MODULE-5: Managing State

5.1 the problem of state in Web applications

Much of the programming in the previous several chapters has analogies to most typical nonweb application programming. Almost all applications need to process user inputs, output information, and read and write from databases or other storage media. But in this chapter we will be examining a development problem that is unique to the world of web development: how can one request share information with another request?

At first glance this problem does not seem especially formidable. Single-user desktop applications do not have this challenge at all because the program information for the user is stored in memory (or in external storage) and can thus be easily accessed throughout the application. Yet one must always remember that web applications differ from desktop applications in a fundamental way. Unlike the unified single process that is the typical desktop application, a web application consists of a series of disconnected HTTP requests to a web server where each request for a server page is essentially a request to run a separate program, as shown in Figure 5.1.

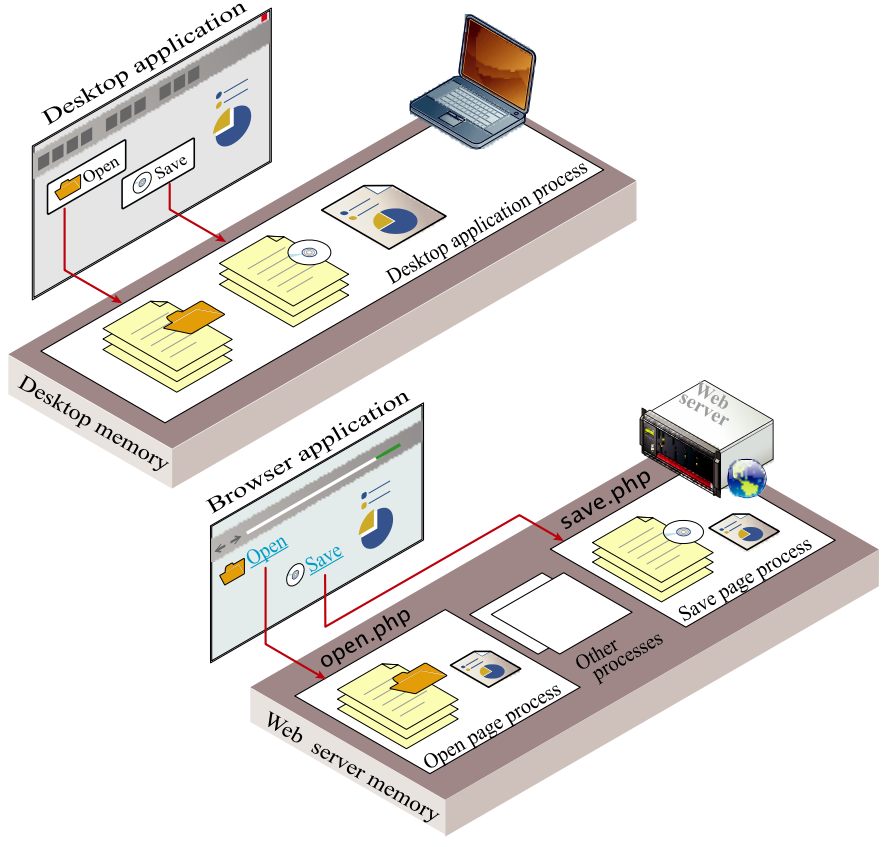

FIGURE 5.1 Desktop applications versus web applications

Furthermore, the web server sees only requests. The HTTP protocol does not, without programming intervention, distinguish two requests by one source from two requests from two different sources, as shown in Figure 5.2.

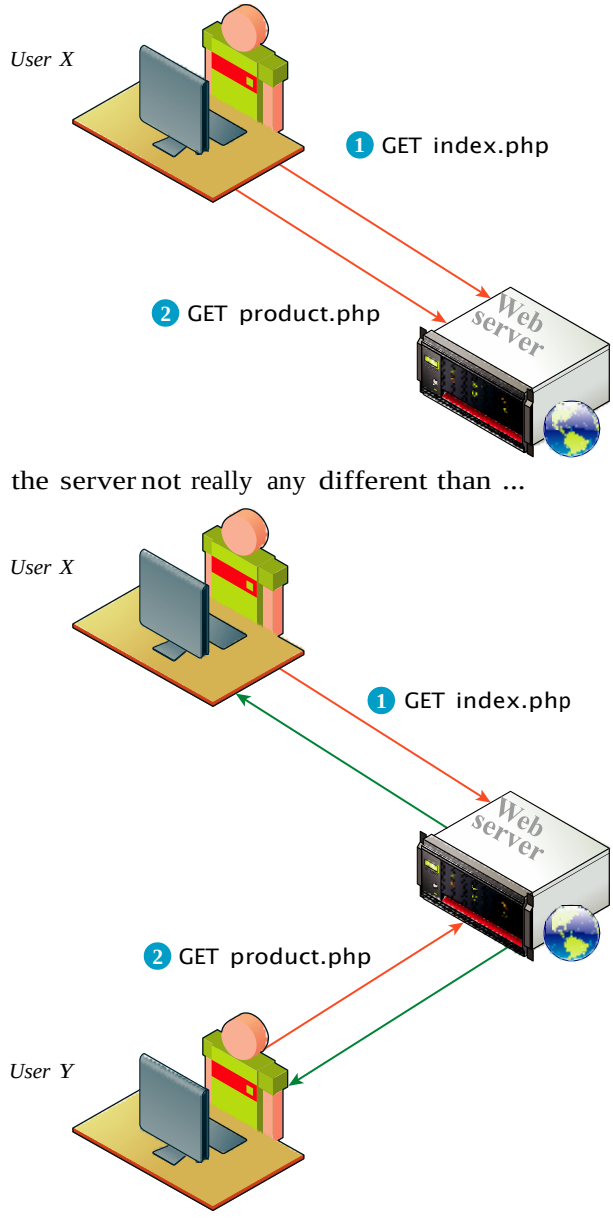

Figure 5.2 What the web server sees

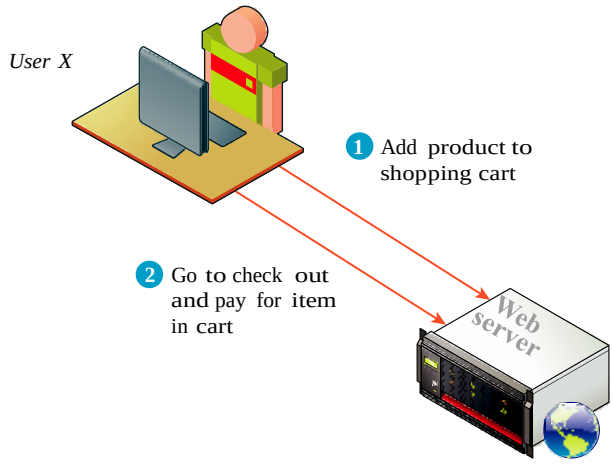

Figure 5.3 What the user wants the server to see

While the HTTP protocol disconnects the user's identity from his or her requests, there are many occasions when we want the web server to connect requests together. Consider the scenario of a web shopping cart, as shown in Figure 5.3. In such a case, the user (and the website owner) most certainly wants the server to recognize that the request to add an item to the cart and the subsequent request to check out and pay for the item in the cart are connected to the same individual.

The rest of this chapter will explain how web programmers and web development environments work together through the constraints of HTTP to solve this particular problem. As we will see, there is no single "perfect" solution, but a variety of different ones each with their own unique strengths and weaknesses.

The starting point will be to examine the somewhat simpler problem of how does one web page pass information to another page? That is, what mechanisms are available within HTTP to pass information to the server in our requests? As we have already seen in Chapters 1, 4, and 9, what we can do to pass information is constrained by the basic request-response interaction of the HTTP protocol. In HTTP, we can pass information using:

- Query strings
- Cookies

5.2 passing information via Query strings

As you will recall from earlier chapters, a web page can pass query string information from the browser to the server using one of the two methods: a query string within the URL (GET) and a query string within the HTTP header (POST). Figure 5.4 reviews these two different approaches.

