—you need to resolve another one—namely, why should you use it?

The key point to understand is that geolocation tells your code the approximate geographic coordinates of a person—and that's it. You need to combine this simple but essential information with more detailed location data. This data could be provided by your web server (typically fetched out of a huge server-side database) or another geographic web service (like Google Maps).

For example, if you're a big business with a physical presence in the real world, you might compare the user's position with the coordinates of your different locations. You could then determine which location is closest. Or, if you're building some sort of social networking tool, you might plot the information of a group of people to show them how close they are to one another. Or you might take someone else's location data and use that to provide a service for your visitors, like hunting down the nearest chocolate store, or finding the closest clean toilet in Brooklyn. Either way, the geolocation coordinates of the visitor become important only when they're combined with more geographic data.

Although other businesses' mapping and geographic services are outside the scope of this chapter, you'll get the chance to try out an example with Google Maps on page 409.

Finding a Visitor's Coordinates

The geolocation feature is strikingly simple. It consists of three methods that are packed into the `navigator.geolocation` object: `getCurrentPosition()`, `watchPosition()`, and `clearWatch()`.

NOTE If you aren't already familiar with the navigator object, it's a relatively minor part of JavaScript, with a few properties that tell you about the current browser and its capabilities. The most useful of these is `navigator.userAgent`, which provides an all-in-one string that details the browser, its version number, and the operating system on which it's running.

To get a web visitor's location, you call `getCurrentPosition()`. Of course, the location-finding process isn't instantaneous, and no browser wants to lock up a page while it's waiting for location data. For that reason, the `getCurrentPosition()` method is asynchronous—it carries on immediately, without stalling your code. When the geolocation process is finished, it triggers another piece of code to handle the results.

You might assume that geolocation uses an event to tell you when it's done, in much the same way that you react when an image has been loaded or a text file has been read. But JavaScript is nothing if not inconsistent. Instead, when you call `getCurrentPosition()` you supply the *completion function*.

Here's an example:

```
navigator.geolocation.getCurrentPosition(
  function(position) {
    alert("You were last spotted at (" + position.coords.latitude +
     "," + position.coords.longitude + ")");
  }
);
```

When this code runs, it calls `getCurrentPosition()` and passes in a function. Once the browser determines the location, it triggers that function, which shows a message box. Figure 13-2 shows the result in Internet Explorer.

To keep your code clear and organized, you probably won't define your completion function right inside the `getCurrentPosition()` call (as done in this example). Instead, you can put it in a separate, named function:

```
function geolocationSuccess(position) {
  alert("You were last spotted at (" + position.coords.latitude +
   "," + position.coords.longitude + ")");
}
```

Then you can point to it when you call `getCurrentLocation()`:

```
navigator.geolocation.getCurrentPosition(geolocationSuccess);
```

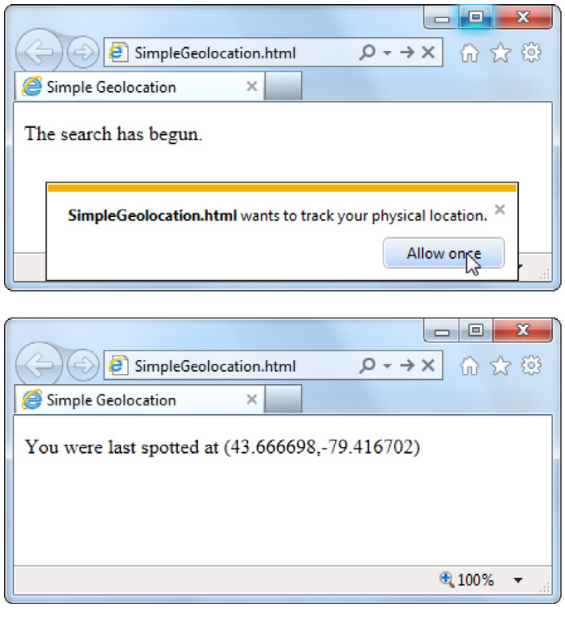


FIGURE 13-2

Top: First, you need to agree to let the browser tell the web server about your position.

Bottom: The results are in—your coordinates on the globe.

Remember, you need to use a browser that supports geolocation and let the web page track you. Also, it's a good idea to upload your page to a test server before trying it out. Otherwise, you'll see some quirks (for example, geolocation error-handling won't work) and some browsers will fail to detect your position altogether (like Chrome).

If you're wondering, "What good are geographic coordinates to me?" you've asked a good question. You'll explore how you can use the geolocation data shortly (page 409). But first, you should understand how to catch errors and configure a few geolocation settings.

Dealing with Errors

Geolocation doesn't run so smoothly if the visitor opts out and decides not to share the location data with your page. In the current example, the completion function won't be called at all, and your page won't have any way to tell whether the browser is still trying to dig up the data or has run into an error. To deal with this sort of situation, you supply two functions when you call `getCurrentLocation()`. The first function is called if your page meets with success, while the second is called if your geolocation attempt ends in failure.

POWER USERS' CLINIC

Finding Out the Accuracy of a Geolocation Guess

When the `getCurrentPosition()` method meets with success, your code gets a position object that has two properties: `timestamp` (which records when the geolocation was performed) and `coords` (which indicates the geographic coordinates).

As you've seen, the `coords` object gives you the latitude and longitude—the coordinates that pin down your position on the globe. However, the `coords` object bundles up a bit more information that you haven't seen yet. For example, there are more specialized `altitude`, `heading`, and `speed` properties, none of which are currently supported by any browser.

More interesting is the `accuracy` property, which tells you how precise the geolocation information is, in meters. (Somewhat confusingly, that means the value of the `accuracy` property increases as the accuracy of the location data decreases.) For example, an `accuracy` of 2,135 meters converts to about 1.3 miles, meaning the geolocation coordinates have pinpointed the current visitor's position within that distance. To visualize this, imagine a circle with the center at the geolocation coordinates and a radius of 1.3 miles. Odds are the visitor is somewhere in that circle.

The `accuracy` property is useful for identifying bad geolocation results. For example, if you get an `accuracy` result that's tens of thousands of meters, then the location data isn't reliable:

```
if (position.coords.accuracy > 50000) {
  results.innerHTML  =
   "This guess is all over the map.";
}
```

At this point, you might want to warn the user or offer him the chance to enter the right position information himself.

Here's an example that uses both a completion function and an error function:

```
// Store the element where the page displays the result.
var results;

window.onload = function() {
  results = document.getElementById("results");

  // If geolocation is available, try to get the visitor's position.
  if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(
     geolocationSuccess, geolocationFailure
    );
    results.innerHTML = "The search has begun.";
  }
  else {
    results.innerHTML = "This browser doesn't support geolocation.";
  }
};
```

```
function geolocationSuccess(position) {
  results.innerHTML = "You were last spotted at (" +
   position.coords.latitude + "," + position.coords.longitude + ")";
}

function geolocationFailure(positionError) {
  results.innerHTML = "Geolocation failed.";
}
```

When the error function is called, the browser hands it an error object with two properties: code (a numeric code that classifies it as one of four types of problems) and message (which provides a short text message reporting the problem). Generally, the message is intended for testing, and your code will use the error code to decide how it should handle the issue.