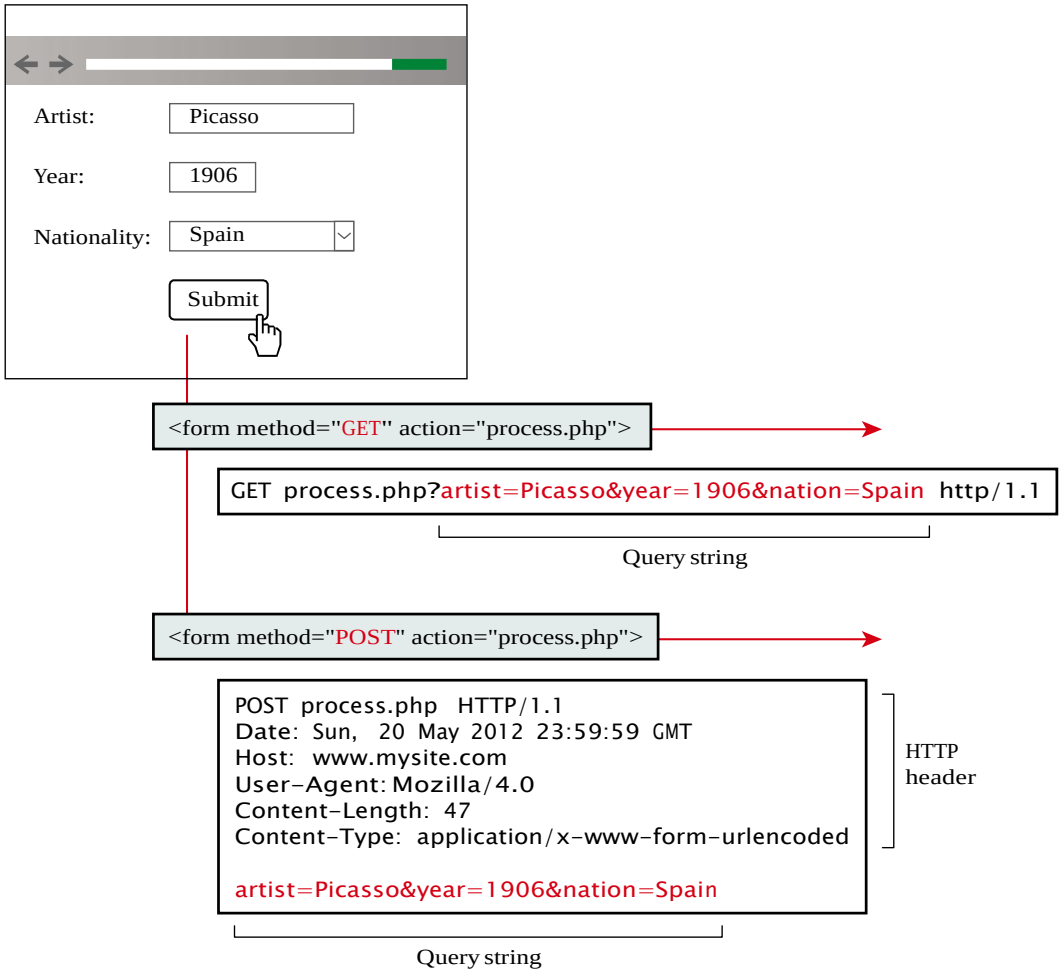

Figure 5.4 Recap of GET versus POST

5.3 passing information via the UrL path

While query strings are a vital way to pass information from one page to another, they do have a drawback. The URLs that result can be long and complicated. While for many users this is not that important, many feel that for one particular type of user, query strings are not ideal. Which type of user? Perhaps the single most important user: search engines.

While there is some dispute about whether dynamic URLs (i.e., ones with query string parameters) or static URLs are better from a search engine result optimization (or SEO for search engine optimization) perspective, the consensus is that static URLs do provide some benefits with search engine result rankings. Many factors affect a page's ranking in a search engine, as you will see in Chapter 20, but the appearance of search terms within the URL does seem to improve its relative position. Another benefit to static URLs is that users tend to prefer them.

As we have seen, dynamic URLs (i.e., query string parameters) are a pretty essential part of web application development. How can we do without them? The answer is to rewrite the dynamic URL into a static one (and vice versa). This process is commonly called **URL rewriting**.

For instance, in Figure 5.5, the top four commerce-related results for the search term "reproductions Raphael portrait la donna velata" are shown along with their URLs. Notice how the top three do not use query string parameters but instead put the relevant information within the folder path or the file name.

You might notice as well that the extension for the first three results is **.html**. This doesn't mean that these sites are serving static HTML files (in fact two of them are using PHP); rather the file name extension is also being rewritten to make the URL friendlier.

We can try doing our own rewriting. Let us begin with the following URL with its query string information:

```
www.somedomain.com/DisplayArtist.php?artist=16
```

One typical alternate approach would be to rewrite the URL to:

```
www.somedomain.com/artists/16.php
```

Notice that the query string name and value have been turned into path names. One could improve this to make it more SEO friendly using the following:

```
www.somedomain.com/artists/Mary-Cassatt
```

5.3.1 UrL rewriting in apache and Linux

Depending on your web development platform, there are different ways to implement URL rewriting. On web servers running Apache, the solution typically involves using the `mod_rewrite` module in Apache along with the `.htaccess` file.

FIGURE 13.5 URLs within a search engine result page

The mod_rewrite module uses a rule-based rewriting engine that utilizes Perl-compatible regular expressions to change the URLs so that the requested URL can be mapped or redirected to another URL internally.

URL rewriting requires knowledge of the Apache web server, so the details of URL rewriting are covered in Section 19.3.12 of Chapter 19, after some more background on Apache has been presented.

5.4 Cookies

There are few things in the world of web development so reviled and misunderstood as the HTTP cookie. **Cookies** are a client-side approach for persisting state information. They are name=value pairs that are saved within one or more text

files that are managed by the browser. These pairs accompany both server requests and responses within the HTTP header. While cookies cannot contain viruses, third-party tracking cookies have been a source of concern for privacy advocates.

Cookies were intended to be a long-term state mechanism. They provide web-site authors with a mechanism for persisting user-related information that can be stored on the user's computer and be managed by the user's browser.

Cookies are not associated with a specific page but with the page's domain, so the browser and server will exchange cookie information no matter what page the user requests from the site. The browser manages the cookies for the different domains so that one domain's cookies are not transported to a different domain.

While cookies can be used for any state-related purpose, they are principally used as a way of maintaining continuity over time in a web application. One typical use of cookies in a website is to "remember" the visitor, so that the server can customize the site for the user. Some sites will use cookies as part of their shopping cart implementation so that items added to the cart will remain there even if the user leaves the site and then comes back later. Cookies are also frequently used to keep track of whether a user has logged into a site.

5.4.1 how Do Cookies Work?

While cookie information is stored and retrieved by the browser, the information in a cookie travels within the HTTP header. Figure 5.6 illustrates how cookies work.

There are limitations to the amount of information that can be stored in a cookie (around 4K) and to the number of cookies for a domain (for instance, Internet Explorer 6 limited a domain to 20 cookies).

Like their similarly named chocolate chip brethren beloved by children world-wide, HTTP cookies can also expire. That is, the browser will delete cookies that are beyond their expiry date (which is a configurable property of a cookie). If a cookie does not have an expiry date specified, the browser will delete it when the browser closes (or the next time it accesses the site). For this reason, some commentators will say that there are two types of cookies: session cookies and persistent cookies. A **session cookie** has no expiry stated and thus will be deleted at the end of the user browsing session. **Persistent cookies** have an expiry date specified; they will persist in the browser's cookie file until the expiry date occurs, after which they are deleted.

The most important limitation of cookies is that the browser may be configured to refuse them. As a consequence, sites that use cookies should not depend on their availability for critical features. Similarly, the user can also delete cookies or even tamper with the cookies, which may lead to some serious problems if not handled. Several years ago, there was an instructive case of a website selling stereos and televisions that used a cookie-based shopping cart. The site placed not only the product

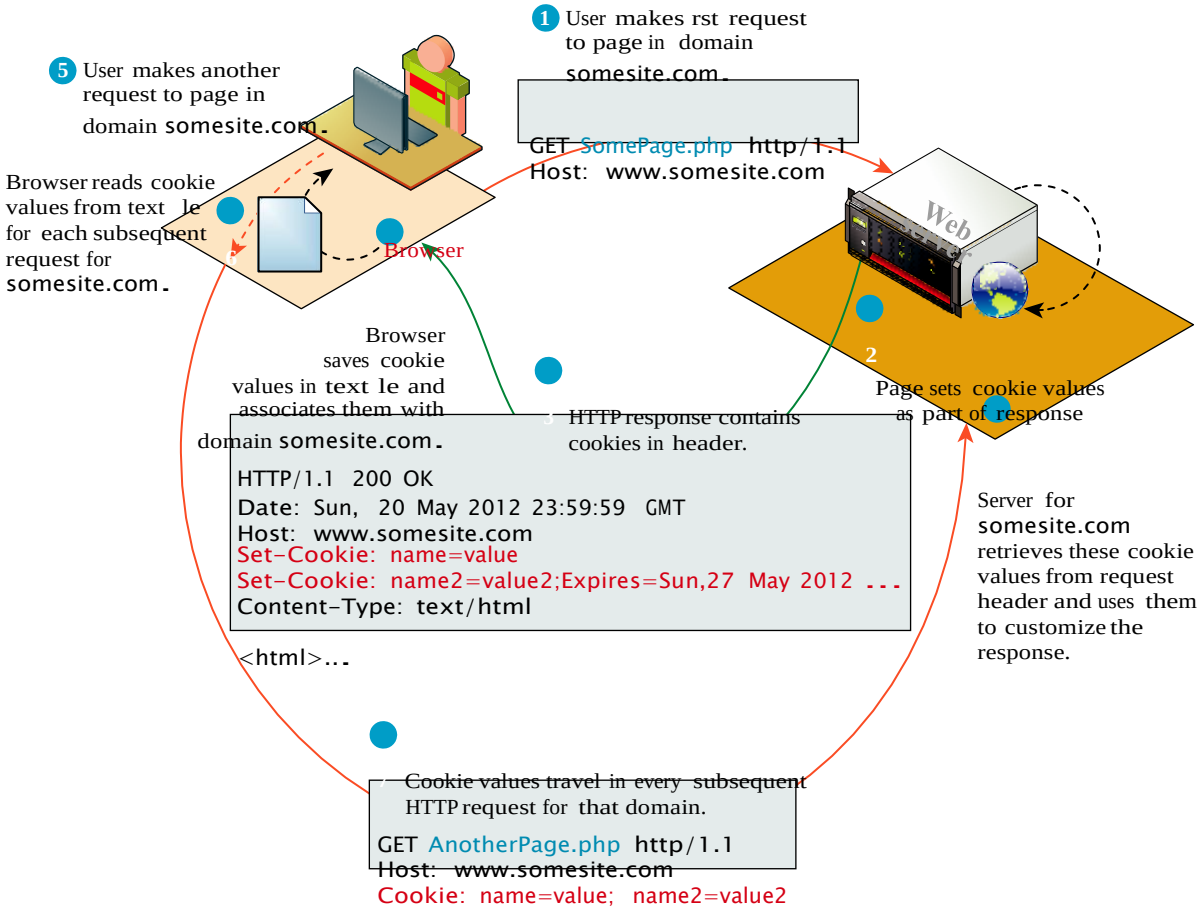

Figure 5.6 Cookies at work

identifier but also the product price in the cart. Unfortunately, the site then used the price in the cookie in the checkout. Several curious shoppers edited the price in the cookie stored on their computers, and then purchased some big-screen televisions for only a few cents!

5.4.2 Using Cookies

Like any other web development technology, PHP provides mechanisms for writing and reading cookies. Cookies in PHP are *created* using the `setcookie()` function and are *retrieved* using the `$_COOKIES` superglobal associative array, which works like the other superglobals covered in Chapter 9.

Listing 5.1 illustrates the writing of a persistent cookie in PHP. **It is important to note that cookies must be written before any other page output**.

```
<?php
    // add 1 day to the current time for expiry time
    $expiryTime = time()+60*60*24;

    // create a persistent cookie
    $name = "Username";
    $value = "Ricardo";
    setcookie($name, $value, $expiryTime);
?>
```

listing 5.1 Writing a cookie

The `setcookie()` function also supports several more parameters, which further customize the new cookie. You can examine the online official PHP documentation for more information.[1]

Listing 5.2 illustrates the reading of cookie values. Notice that when we read a cookie, we must also check to ensure that the cookie exists. In PHP, if the cookie has expired (or never existed in the first place), then the client's browser would not send anything, and so the `$_COOKIE` array would be blank.

5.4.3 persistent Cookie best practices

So what kinds of things should a site store in a persistent cookie? Due to the limitations of cookies (both in terms of size and reliability), your site's correct operation

```
<?php
   if( !isset($_COOKIE['Username']) ) {
      //no valid cookie found
   }
   else {
      echo "The username retrieved from the cookie is:";
      echo $_COOKIE['Username'];
   }
?>
```

Listing 5.2 Reading a cookie

should not be dependent upon cookies. Nonetheless, the user's experience might be improved with the judicious use of cookies.

Many sites provide a "Remember Me" checkbox on login forms, which relies on the use of a persistent cookie. This login cookie would contain the user's username but not the password. Instead, the login cookie would contain a random token; this random token would be stored along with the username in the site's back-end database. Every time the user logs in, a new token would be generated and stored in the database and cookie.

Another common, nonessential use of cookies would be to use them to store user preferences. For instance, some sites allow the user to choose their preferred site color scheme or their country of origin or site language. In these cases, saving the user's preferences in a cookie will make for a more contented user, but if the user's browser does not accept cookies, then the site will still work just fine; at worst the user will simply have to reselect his or her preferences again.

Another common use of cookies is to track a user's browsing behavior on a site. Some sites will store a pointer to the last requested page in a cookie; this information can be used by the site administrator as an analytic tool to help understand how users navigate through the site.

5.5 serialization

Serialization is the process of taking a complicated object and reducing it down to zeros and ones for either storage or transmission. Later that sequence of zeros and ones can be reconstituted into the original object as illustrated in Figure 5.7.