Here's a revised error function that checks all possible error code values:

```
function geolocationFailure(positionError) {
  if (positionError.code == 1) {
    results.innerHTML =
     "You decided not to share, but that's OK. We won't ask again.";
  }
  else if (positionError.code == 2) {
    results.innerHTML =
     "The network is down or the positioning service can't be reached.";
  }
  else if (positionError.code == 3) {
    results.innerHTML =
     "The attempt timed out before it could get the location data.";
  }
  else {
    results.innerHTML =
     "This the mystery error. We don't know what happened.";
  }
}
```

NOTE If you're running the test web page from your computer (not a real web server), the error function won't be triggered when you decline to share your location.

Setting Geolocation Options

So far, you've seen how to call getCurrentLocation() with two arguments: the success function and the failure function. You can also supply a third argument, which is an object that sets certain geolocation options.

Currently, there are three options you can set, and each one corresponds to a different property on the geolocation options object. You can set just one or any combination. Here's an example that sets one, named enableHighAccuracy:

```
navigator.geolocation.getCurrentPosition(geolocationSuccess,
 geolocationFailure, {enableHighAccuracy: true});
```

And here's an example that sets all three:

```
navigator.geolocation.getCurrentPosition(
 geolocationSuccess, geolocationFailure, {enableHighAccuracy: true,
 timeout: 10000,
 maximumAge: 60000}
);
```

Both of these examples supply the geolocation options using a JavaScript object literal. This technique works perfectly as long as you use the right property names, such as `enableHighAccuracy` and `timeout`, because these are the properties that the `getCurrentPosition()` method is expecting. (If this code still looks a bit weird to you, check out the more detailed object introduction on page 468 in Appendix B, "JavaScript: The Brains of Your Page.")

So what do these properties mean? The `enableHighAccuracy` property opts into high-precision GPS-based location detection, if the device supports it (and the user allows it). Don't choose this option unless you need exact coordinates, because it can draw serious battery juice and may take more time. The default for `enableHighAccuracy`, should you choose not to set it, is `false`.

The `timeout` property sets the amount of time your page is willing to wait for location data before throwing in the towel. The `timeout` is in milliseconds, so a value of 10,000 milliseconds means a maximum wait of 10 seconds. The countdown begins *after* the user agrees to share the location data. By default, `timeout` is 0, meaning the page will wait indefinitely, without ever triggering the timeout error.

The `maximumAge` property lets you use cached location data. For example, if you set `maximumAge` to 60,000 milliseconds, you'll accept a previous value that's up to a minute old. This saves the effort of repeated geolocation calls, but it also means your results will be less accurate for a person on the move. By default, `maximumAge` is 0, meaning cached location data is never used. (You can also use a special value of `Infinity`, which means use any cached location data, no matter how old it is.)

Showing a Map

Being able to grab someone's geographic coordinates is a neat trick. But the novelty wears off fast unless you have something useful to do with that information. Hardcore geo-junkies know that there's a treasure trove of location information out there. (Often, the problem is taking this information and converting it to a form that's useful to your web application.) There are also several web-based mapping services, the king of which is Google Maps. In fact, good estimates suggest that Google Maps is the most heavily used web application service, for *any* purpose.

Using Google Maps, you can create a map for any portion of the world, at any size you want. You can control how your visitors interact with that map, generate driving instructions, and—most usefully—overlay your own custom data points on that

map. For example, a Google Maps–fortified page can show visitors your business locations or flag interesting sights in a Manhattan walking tour. To get started with Google Maps, check out the documentation at *http://tinyurl.com/maps-docs*.

NOTE Google Maps is free to use, even for commercial websites, provided you aren't charging people to access your site. (And if you are, Google has a premium mapping service you can pay to use.) Currently, Google Maps does not show ads, although the Google Maps license terms explicitly reserve the right to do that in the future.

Figure 13-3 shows a revised version of the geolocation page. Once it grabs the current user's coordinates, it shows that position in a map.

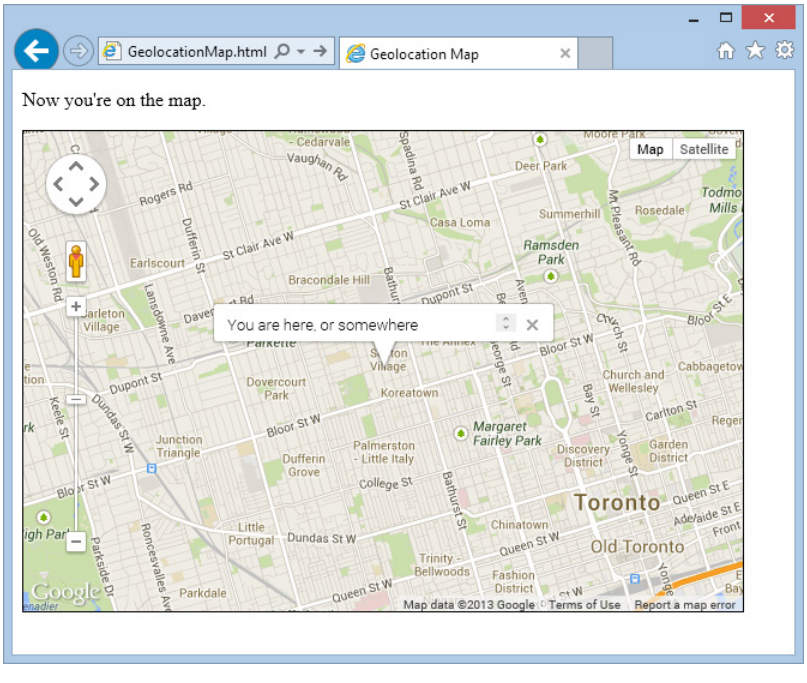


FIGURE 13-3
Geolocation and Google Maps make a potent combination. They let you generate a map for any position, with just a few extra lines of JavaScript.

Creating this page is easy. First, you need a link to the scripts that power the Google Maps API. Place this before any script blocks that use the mapping functionality:

```
<head>
  <meta charset="utf-8">
  <title>Geolocation Map</title>
  <script src="http://maps.google.com/maps/api/js?sensor=true"></script>
 ...
</head>
```

Next, you need a element that will hold the dynamically generated map. Give it a unique ID for easy reference:

```
<body>
  <p id="results">Where do you live?</p>
  <div id="mapSurface"></div>
</body>
```

You can then use a style sheet rule to set the size of your map:

```
#mapSurface {
  width: 600px;
  height: 400px;
  border: solid 1px black;
}
```

Now you're ready to start using Google Maps. The first job is to create the map surface. This example creates the map when the page loads, so that you can use it in the success or failure function. (After all, failure doesn't mean the visitor can't use the mapping feature in your page; it just means that you can't determine that visitor's current location. You'll probably still want to show the map, but just default to a different starting point.)