In PHP objects can easily be reduced down to a binary string using the `serialize()` function. The resulting string is a binary representation of the object and therefore may contain unprintable characters. The string can be reconstituted back into an object using the `unserialize()` method.[2]

While arrays, strings, and other primitive types will be serializable by default, classes of our own creation must implement the `Serializable` interface shown in Listing 5.3, which requires adding implementations for `serialize()` and `unserialize()` to any class that implements this interface.

```
interface Serializable {
    /* Methods */
    public function serialize();
    public function unserialize($serialized);
}
```

listing 5.3 The Serializable interface

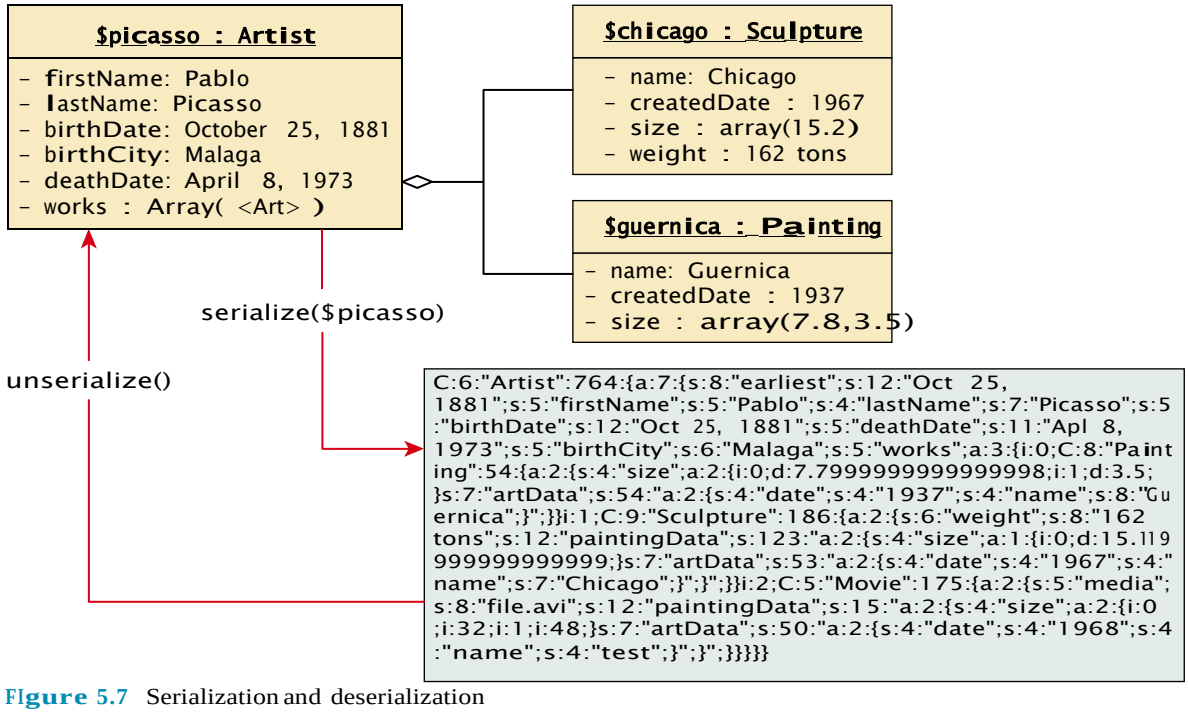

Figure 5.7 Serialization and deserialization

Listing 5.4 shows how the Artist class must be modified to implement the Serializable interface by adding the implements keyword to the class definition and adding implementations for the two methods.

```
class Artist implements Serializable {
    //...
    // Implement the Serializable interface methods
    public function serialize() {
        // use the built-in PHP serialize function
        return serialize(
                array("earliest" =>self::$earliestDate,
                      "first" => $this->firstName,
                      "last" => $this->lastName,
                      "bdate" => $this->birthDate,
                      "ddate" => $this->deathDate,
                      "bcity" => $this->birthCity,
                      "works" => $this->artworks
                      );
                );
    }
    public function unserialize($data) {
        // use the built-in PHP unserialize function
        $data = unserialize($data);
        self::$earliestDate = $data['earliest'];
        $this->firstName = $data['first'];
        $this->lastName = $data['last'];
        $this->birthDate = $data['bdate'];
        $this->deathDate = $data['ddate'];
        $this->birthCity = $data['bcity'];
        $this->artworks = $data['works'];
    }
    //...
}
```

Listing 5.4 Artist class modified to implement the Serializable interface

Note that in order for our Artist class to save successfully, the Art, Painting, and other classes must also implement the Serializable interface (not shown here). It should be noted that references to other objects stored at the same time will be preserved while references to objects not serialized in this operation will be lost. This will influence how we use serialization, since if we want to store an object model, we must store all associated objects at once.

The output of calling serialize($picasso) is:

```
C:6:"Artist":764:{a:7:{s:8:"earliest";s:5:"Oct 25, 1881";s:5:"first
Name";s:5:"Pablo";s:4:"lastName";s:7:"Picasso";s:5:"birthDate";s:5:
        "Oct 25, 1881";s:5:"deathDate";s:11:"Apl 8, 1973";s:5:"birthCity";
        s:6:"Malaga";s:5:"works"; a:3:{i:0;C:8:"Painting":54:{a:2:{s:4:"size";
        a:2:{i:0;d:7.7999999999999998;i:1;d:3.5;}s:7:"artData";s:54:"a:2:
        {s:4:"date";s:4:"1937";s:4:"name";s:8:"Guernica";}";}}i:1;C:9:"Sculpture"
        :186:{a:2:{s:6:"weight";s:8:"162 tons";s:5:"paintingData"; s:53:
        "a:2:{s:4:"size";a:1:{i:0;d:15.119999999999999;}s:7:"artData";s:53:"
        a:2:{s:4:"date";s:4:"1967";s:4:"name";s:7:"Chicago";}";}";}}i:2;C:5:
        "Movie":175:{a:2:{s:5:"media";s:8:"file.avi";s:5:"paintingData";s:1
        5:"a:2:{s:4:"size";a:2:{i:0;i:32;i:1;i:48;}s:7:"artData";s:50:"a:2:
        {s:4:"date";s:4:"1968";s:4:"name";s:4:"test";}";}";}}}}}
```

Although nearly unreadable to most people, this data can easily be used to reconstitute the object by passing it to unserialize(). If the data above is assigned to $data, then the following line will instantiate a new object identical to the original:

```
$picassoClone = unserialize($data);
```

5.5.1 application of serialization

Since each request from the user requires objects to be reconstituted, using serialization to store and retrieve objects can be a rapid way to maintain state between requests. At the end of a request you store the state in a serialized form, and then the next request would begin by deserializing it to reestablish the previous state.

In the next section, you will encounter session state, and will discover that PHP serializes objects for you in its implementation of session state.

5.6 session state

All modern web development environments provide some type of session state mechanism. **Session state** is a server-based state mechanism that lets web applications store and retrieve objects of any type for each unique user session. That is, each browser session has its own session state stored as a serialized file on the server, which is deserialized and loaded into memory as needed for each request, as shown in Figure 5.8.

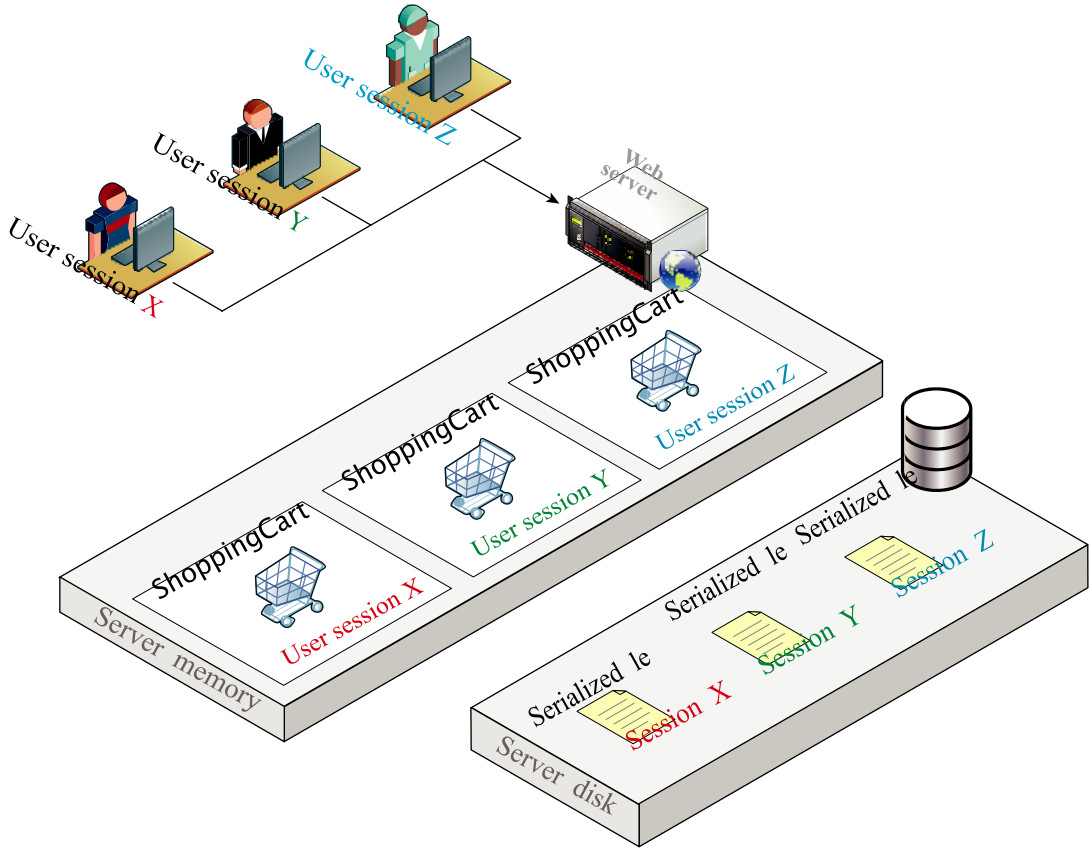

Figure 5.8 Session state

Because server storage is a finite resource, objects loaded into memory are released when the request completes, making room for other requests and their session objects. This means there can be more active sessions on disk than in memory at any one time.

Session state is ideal for storing more complex (but not too complex . . . more on that later) objects or data structures that are associated with a user session. The classic example is a shopping cart. While shopping carts could be implemented via cookies or query string parameters, it would be quite complex and cumbersome to do so.

In PHP, session state is available to the developer as a superglobal associative array, much like the $_GET, $_POST, and $_COOKIE arrays.[3] It can be accessed via the $_SESSION variable, but unlike the other superglobals, you have to take additional steps in your own code in order to use the $_SESSION superglobal.

To use sessions in a script, you must call the `session_start()` function at the beginning of the script as shown in Listing 5.5. In this example, we differentiate a logged-in user from a guest by checking for the existence of the `$_SESSION["user"]` variable.

```
<?php

session_start();

if ( isset($_SESSION['user']) ) {
   // User is logged in
}
else {
   // No one is logged in (guest)
}
?>
```

Listing 5.5 Accessing session state

Session state is typically used for storing information that needs to be preserved across multiple requests by the same user. Since each user session has its own session state collection, it should not be used to store large amounts of information because this will consume very large amounts of server memory as the number of active sessions increase.

As well, since session information does eventually time out, one should always check if an item retrieved from session state still exists before using the retrieved object. If the session object does not yet exist (either because it is the first time the user has requested it or because the session has timed out), one might generate an error, redirect to another page, or create the required object using the lazy initialization approach as shown in Listing 5.6. In this example `ShoppingCart` is a user-defined class. Since PHP sessions are serialized into files, one must ensure that any

```
<?php
include_once("ShoppingCart.class.php");

session_start();

// always check for existence of session object before accessing it
if ( !isset($_SESSION["Cart"]) ) {
   //session variables can be strings, arrays, or objects, but
   // smaller is better
   $_SESSION["Cart"] = new ShoppingCart();
}
$cart = $_SESSION["Cart"];
?>
```

Listing 5.6 Checking session existence

classes stored into sessions can be serialized and deserialized, and that the class definitions are parsed before calling `session_start()`.

5.6.1 how Does session state Work?

Typically when our students learn about session state, their first reaction is to say "Why didn't we learn this first? This solves all our problems!" Indeed because modern development environments such as ASP.NET and PHP make session state remarkably easy to work with, it is tempting to see session state as a one-stop solution to all web state needs. However, if we take a closer look at how session state works, we will see that session state has the same limitations and issues as the other state mechanisms examined in this chapter.

The first thing to know about session state is that it works within the same HTTP context as any web request. The server needs to be able to identify a given HTTP request with a specific user request. Since HTTP is stateless, some type of user/session identification system is needed. Sessions in PHP (and ASP.NET) are identified with a unique session ID. In PHP, this is a unique 32-byte string that is by default transmitted back and forth between the user and the server via a session cookie (see Section 5.4.1 above), as shown in Figure 5.9.

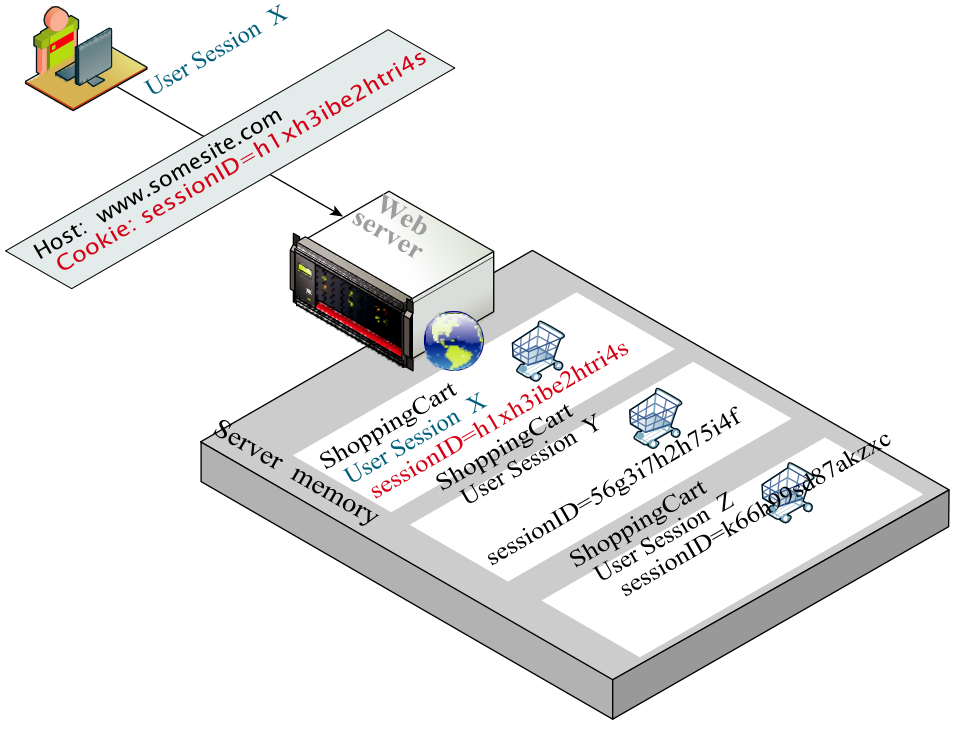

Figure 5.9 Session IDs

As we learned earlier in the section on cookies, users may disable cookie support in their browser; for that reason, PHP can be configured (in the **php.ini** file) to instead send the session ID within the URL path.

So what happens besides the generating or obtaining of a session ID after a new session is started? For a brand new session, PHP assigns an initially empty dictionary-style collection that can be used to hold any state values for this session. When the request processing is finished, the session state is saved to some type of state storage mechanism, called a session state provider (discussed in next section). Finally, when a new request is received for an already existing session, the session's dictionary collection is filled with the previously saved session data from the session state provider.

5.6.2 session storage and Configuration

You may have wondered why session state providers are necessary. In the example shown in Figure 5.8, each user's session information is kept in serialized files, one per session (in ASP.NET, session information is by default not stored in files, but in memory). It is possible to configure many aspects of sessions including where the session files are saved. For a complete listing refer to the session configuration options in **php.ini**.