Here's the code that runs when the page loads. It creates the map and then starts a geolocation attempt:

```
var results;
var map;

window.onload = function() {
  results = document.getElementById("results");

  // Set some map options. This example sets the starting zoom level and the
  // map type, but see the Google Maps documentation for all your options.
  var mapOptions = {
    zoom: 13,
    mapTypeId: google.maps.MapTypeId.ROADMAP };

  // Create the map, with these options.
  map = new google.maps.Map(document.getElementById("mapSurface"), mapOptions);

  // Try to find the visitor's position.
  if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(geolocationSuccess,
     geolocationFailure);
    results.innerHTML = "The search has begun.";
  }
  else {
    results.innerHTML = "This browser doesn't support geolocation.";
```

```
    goToDefaultLocation();
  }
};
```

Even after you've created the map with this code, you still won't see it in the page. That's because you haven't set a geographic position. To do that, you need to create a specific global *point* using the `LatLng` object. You can then place that point on the map with the map's `setCenter()` method. Here's the code that does that with the visitor's coordinates:

```
function geolocationSuccess(position) {
  // Turn the geolocation position into a LatLng object.
  location = new google.maps.LatLng(
   position.coords.latitude, position.coords.longitude);

  // Map that point.
  map.setCenter(location);
```

This code is sufficient for displaying a map, like the one in Figure 13-3. But you can also add adornments to that map, like other places or an info bubble. For the latter, you need to create an `InfoWindow` object. Here's the code that creates the info bubble shown in Figure 13-3:

```
  // Create the info bubble and set its text content and map coordinates.
  var infowindow = new google.maps.InfoWindow();
  infowindow.setContent("You are here, or somewhere thereabouts.");
  infowindow.setPosition(location);

  // Make the info bubble appear.
  infowindow.open(map);

  results.innerHTML = "Now you're on the map.";
}
```

Finally, if geolocation fails or isn't supported, you can carry out essentially the same process. Just use the hard-coded coordinates of a place you know:

```
function geolocationFailure(positionError) {
  ...
  goToDefaultLocation();
}

function goToDefaultLocation() {
  // This is the location of New York.
  var newYork = new google.maps.LatLng(40.69847, -73.95144);

  map.setCenter(newYork);
}
```

Monitoring a Visitor's Moves

All the examples you've used so far have relied on the `getCurrentPosition()` method, which is the heart of geolocation. However, the geolocation object has two more methods that allow you to track a visitor's position, so your page receives notifications as the location changes.

It all starts with the `watchPosition()` method, which looks strikingly similar to `getCurrentPosition()`. Like `getCurrentPosition()`, `watchPosition()` accepts three arguments: a success function (which is the only required detail), a failure function, and an options object:

```
navigator.geolocation.watchPosition(geolocationSuccess, geolocationFailure);
```

The difference between `getCurrentPosition()` and `watchPosition()` is that `watchPosition()` may trigger the success function multiple times—when it gets the location for the first time, and again whenever it detects a new position. (It's not in your control to set how often the device checks for a new position. All you need to know is that the device won't bother you if the position hasn't changed, but it will trigger the success function again if it has.) On a desktop computer, which never moves, the `getCurrentPosition()` and `watchPosition()` methods have exactly the same effect.

Unlike `getCurrentPosition()`, `watchPosition()` returns a number. You can hold onto this number and pass it in to `clearWatch()` to stop paying attention to location changes. Or you can ignore this step and keep receiving notifications until the visitor surfs to another page:

```
var watch = navigator.geolocation.watchPosition(geolocationSuccess,
 geolocationFailure);
...

navigator.geolocation.clearWatch(watch);
```

Browser Compatibility for Geolocation

The geolocation feature has good support in every modern browser, including mobile browsers. The only exception is old versions of Internet Explorer. Sadly, geolocation isn't available in IE 8 or IE 7. If you expect your audience to include people using older versions of Internet Explorer, you can polyfill the gap. There are a number of simple JavaScript libraries that solve the problem. Usually, they use the IP lookup technique described on page 403, which is the crudest form of geolocation. For example, the geolocation polyfill at *http://github.com/inexorabletash/polyfill* grabs the router's IP address and looks up its physical location in the database at *http://freegeoip.net*.

Alternatively, you can choose to pick a default starting position without trying to get the user's current location. Or, if you're using Google Maps, you can let the user pick a point from a map and then use those coordinates. The documentation for the Google Maps API is filled with examples like these—start with *http://tinyurl.com/qbmqdsq* for one example that intercepts clicks on a map.

Web Workers

Way back when JavaScript was first created, no one worried too much about performance. JavaScript was built to be a straightforward language for running small bits of script in a web page. JavaScript was a frill—a simplified scripting language for amateur programmers. It certainly wasn't meant to run anyone's business.

Fast-forward nearly 20 years, and JavaScript has taken over the Web. Developers use it to add interactivity to almost every sort of page, from games and mapping tools to shopping carts and fancy forms. But in many ways, the JavaScript language is still scrambling to catch up to its high status.

One example is the way JavaScript deals with big jobs that require hefty calculations. In most modern programming systems, work like this would happen quietly in the *background*, while the person using the application carried on, undisturbed. But in JavaScript, code always runs in the *foreground*. So any time-consuming piece of code will interrupt the user and freeze up the page until the job is done. Ignore this problem, and you'll wind up with some seriously annoyed, never-to-return visitors.

NOTE Crafty web developers have found some partial solutions to the JavaScript freeze-up problem. These involve splitting long-running tasks into small pieces and using `setInterval()` or `setTimeout()` to run one piece at a time. For certain types of tasks, this solution works well (for example, it's a practical way to animate a canvas, as demonstrated on page 301). But if you need to run a single, very long operation from start to finish, this technique adds complexity and confusion.

HTML5 introduces a better solution. It adds a dedicated object, called a *web worker*, that's designed to do background work. If you have a time-consuming job to polish off, you create a new web worker, supply it with your code, and start it on its way. While it works, you can communicate with it in a safe but limited way—by passing text messages.

A Time-Consuming Task

Before you can see the benefits of web workers, you need to find a suitable intensive piece of code. There's no point in using web workers for short tasks. But if you plan to run some CPU-taxing calculations that could tie up the web browser for more than a few seconds, web workers make all the difference. Consider, for example, the prime number searcher shown in Figure 13-4. Here, you can hunt for prime numbers that fall in a given range. The code is simple, but the task is *computationally difficult*, which means it could take some serious number-crunching time.

Clearly, this page can be improved with web workers. But before you get to that, you need to take a quick look through the existing markup and JavaScript code.

UP TO SPEED

Web Worker Safety Measures

JavaScript's web worker lets your code work in the background while something else takes place in the foreground. This brings up a well-known theme of modern programming: If an application can do two things at once, one of them has the potential to mess up the other.

The problem occurs when two different pieces of code fight over the same data, at the same time. For example, one piece of code may attempt to read some data, while another attempts to set it. Or both may attempt to set a variable at the same time, causing one change to be overwritten. Or two pieces of code may attempt to manipulate the same object in different ways, pushing it into an inconsistent state. The possible issues are endless, and they're notoriously difficult to discover and solve. Often, a multithreaded application (that's an application that uses several *threads* of independently executing code) works fine during testing. But when you start using it in the real world, maddeningly inconsistent errors appear.

Fortunately, you won't face these problems with JavaScript's web workers feature. That's because it doesn't let you share the same data between your web page and your web workers. You can *send* data from your web page to a web worker (or vice versa), but JavaScript automatically makes a copy of your data and sends that. That means there's no possible way for two threads to get hold of the same memory slot at the same time and cause subtle issues. Of course, this simplified model also restricts some of the things that web workers can do, but a minor reduction in capabilities is the cost of making sure ambitious programmers can't shoot themselves in the foot.

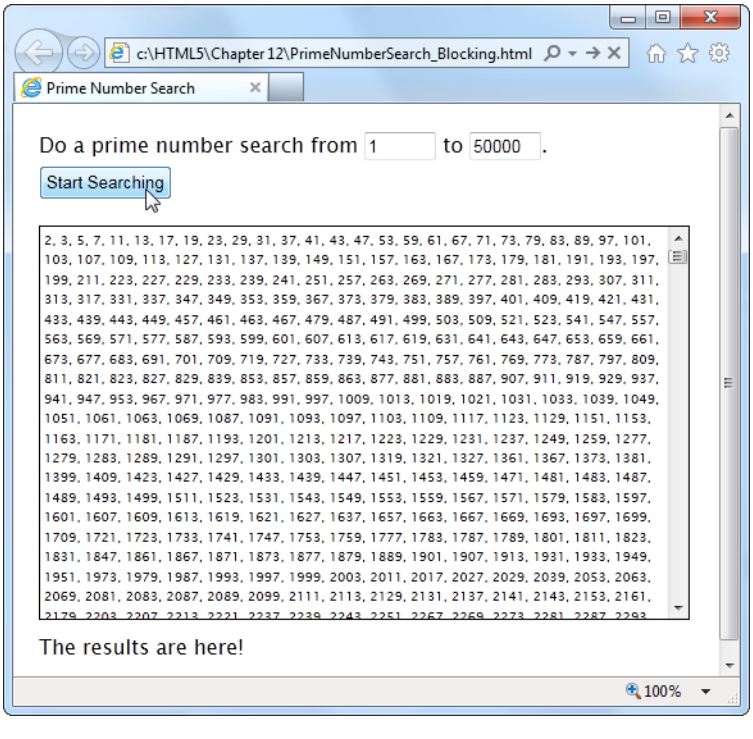


FIGURE 13-4

Pick your range and click the button to start the search. Pick a relatively narrow range (like this one, from 1 to 50,000), and the task completes in seconds, without inconveniencing anyone. But launch a broader search (say, from 1 to 500,000) and your page could become unresponsive for minutes or more. You won't be able to click, scroll, or interact with anything—and the browser may even give you a "long-running script" warning or gray out the entire page.

The markup is short and concise. The page uses two <input> controls, one for each text box. It also includes a button to start the search and two elements, one to hold the results and another to hold the status message underneath. Here's the complete markup from inside the <body> element:

```
<p>Do a prime number search from <input id="from" value="1"> to
 <input id="to" value="20000">.</p>
<button id="searchButton" onclick="doSearch()">Start Searching</button>

<div id="primeContainer">
</div>

<div id="status"></div>
```

One interesting detail is the styling of the element that holds the prime number list. It's given a fixed height and a maximum width, and the overflow and overflow-x properties work together to add a vertical scroll bar (but not a horizontal one):

```
#primeContainer {
  border: solid 1px black;
  margin-top: 20px;
  margin-bottom: 10px;
  padding: 3px;
  height: 300px;
  max-width: 500px;
  overflow: scroll;
  overflow-x: hidden;
  font-size: x-small;
}
```

The JavaScript code is a bit longer, but not much more complicated. It retrieves the numbers from the text boxes, starts the search, and then adds the prime number list to the page. It doesn't actually perform the mathematical operations that find the prime numbers—this is handled through a separate function, which is named findPrimes() and stored in a separate JavaScript file.

TIP You don't need to see the findPrimes() function to understand this example or web workers—all you need is a suitably long task. However, if you're curious to see the math that makes this page work, or if you just want to run a few prime number searches yourself, check out the full code on the try-out site at *http://prosetech.com/html5*.

Here's the complete code for the doSearch() function:

```
function doSearch() {
  // Get the numbers for the search range.
  var fromNumber = document.getElementById("from").value;
  var toNumber = document.getElementById("to").value;
```

```
    // Perform the prime search. (This is the time-consuming step.)
    var primes = findPrimes(fromNumber, toNumber);

    // Loop over the array of prime numbers, and paste them together into
    // one long piece of text.
    var primeList = "";
    for (var i=0; i<primes.length; i++) {
      primeList += primes[i];
      if (i != primes.length-1) primeList += ", ";
    }

    // Insert the prime number text into the page.
    var displayList = document.getElementById("primeContainer");
    displayList.innerHTML = primeList;

    // Update the status text to tell the user what just happened.
    var statusDisplay = document.getElementById("status");
    if (primeList.length == 0) {
      statusDisplay.innerHTML = "Search failed to find any results.";
    }
    else {
      statusDisplay.innerHTML = "The results are here!";
    }
  }
```

As you can see, the markup and code is short, simple, and to the point. Unfortunately, if you plug in a large search you'll find that it's also as slow and clunky as riding a golf cart up a steep hill.

Doing Work in the Background

The web worker feature revolves around a new object called the `Worker`. When you want to run something in the background, you create a new `Worker`, give it some code, and send it some data.

Here's an example that creates a new web worker that runs the code in the file named *PrimeWorker.js*:

```
var worker = new Worker("PrimeWorker.js");
```

The code that a worker runs is *always* stored in a separate JavaScript file. This design discourages newbie programmers from writing web worker code that attempts to use global variables or directly access elements on the page. Neither of these operations is possible.

NOTE Browsers enforce a strict separation between your web page and your web worker code. For example, there's no way for the code in *PrimeWorker.js* to write prime numbers into a `<div>` element. Instead, your worker code needs to send its data back to JavaScript code on the page, so the web page code can display the results.

Web pages and web workers communicate by exchanging messages. To send data to a worker, you call the worker's `postMessage()` method:

```
worker.postMessage(myData);
```

The worker then receives an `onMessage` event that provides a copy of the data. This is when it starts working.

Similarly, when your worker needs to talk back to the web page, it calls its own `postMessage()` method, along with some data, and the web page receives an `onMessage` event. Figure 13-5 shows this interaction close up.

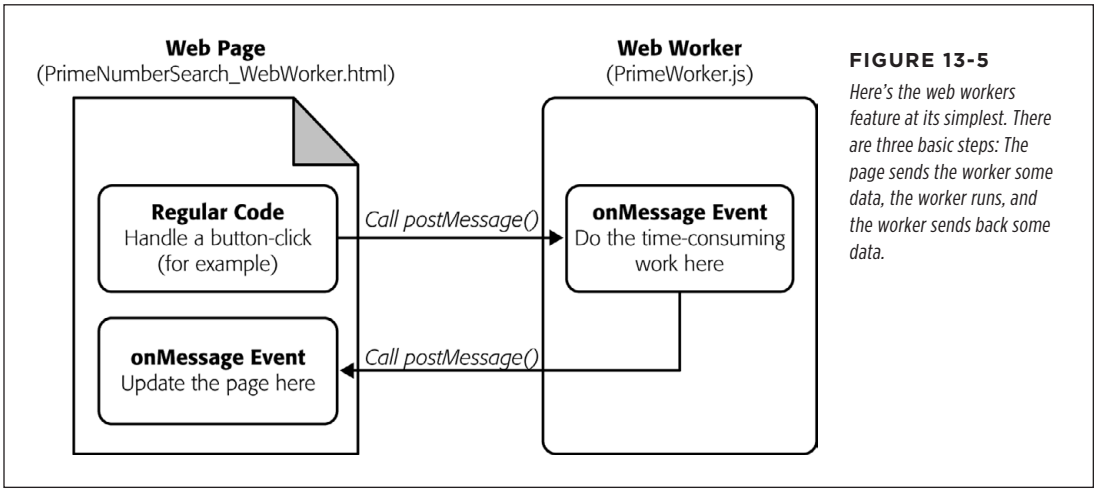


FIGURE 13-5
Here's the web workers feature at its simplest. There are three basic steps: The page sends the worker some data, the worker runs, and the worker sends back some data.