The decision to save sessions to files rather than in memory (like ASP.NET) addresses the issue of memory usage that can occur on shared hosts as well as persistence between restarts. Many sites run in commercial hosting environments that are also hosting many other sites. For instance, one of the book author's personal sites (**randyconnolly.com**, which is hosted by **discountasp.net**) is, according to a Reverse IP Domain Check, on a server that was hosting 68 other sites when this chapter was being written. Inexpensive web hosts may sometimes stuff hundreds or even thousands of sites on each machine. In such an environment, the server memory that is allotted per web application will be quite limited. And remember that for each application, server memory may be storing not only session information, but pages being executed, and caching information, as shown in Figure 5.10.

On a busy server hosting multiple sites, it is not uncommon for the Apache application process to be restarted on occasion. If the sessions were stored in

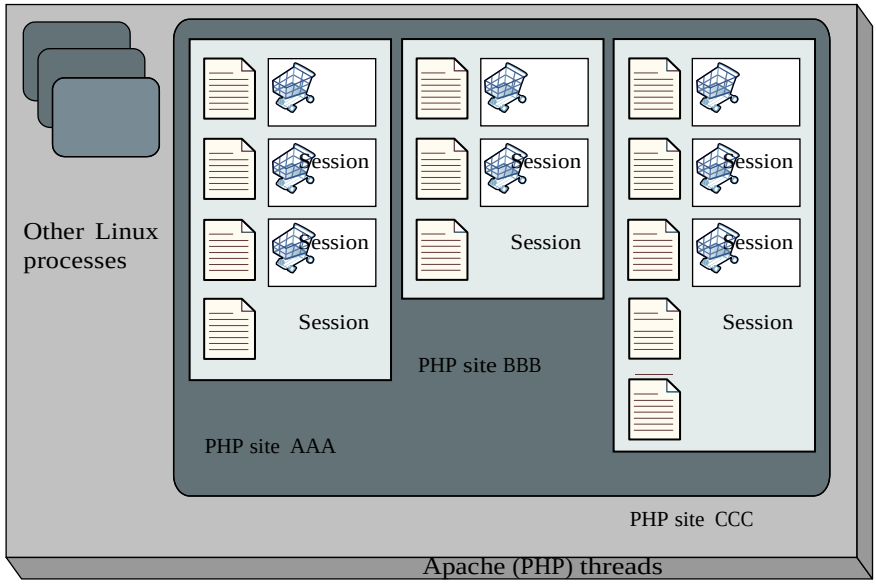

Figure 5.10 Applications and server memory

memory, the sessions would all expire, but as they are stored into files, they can be instantly recovered as though nothing happened. This can be an issue in environments where sessions are stored in memory (like ASP.NET), or a custom session handler is involved. One downside to storing the sessions in files is a degradation in performance compared to memory storage, but the advantages, it was decided, outweigh those challenges.

Higher-volume web applications often run in an environment in which multiple web servers (also called a web farm) are servicing requests. Each incoming request is forwarded by a load balancer to any one of the available servers in the farm. In such a situation the in-process session state will not work, since one server may service one request for a particular session, and then a completely different server may service the next request for that session, as shown in Figure 5.11.

There are a number of different ways of managing session state in such a web farm situation, some of which can be purchased from third parties. There are effectively two categories of solution to this problem.

1. Configure the load balancer to be "session aware" and relate all requests using a session to the same server.

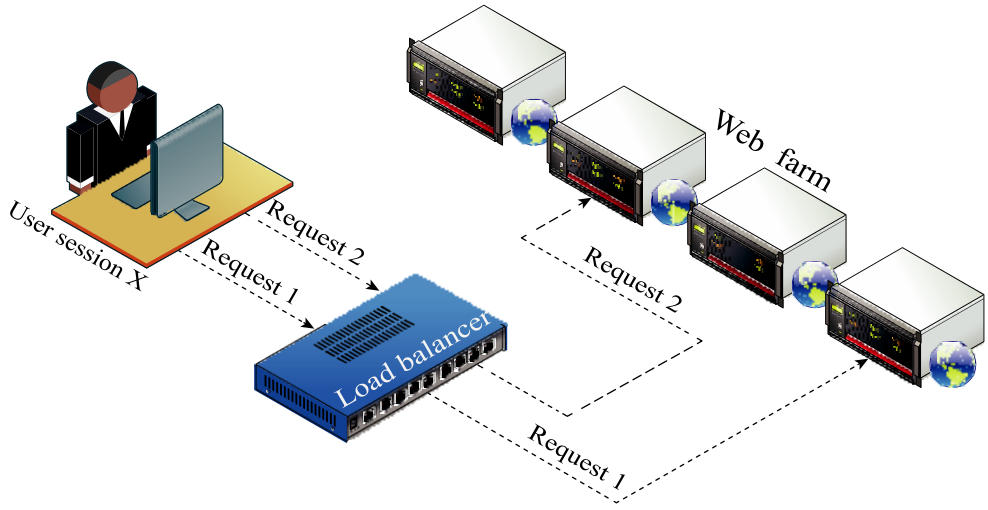

Figure 5.11 Web farm

2. Use a shared location to store sessions, either in a database, memcache (covered in the next section), or some other shared session state mechanism as seen in Figure 5.12.

Using a database to store sessions is something that can be done programmatically, but requires a rethinking of how sessions are used. Code that was written to work on a single server will have to be changed to work with sessions in a shared database, and therefore is cumbersome. The other alternative is to configure PHP to

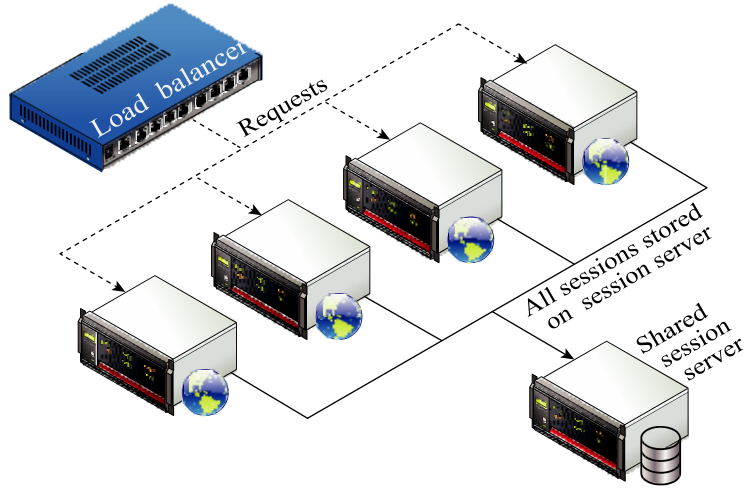

Figure 5.12 Shared session provider

use memcache on a shared server (covered in Section 5.8). To do this you must have PHP compiled with memcache enabled; if not, you may need to install the module. Once installed, you must change the **php.ini** on all servers to utilize a shared location, rather than local files as shown in Listing 5.7.

```
[Session]
; Handler used to store/retrieve data.
session.save_handler = memcache
session.save_path = "tcp://sessionServer:11211"
```

listing 5.7 Configuration in php.ini to use a shared location for sessions

5.7 HTML5 Web storage

Web storage is a new JavaScript-only API introduced in HTML5.[4] It is meant to be a replacement (or perhaps supplement) to cookies, in that web storage is managed by the browser; unlike cookies, web storage data is not transported to and from the server with every request and response. In addition, web storage is not limited to the 4K size barrier of cookies; the W3C recommends a limit of 5MB but browsers are allowed to store more per domain. Currently web storage is supported by current versions of the major browsers, including IE8 and above. However, since JavaScript, like cookies, can be disabled on a user's browser, web storage should not be used for mission-critical application functions.