There's one more wrinkle to consider before you dive in. The `postMessage()` function allows only a single value. This fact is a stumbling block for the prime number cruncher, because it needs *two* pieces of data (the two numbers in the range). The solution is to package these two details into an object literal (see page 469). This code shows one example, which gives the object two properties (the first named `from`, and the second named `to`), and assigns values to both of them:

```
worker.postMessage(
 { from: 1,
  to: 20000 }
);
```

NOTE Incidentally, you can send virtually any object to a worker. Behind the scenes, the browser uses JSON (page 329) to convert your object to a harmless piece of text, duplicate it, and re-objectify it.

With these details in mind, you can revise the `doSearch()` function you saw earlier. Instead of performing the prime number search itself, the `doSearch()` function creates a worker and gets it to do the real job:

```
var worker;

function doSearch() {
  // Disable the button, so the user can't start more than one search
  // at the same time.
  searchButton.disabled = true;

  // Create the worker.
  worker = new Worker("PrimeWorker.js");

  // Hook up to the onMessage event, so you can receive messages
  // from the worker.
  worker.onmessage = receivedWorkerMessage;

  // Get the number range, and send it to the web worker.
  var fromNumber = document.getElementById("from").value;
  var toNumber = document.getElementById("to").value;

  worker.postMessage(
   { from: fromNumber,
    to: toNumber }
  );

  // Let the user know that things are on their way.
  statusDisplay.innerHTML = "A web worker is on the job ("+
   fromNumber + " to " + toNumber + ") ...";
}
```

Now the code in the *PrimeWorker.js* file springs into action. It receives the onMessage event, performs the search, and then posts a new message back to the page, with the prime list:

```
onmessage = function(event) {
  // The object that the web page sent is stored in the event.data property.
  var fromNumber = event.data.from;
  var toNumber = event.data.to;

  // Using that number range, perform the prime number search.
  var primes = findPrimes(fromNumber, toNumber);

  // Now the search is finished. Send back the results.
  postMessage(primes);
};

function findPrimes(fromNumber, toNumber) {
  // (The boring prime number calculations go in this function.)
}
```

When the worker calls postMessage(), it fires the onMessage event, which triggers this function in the web page:

```
function receivedWorkerMessage(event) {
  // Get the prime number list.
  var primes = event.data;

  // Copy the list to the page.
  ...

  // Allow more searches.
  searchButton.disabled = false;
}
```

You can now use the same code you saw earlier (page 417) to convert the array of prime numbers into a piece of text and insert that text into the web page.

Overall, the structure of the code has changed a bit, but the logic is mostly the same. The result, however, is dramatically different. Now, when a long prime number search is under way, the page remains responsive. You can scroll down, type in the text boxes, and select numbers in the list from the previous search. Other than the message at the bottom of the page, there's nothing to reveal that a web worker is plugging away in the background.

TIP Does your web worker need access to the code in another JavaScript file? There's a simple solution with the `importScripts()` function. For example, if you want to call functions from the *FindPrimes.js* file in *PrimeWorker.js*, just add this line of code before you do:

```
importScripts("FindPrimes.js");
```

GEM IN THE ROUGH

Running Web Workers Offline

If you take the time to prepare a complete web worker example and upload it to a web server, you can test your page complication-free. Of course, in the real world it's easier to try things out on your desktop and run them straight from your hard drive. This method works in Firefox and current versions of Internet Explorer, but not in Chrome—unless you take additional steps.

If you're running your web worker page from a local file, Chrome will fail unless you start it with the `--allow-file-access-from-files` parameter. In Windows, you accomplish this by changing your Chrome shortcut (right-click it and choose Properties) or by creating a new Chrome shortcut and customizing that. Either way, you need to look in the Target box and tack the parameter onto the end of the command line. For example, you would change a shortcut like this:

```
C:\Users\billcruft\ ... \chrome.exe
```

to this:

```
C:\Users\billcruft\ ... \chrome.exe
--allow-file-access-from-files
```

Now you can try out your web worker code in the comfort and privacy of your own computer, no uploads required.

Handling Worker Errors

As you've learned, the `postMessage()` method is the key to communicating with web workers. However, there's one more way that a web worker can notify your web page—with the `onerror` event that signals an error:

```
worker.onerror = workerError;
```

Now, if some dodgy script or invalid data causes an error in your background code, the error details are packaged up and sent back to the page. Here's some web page code that simply displays the text of the error message:

```
function workerError(error) {
  statusDisplay.innerHTML = error.message;
}
```

Along with the `message` property, the error object also includes a `lineno` and `filename` property, which report the line number and file name where the error occurred.

Canceling a Background Task

Now that you've built a basic web worker example, it's time to add a few refinements. First is cancellation support, which lets your page shut down a worker in mid-calculation.

There are two ways to stop a worker. First, a worker can stop itself by calling `close()`. More commonly, the page that created the worker will shut it down by calling the worker's `terminate()` method. For example, here's the code you can use to power a straightforward cancel button:

```
function cancelSearch() {
  worker.terminate();
  statusDisplay.innerHTML = "";
  searchButton.disabled = false;
}
```

Click this button to stop the current search and re-enable the search button. Just remember that once a worker is stopped in this way, you can't send any more messages, and it can't be used to do any more operations. To perform a new search, you need to create a new worker object. (The current example does this already, so it works perfectly.)

Passing More Complex Messages

The last trick you'll learn to do with a web worker is return progress information. Figure 13-6 shows a revised version of the web worker page that adds this feature.

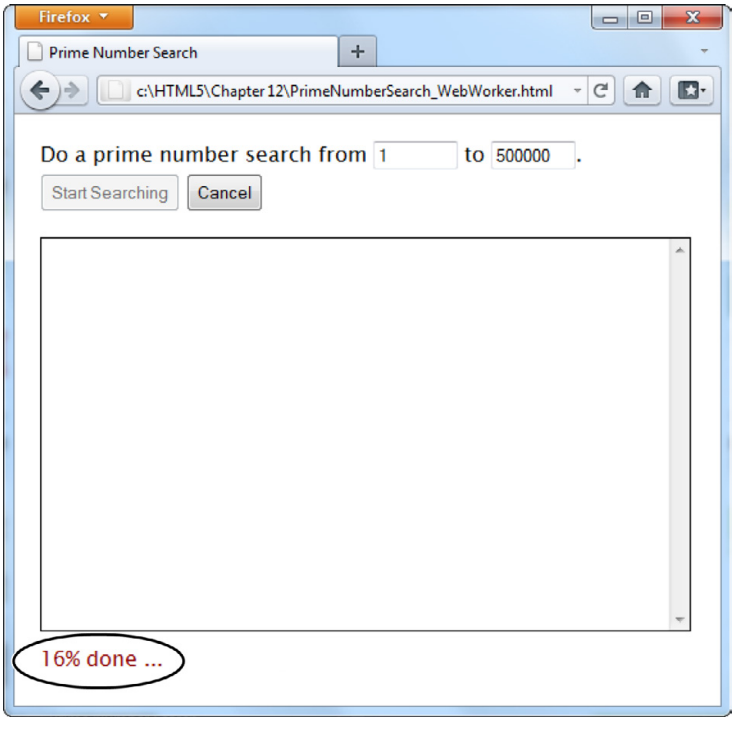


FIGURE 13-6

As the prime number search is under way, the search updates the status display to tell you how close the task is to completion. For a fancier display, you could use a color-filling progress bar like the one on page 165.

To build the progress display, the web worker needs to send the progress percentage to the page while it works. But as you already know, web workers have just one way to talk to the pages that own them—with the `postMessage()` method. So to create this example, the web worker needs to send *two* types of messages: progress notifications (while the work is under way) and the prime number list (when the work is finished). The trick is making the difference between these two messages clear, so the `onMessage` event handler in the page can easily distinguish between the two types.

The best approach is to add a bit of extra information to the message. For example, when the web worker sends progress information, it can slap the text label "Progress" on its message. And when the web worker sends the prime number list, it can add the label "PrimeList."

To bundle all the information you want together in one message, you need to create an object literal. This is the same technique the web page used to send the number range data to the web worker. The extra piece of information is the text that describes the type of message, which is placed in a property called `messageType` in this example. The actual data goes in a second property, named `data`.

Here's how you would rewrite the web worker code to add a message type to the prime number list:

```
onmessage = function(event) {
  // Perform the prime number search.
  var primes = findPrimes(event.data.from, event.data.to);

  // Send back the results.
  postMessage(
   {messageType: "PrimeList", data: primes}
  );
};
```

The code in the `findPrimes()` function also uses the `postMessage()` method to send a message back to the web page. It uses the same two properties—`messageType` and `data`. But now the `messageType` indicates that the message is a progress notification, and `data` holds the progress percentage:

```
function findPrimes(fromNumber, toNumber) {
  ...

  // Calculate the progress percentage.
  var progress = Math.round(i/list.length*100);

  // Only send a progress update if the progress has changed at least 1%.
  if (progress != previousProgress) {
    postMessage(
      {messageType: "Progress", data: progress}
    );
    previousProgress = progress;
  }
  ...
}
```

When the page receives a message, it needs to start by checking the `messageType` property to determine what sort of message it has just received. If it's a prime list, then the results are shown in the page. If it's a progress notification, then the progress text is updated:

```
function receivedWorkerMessage(event) {
  var message = event.data;

  if (message.messageType == "PrimeList") {
    var primes = message.data;

    // Display the prime list. This code is the same as before.
    ...
  }
```

```
    else if (message.messageType == "Progress") {
      // Report the current progress.
      statusDisplay.innerHTML = message.data + "% done ...";
    }
  }
```

NOTE There's another way to design this page. You could get the worker to call `postMessage()` every time it finds a prime number. The web page would then add each prime number to the list and show it in the page immediately. This approach has the advantage of showing results as they arrive. However, it also has the drawback of continually interrupting the page (because the web worker will find prime numbers quite quickly). The ideal design depends on the nature of your task—how long it takes to complete, whether partial results are useful, how quickly each partial result is calculated, and so on.

POWER USERS' CLINIC

More Ways to Use a Web Worker

The prime number search uses web workers in the most straightforward way possible—to perform one well-defined task. Every time the search is started, the page creates a new web worker. That web worker is responsible for a single task. It receives a single message and sends a single message back.

Your pages don't need to be this simple. Here are a few examples of how you can extend your web-worker designs to do more complicated things:

- **Reuse a web worker for multiple jobs.** When a worker finishes its work and reaches the end of the `onMessage` event handler, it doesn't die. It simply goes idle and waits quietly. If you send the worker another message, it springs back to life and does the work.
- **Create multiple web workers.** Your page doesn't need to stick to one worker. For example, imagine you want to let a visitor launch several prime number searches at a time. You could create a new web worker for each search and keep track of all your workers in an array. Each time a web worker responds with its list of prime numbers, you add that to the page, taking care not to overwrite any other worker's result. (However, some words of caution are in order. Web workers have a relatively high overhead, and running a dozen at once could swamp the computer with work.)
- **Create web workers inside a web worker.** A web worker can start its own web workers, send them messages, and receive their messages back. This technique is useful for complex computational tasks that require recursion, like calculating the Fibonacci sequence.
- **Download data with a web worker.** Web workers can use the `XMLHttpRequest` object (page 377) to grab new pages or to send requests to a web service. When they get the information they need, they can call `postMessage()` to send it up to the page.
- **Do periodic tasks with a web worker.** Web workers can use the `setTimeout()` and `setInterval()` functions, just like ordinary web pages. For example, you might create a web worker that checks a website for new data every minute.

Browser Compatibility for Web Workers

The web worker feature isn't supported as broadly as the geolocation feature. Table 13-1 shows which browsers you can rely on.

TABLE 13-1 *Browser support for web workers*

	IE	FIREFOX	CHROME	SAFARI	OPERA	SAFARI IOS	CHROME FOR ANDROID
Minimum version	10	3.5	3	4	10.6	5	29

So what can you do if you face a browser that doesn't have web worker support? The easiest option is to simply do the same work in the foreground:

```
if (window.Worker) {
  // Web workers are supported.
  // So why not create a web worker
  // and start it?
} else {
  // Web workers aren't available.
  // You can just call the prime search
  // function, and wait.
}
```

This approach doesn't force you to write any extra code, because the prime-number-searching function is already written, and you can call it with or without a web worker. However, if you have a long task, this approach could lock up the browser for a bit. So if you use this strategy, it's wise to warn the user (for example, with a message on the page), that he's using a less-supported browser and the calculation process may temporarily freeze up the page.

An alternate (but more tedious) approach is to try to fake a background job using the `setInterval()` or `setTimeout()` methods. For example, you could write some code that tests just a few numbers every interval. Some polyfills even attempt to add this sort of system (see the Web Workers section on *http://tinyurl.com/polyfills*), but this approach gets messy quickly.

History Management

Session history is an HTML5 add-on that extends the capabilities of the JavaScript history object. This sounds simple, but the trick is knowing when and why you should use it.

If you've never noticed the history object before, don't be alarmed. Up until now, it's had very little to offer. In fact, the traditional history object has just one property and three basic methods. The property is *length*, and it tells you how many entries are

in the browser's History list (the list of recently visited web pages that the browser maintains as you skip from page to page across the Web). Here's an example that uses it:

```
alert("You have " + history.length +
 " pages in your browser's history list.");
```

The most useful history method is `back()`. It lets you send a visitor one step back in the browsing history:

```
history.back();
```

This method has the same effect as if the visitor clicked the browser's Back button. Similarly, you can use the `forward()` method to step forward, or the `go()` method to move a specified number of steps backward or forward.

All this adds up to relatively little, unless you want to design your own custom Back and Forward buttons on a web page. But HTML5 adds a bit more functionality, which you can put to far more ambitious purposes. The centerpiece is the `pushState()` method, which lets you change the URL in the browser window without triggering a page refresh. This comes in handy in a specific scenario—namely, when you're building dynamic pages that quietly load new content and seamlessly update themselves. In this situation, the page's URL and the page's content can become out of sync. For example, if a page loads content from another page, the first page's URL stays in the browser's address box, which can cause all sorts of bookmarking confusion. Session history gives you a way to patch this hole.

If you're having a bit of trouble visualizing this scenario, hold on. In the next section, you'll see a page that's a perfect candidate for session history.

The URL Problem

In the previous chapter, you considered a page about Chinese tourism that had a built-in slideshow (page 382). Using the Previous and Next buttons on this page, the viewer could load different slides. But the best part about this example is that each slide was loaded quietly and unobtrusively and without reloading the page, thanks to the trusty `XMLHttpRequest` object.