Just as there were two types of cookies, there are two types of global web storage objects: `localStorage` and `sessionStorage`. The `localStorage` object is for saving information that will persist between browser sessions. The `sessionStorage` object is for information that will be lost once the browser session is finished.

These two objects are essentially key-value collections with the same interface (i.e., the same JavaScript properties and functions).

5.7.1 Using Web storage

Listing 5.8 illustrates the JavaScript code for writing information to web storage. Do note that it is *not* PHP code that interacts with the web storage mechanism but JavaScript. As demonstrated in the listing, there are two ways to store values in web storage: using the `setItem()` function, or using the property shortcut (e.g., `sessionStorage.FavoriteArtist`).

Listing 5.9 demonstrates that the process of reading from web storage is equally straightforward. The difference between `sessionStorage` and `localStorage` in this example is that if you close the browser after writing and then run the code in Listing 5.8, only the `localStorage` item will still contain a value.

```
<form ... >
    <h1>Web Storage Writer</h1>
    <script language="javascript" type="text/javascript">

        if (typeof (localStorage) === "undefined" ||
              typeof (sessionStorage) === "undefined") {
            alert("Web Storage is not supported on this browser...");
        }
        else {
            sessionStorage.setItem("TodaysDate", new Date());
            sessionStorage.FavoriteArtist = "Matisse";

            localStorage.UserName = "Ricardo";
            document.write("web storage modified");
        }
    </script>
    <p><a href="WebStorageReader.php">Go to web storage reader</a></p>
</form>
```

listing 5.8 Writing web storage

```
<form id="form1" runat="server">
    <h1>Web Storage Reader</h1>
    <script language="javascript" type="text/javascript">

        if (typeof (localStorage) === "undefined" ||
              typeof (sessionStorage) === "undefined") {
            alert("Web Storage is not supported on this browser...");
        }
        else {
            var today = sessionStorage.getItem("TodaysDate");
            var artist = sessionStorage.FavoriteArtist;

            var user = localStorage.UserName;
            document.write("date saved=" + today);
            document.write("<br/>favorite artist=" + artist);
            document.write("<br/>user name = " + user);
        }
    </script>
</form>
```

listing 5.9 Reading web storage

5.7.2 Why Would We Use Web storage?

Looking at the two previous listings you might wonder why we would want to use web storage. Cookies have the disadvantage of being limited in size, potentially disabled by the user, vulnerable to XSS and other security attacks, and being sent in every single request and response to and from a given domain. On the other hand, the fact that cookies are sent with every request and response is also their main advantage: namely, that it is easy to implement data sharing between the client browser and the server. Unfortunately with web storage, transporting the information within web storage back to the server is a relatively complicated affair involving the construction of a web service on the server (see Chapter 17) and then using asynchronous communication via JavaScript to push the information to the server.

A better way to think about web storage is not as a cookie replacement but as a local cache for relatively static items available to JavaScript. One practical use of web storage is to store static content downloaded asynchronously such as XML or JSON from a web service in web storage, thus reducing server load for subsequent requests by the session.

Figure 5.5 illustrates an example of how web storage could be used as a mechanism for reducing server data requests, thereby speeding up the display of the page on the browser, as well as reducing load on the server.

5.8 Caching

Caching is a vital way to improve the performance of web applications. Your browser uses caching to speed up the user experience by using locally stored versions of images and other files rather than re-requesting the files from the server. While important, from a server-side perspective, a server-side developer only has limited control over browser caching .

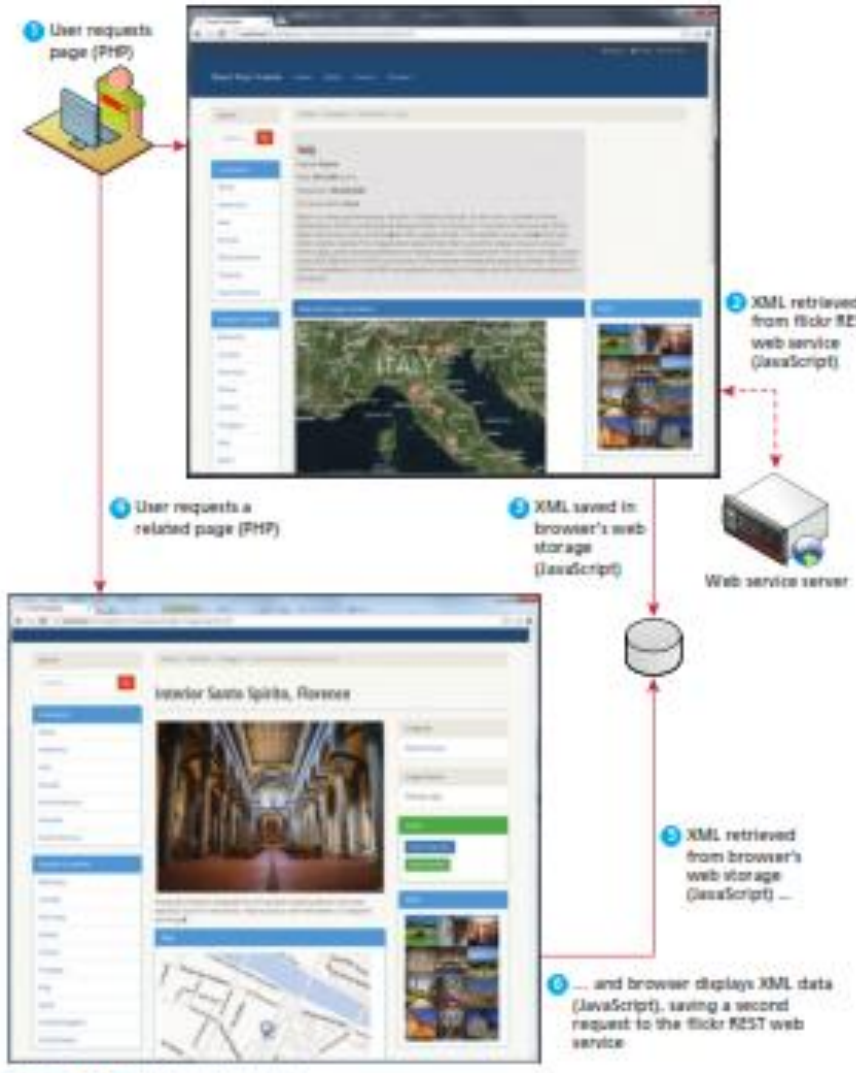

FIGURE 13.13 Using web storage

There is a way, however, to integrate caching on the server side. Why is this necessary? Remember that every time a PHP page is requested, it must be fetched, parsed, and executed by the PHP engine, and the end result is HTML that is sent back to the requestor. For the typical PHP page, this might also involve numerous database queries and processing to build. If this page is being served thousands of times per second, the dynamic generation of that page may become unsustainable.

One way to address this problem is to **cache** the generated markup in server memory so that subsequent requests can be served from memory rather than from the execution of the page.

There are two basic strategies to caching web applications. The first is **page output caching**, which saves the rendered output of a page or user control and reuses the output instead of reprocessing the page when a user requests the page again. The second is **application data caching**, which allows the developer to programmatically cache data.

5.8.1 page Output Caching

In this type of caching, the contents of the rendered PHP page (or just parts of it) are written to disk for fast retrieval. This can be particularly helpful because it allows PHP to send a page response to a client without going through the entire page processing life cycle again (see Figure 5.14). Page output caching is especially useful for pages whose content does not change frequently but which require significant processing to create.

There are two models for page caching: full page caching and partial page caching. In full page caching, the entire contents of a page are cached. In partial page caching, only specific parts of a page are cached while the other parts are dynamically generated in the normal manner.