Pages that include dynamic content and use this sort of design have a well-known limitation. Even though the page changes when it loads in new content, the URL stays the same in the browser's address bar (Figure 13-7).

To understand the problem, imagine that Joe reads the article shown in Figure 13-7, looks at the different sights, and is excited by the wishing tree in the fifth slide. Joe bookmarks the page, sends the URL to his friend Claire via email, and promotes it to the whole world with a Twitter message ("Throwing paper into a tree beats dropping coins in a fountain. Check it out at http://..."). The problem is that when Joe returns to his bookmark, or when Claire clicks the link in the email, or when any of Joe's followers visit the link in the tweet, they all end up at the first slide. They may not have the patience to click through to the fifth slide, or they may not even know

where it is. And this problem grows worse if there are more than just five slides—for example, a Flickr photo stream could have dozens or hundreds of pictures.

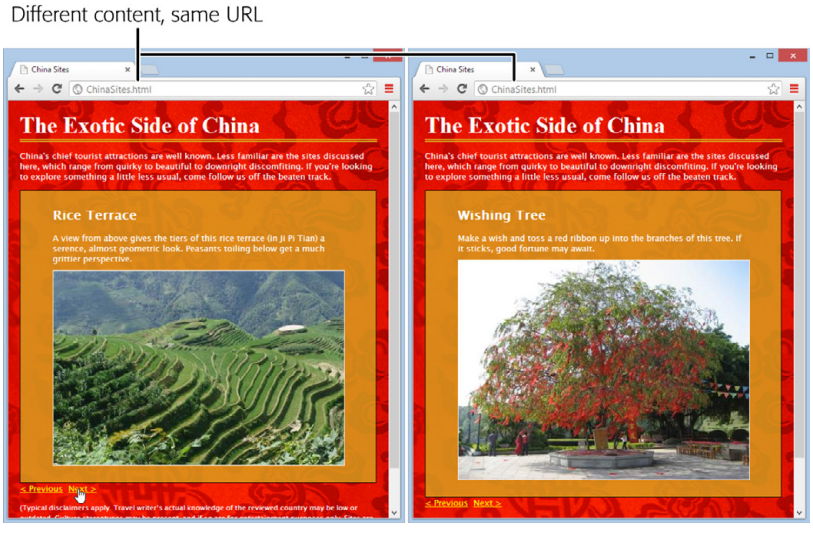


FIGURE 13-7
Here are two versions of the ChinaSites.html page, with different slides loaded. In both pages, the URL stays the same (it's ChinaSites.html).

The Old Solution: Hashbang URLs

To deal with this problem, some web pages tack extra information onto the end of the URL. Just a few years ago, leading sites like Facebook, Twitter, and Google fell over themselves in excitement to implement a controversial strategy called the *hashbang* technique. To use the hashbang technique, you add the characters #! at the end of any URL, followed by some additional information. Here's an example:

```
http://jjtraveltales.com/ChinaSites.html#!/Slide5
```

The reason the hashbang approach works is because browsers treat everything after the # character as the *fragment* portion of a URL. So in the example shown here, the web browser knows that you're still referring to the same *ChinaSites.html* page, just with an extra fragment added to the end.

On the other hand, consider what happens if your JavaScript code changes the URL without using the # character:

```
http://jjtraveltales.com/ChinaSites.html/Slide5
```

Now the web browser will immediately send this request to the web server and attempt to download a new page. This isn't what you want.

So how would you implement the hashbang technique? First, you need to change the URL that appears in the browser whenever your page loads a new slide. (You can do this by setting the `location.href` property in your JavaScript code.) Second, you need to check the URL when the page first loads, retrieve the fragment, and

fetch the corresponding bit of dynamic content from the web server. All of this adds up to a fair bit of juggling, but you can use a JavaScript library like PathJS (*https://github.com/mtrpcic/pathjs*) to make life much easier.

Recently, the hashbang technique has fallen into disrepute, and many of its former supporters have dropped it altogether (for the reasons discussed in the box on this page). However, hashbangs still turn up in some heavily trafficked sites, like Google Groups.

UP TO SPEED

Why Nobody Likes Hashbangs

In recent Web history, the hashbang approach was widely used but deeply controversial. Today, web designers are backing away from it for a number of reasons:

- **Complex URLs.** Facebook is a good example of the problem. In the past, it wouldn't take much browsing before the browser's URL would be polluted with extra information, as in *http://www.facebook.com/profile.php?id=1586010043#!/pages/Haskell/401573824771*. Now designers use session history, if the browser supports it.
- **Inflexibility.** Hashbang pages store a lot of information in the URL. If you change the way a hashbanged page works, or the way it stores information, old URLs could stop working, which is a major website fail.
- **Search engine optimization.** Search engines may treat different hashbanged URLs as essentially the same page. In the *ChinaSites.html* page, that means you won't get a separately indexed page for each tourist site—in fact, search engines might ignore this information altogether. This means that if someone searches for "china wishing tree," the *ChinaSites.html* page might not turn up as a match.
- **Cool URLs matter.** Cool URLs are web page addresses that are short, clear, and—most importantly—never change. Tim Berners-Lee, the creator of the Web, explains the philosophy at *www.w3.org/Provider/Style/URI.html*. And no matter how strongly you feel about keeping good web content alive, hashbang URLs are difficult to maintain and unlikely to survive the next stage in web evolution.

Although webmasters differ over how much they tolerate the hashbang approach, most agree that it's a short stage of web development that soon will be replaced by HTML5's session history feature.

The HTML5 Solution: Session History

HTML5's session history feature provides a different solution to the URL problem. It gives you a way to change the URL to whatever you want, without needing to stick in funny characters like the hashbang. For example, when the *ChinaSites.html* page loads the fourth slide, you could change the URL to look like this:

```
http://jjtraveltales.com/ChinaSites4.html
```

When you do this, the browser won't actually attempt to request a page named *ChinaSites4.html*. Instead, it keeps the current page, with the newly loaded slide, which is exactly what you want. The same is true if the visitor goes back through the browser history. For example, if a visitor moves to the next slide (and the URL changes to *ChinaSites5.html*) and then clicks the Back button (returning the URL to *ChinaSites4.html*), the browser sticks with the current page and raises an event that gives you the chance to load the matching slide and restore the right version of the page.

So far, this sounds like a perfect solution. However, there's a significant drawback. If you want this system to work the way it's intended, you actually need to create a page for every URL you use. In this example, that means you need to create *ChinaSites1.html*, *ChinaSites2.html*, *ChinaSites3.html*, and so on. That's because surfers might go directly to those pages—for example, when returning through a bookmark, typing the link in by hand, clicking it in an email message, and so on. For big web outfits (like Facebook or Flickr), this is no big deal, because they can use a scrap of server-side code to serve up the same slide content in a different package. But if you're a small-scale web developer, it might be a bit more work. For some options on how to handle the challenge, see the box on page 431.

Now that you understand how session history fits into your pages (the hard part), actually using it is easy. In fact, session history consists of just two methods and a single event, all of which are added to the history object.

The most important of these is the `pushState()` method, which lets you change the web page portion of the URL to whatever you want. For security reasons, you can't change the rest of the URL. (If you could, hackers would have a powerful tool for faking other people's websites—including, say, the Gmail sign on a bank transaction form.)

Here's an example that changes the web page part of the URL to ChinaSites4.html:

```
history.pushState(null, null, "ChinaSites4.html");
```

The `pushState()` method accepts three arguments. The third one is the only essential detail—it's the URL that appears in the browser's address bar.

The first argument is any piece of data you want to store to represent the current state of this page. As you'll see, you can use this data to restore the page state if the user returns to this URL through the browser's History list. The second argument is the page title you want the browser to show. All browsers are currently unified in ignoring this detail. If you don't want to set either the state or the title, just supply a *null* value, as shown above.

Here's the code you'd add to the *ChinaSites.html* page to change the URL to match the currently displayed slide. You'll notice that the current slide number is used for the page state. That detail will become important in a moment, when you consider the `onPopState` event:

```
function nextSlide() {
  if (slideNumber == 5) {
    slideNumber = 1;
  } else {
    slideNumber += 1;
  }
  history.pushState(slideNumber, null, "ChinaSites" + slideNumber + ".html");
  goToNewSlide();
  return false;
}
```

```
function previousSlide() {
  if (slideNumber == 1) {
    slideNumber = 5;
  } else {
    slideNumber -= 1;
  }
  history.pushState(slideNumber, null, "ChinaSites" + slideNumber + ".html");
  goToNewSlide();
  return false;
}
```

The goToNewSlide() function hasn't changed from the first version of this example (page 385). It still uses the XMLHttpRequest object to fetch the data for the next slide, asynchronously.

Figure 13-8 shows the new URL management system at work.

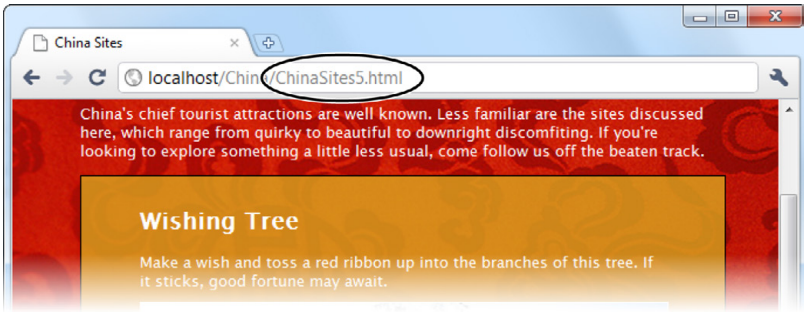


FIGURE 13-8
As a visitor clicks through the slides, the URL changes to match. The URL that appears is clean and sensible, and it refers to exactly the right spot in the slideshow.

If you use the pushState() method, you'll also need to think about the onPopState event, which is its natural counterpart. While the pushState() method puts a new entry into the browser's History list, the onPopState event gives you the chance to deal with it when the user returns.

To understand how it works, consider what happens if a visitor works through all the slides. As she clicks through, the URL in the address bar changes from *ChinaSites.html* to *ChinaSites1.html*, then *ChinaSites2.html*, *ChinaSites3.html*, and so on. Even though the page hasn't actually changed, all of these URLs are added to the browser's history. If the user clicks back to get to a previous slide (for example, moving from *ChinaSites3.html* to *ChinaSites2.html*), the onPopState event is triggered. It provides your code with the state information you stored earlier, with pushState(). Your job is to use that to restore the page to its proper version. In the current example, that means loading the corresponding slide:

```
window.onpopstate = function(e) {
  if (e.state != null) {
```

```
        // What's the slide number for this state?
        // (You could also snip it out of the URL, using the location.href
        // property, but that's more work.)
        slideNumber = e.state;

        // Request this slide from the web server.
        goToNewSlide();
      }
    };
```

You'll notice that this example checks to see if there is any state object before it does its work. That's because some browsers (including Chrome) fire the onPopState event the first time a page is loaded, even if you haven't yet called pushState().

NOTE There's one more new history method, but it's used a lot less frequently—replaceState(). You can use replaceState() to change the state information that's associated with the current page, without adding anything to the History list.

UP TO SPEED

Creating Extra Pages to Satisfy Your URLs

Session history follows the original philosophy of the Web: Every piece of content should be identified with a unique, durable URL. Unfortunately, this means you'll need to make sure that these URLs lead visitors back to the content they want, which is a much stickier affair. For example, when someone types in a request for *ChinaSites3.html*, you need to grab the main content from *ChinaSites.html* and the slide content from *ChinaSites3_slide.html* and somehow stick it together.

If you're a hard-core web programmer, you can write code that runs on the web server, intercepts web requests, and carries out this assembly process on the fly. But if you don't have serious codemaster skills, you'll need to use a different approach.

The simplest option is to make a separate file for each URL—in other words, actually create the files *ChinaSites1.html*, *ChinaSites2.html*, *ChinaSites3.html*, and so on. Of course, you don't want to duplicate the slide content in more than one place (for example, in both *ChinaSites3.html* and *ChinaSites3_slide.html*), because that would create a maintenance nightmare. Fortunately, there are two simple approaches that can simplify your life:

- **Use server-side includes.** If your web server supports this technique (and most do), you can use a special coded instruction like the following:

  ```
  <!--#include file="footer.html" -->
  ```

  Although it looks like a comment, this tells the web server to open the file and insert its contents at that position in the markup. Using this technique, you can insert the main content and the slide content into each slide-specific page. In fact, each slide-specific web page file (*ChinaSites1.html*, *ChinaSites2.html*, and so on) will need just a few lines of markup to create a basic shell of a page.
- **Use templates in a web design tool.** Some web design tools, like Adobe Dreamweaver, allow you to create web templates that can be copied to as many pages as you want. So if you create a template that has the main content and style details, you can reuse it to create all the slide-specific pages you need, quickly and easily.

Browser Compatibility for Session History

Session history has roughly the same level of support as the web workers feature. Once again, it's pre–IE 10 versions of Internet Explorer that are most likely to cause a problem (see Table 13-2).

TABLE 13-2 *Browser support for session history*

	IE	FIREFOX	CHROME	SAFARI	OPERA	SAFARI IOS	ANDROID
Minimum version	10	4	8	5	11.5	5	4.2

There are two ways you can handle a browser that doesn't support session history. If you do nothing at all, the fancy URLs just won't appear. This is what you get if you load the previous example in Internet Explorer—no matter what slide you load up, the URL stays fixed at *ChinaSites.html*. Flickr also uses this approach with its photo streams (to see an example, view *http://tinyurl.com/6hnvanw* with an old version of Internet Explorer).

Another choice is to trigger a full page refresh when the user loads new content on a browser that doesn't support session history. This makes sense if providing a good, meaningful URL is more important that providing the slick experience of dynamically loaded content. However, this approach also takes more work to implement.

One easy way to do it is to enhance your navigation logic so that it performs a page redirect if necessary. In the *ChinaSites.html* page, that involves enhancing the `goToNextSlide()` function, like this:

```
function goToNewSlide() {
  if (window.history) {
    // Session history support is available.
    if (req != null) {
      req.open("GET", "ChinaSites" + slideNumber + "_slide" + ".html", true);
      req.onreadystatechange = newSlideReceived;
      req.send();
    }
    else {
      // There was a problem. Ignore it.
    }
  }
  else {
    // There's no session history support, so direct the browser to a new page.
    window.location = "ChinaSites" + slideNumber + ".html"
  }
}
```

This code checks for session history using the `window.history` property. If support is there, the code downloads just the small chunk of slide data you need and loads it into the existing page. But if session support isn't available, the code abandons this fancy approach and performs an old-school redirect to the new page.