Page caching is not included in PHP by default, which has allowed a marketplace for free and commercial third-party cache add-ons such as Alternative PHP Cache (open source) and Zend (commercial) to flourish. However, one can easily create basic caching functionality simply by making use of the output buffering and time functions. The mod_cache module that comes with the Apache web server engine is the most common way websites implement page caching. This separates server tuning from your application code, simplifying development, and leaving cache control up to the web server rather than the application developer. The details of configuring that Apache cache are described in Section 19.4.6 of Chapter 19.

It should be stressed that it makes no sense to apply page output caching to every page in a site. However, performance improvements can be gained (i.e., reducing server loads) by caching the page output of especially busy pages in which the content is the same for all users.

5.8.2 application Data Caching

One of the biggest drawbacks with page output caching is that performance gains will only be had if the entire cached page is the same for numerous requests.

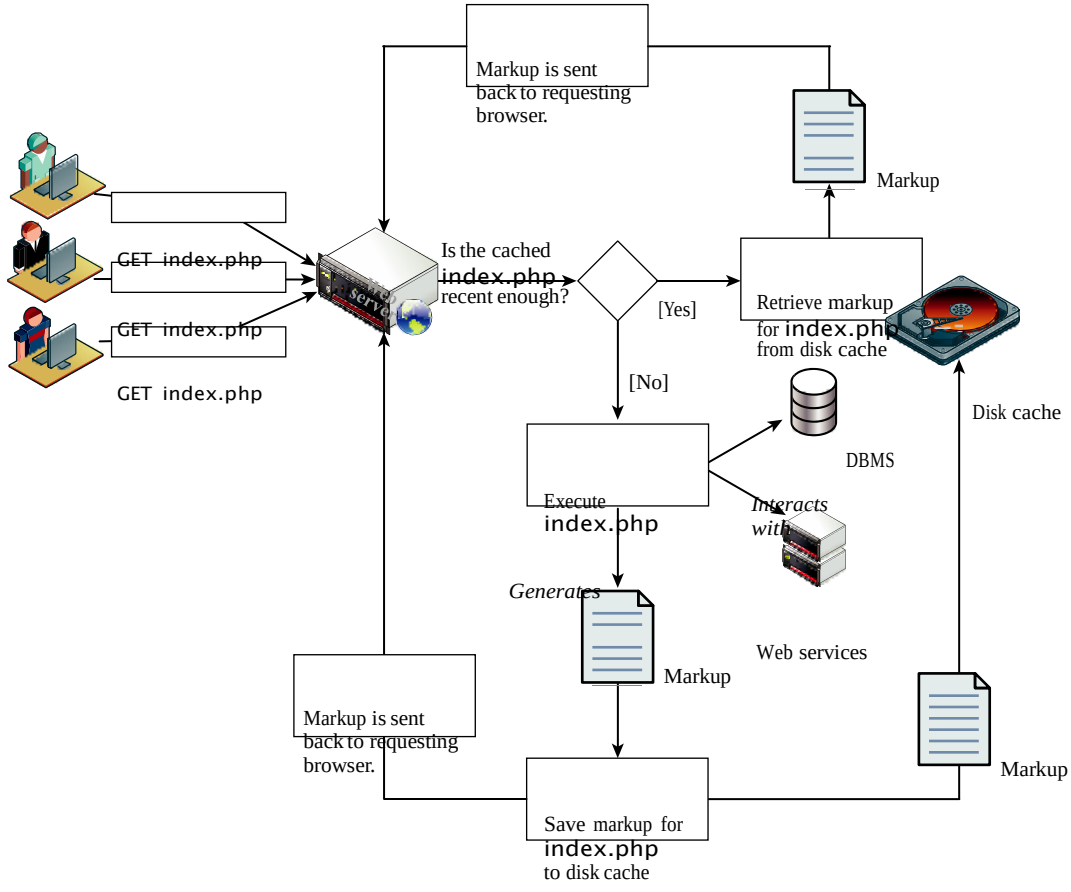

Figure 5.14 Page output caching

However, many sites customize the content on each page for each user, so full or partial page caching may not always be possible.

An alternate strategy is to use application data caching in which a page will programmatically place commonly used collections of data that require time-intensive queries from the database or web server into cache memory, and then other pages that also need that same data can use the cache version rather than re-retrieve it from its original location.

While the default installation of PHP does not come with an application caching ability, a widely available free PECL extension called memcache is widely used to provide this ability.[5] Listing 5.10 illustrates a typical use of memcache.

It should be stressed that memcache should not be used to store large collections. The size of the memory cache is limited, and if too many things are placed in it, its performance advantages will be lost as items get paged in and out. Instead, it should be used for relatively small collections of data that are frequently accessed on multiple pages.

```
<?php

// create connection to memory cache
$memcache = new Memcache;
$memcache->connect('localhost', 11211)
    or die ("Could not connect to memcache server");

$cacheKey = 'topCountries';
/* If cached data exists retrieve it, otherwise generate and cache
    it for next time */
    if ( ! isset($countries = $memcache->get($cacheKey)) ) {

    // since every page displays list of top countries as links
    // we will cache the collection

    // first get collection from database
    $cgate = new CountryTableGateway($dbAdapter);
    $countries = cgate->getMostPopular();

    // now store data in the cache (data will expire in 240 seconds)
    $memcache->set($cacheKey, $countries, false, 240)
        or die ("Failed to save cache data at the server");
}
// now use the country collection
displayCountryList($countries);

?>
```

listing 5.10 Using memcache

15.1.1 javascript pseudo-Classes

Although JavaScript has no formal class mechanism, it does support objects (such as the DOM). While most object-oriented languages that support objects also support classes formally, JavaScript does not. Instead, you define **pseudo-classes** through a variety of interesting and nonintuitive syntax constructs. Many common features of object-oriented programming, such as inheritance and even simple methods, must be arrived at through these nonintuitive means. Despite this challenge, the benefits of using object-oriented design in your JavaScript include increased code reuse, better memory management, and easier maintenance. From a practical perspective, almost all modern frameworks (such as jQuery and the Google Maps API) use prototypes to simulate classes, so understanding the mechanism is essential to apply those APIs in your applications.

This section will demonstrate how you mimic class features through the creation of a simple prototype to represent a single die object (die, the singular for dice) which could be used in a game of some sort. This process will begin with the simplest mechanisms and introduce new syntactic constructs until we arrive at the best way to create and use pseudo-classes (prototypes) in JavaScript.

15.1.1.1 Using Object Literals

Recall that an array in JavaScript can be instantiated with elements in the following way:

```
var daysofWeek = ["sun","mon","tue","wed","thu","fri","sat"];
```

An object can be instantiated using the similar concept of **object literals**: that is, an object represented by the list of key-value pairs with colons between the key and value with commas separating key-value pairs.

A dice object, with a string to hold the color and an array containing the values representing each side (face), could be defined all at once using object literals as follows:

```
var oneDie = { color : "FF0000", faces : [1,2,3,4,5,6] };
```

Once defined, these elements can be accessed using dot notation. For instance, one could change the color to blue by writing:

```
oneDie.color="0000FF";
```

15.1.1.2 emulate Classes through Functions

Although a formal *class* mechanism is not available to us in JavaScript, it is possible to get close by using functions to encapsulate variables and methods together, as shown in Listing 15.1.1.

```
function Die(col) {
   this.color=col;
   this.faces=[1,2,3,4,5,6];
}
```

listing 15.1.1 Very simple Die pseudo-class definition as a function

The `this` keyword inside of a function refers to the instance, so that every reference to internal properties or methods manages its own variables, as is the case with PHP. One can create an instance of the object as follows, very similar to PHP.

```
var oneDie = new Die("0000FF");
```

Developers familiar with using objects in Java or PHP typically use a constructor to instantiate objects. In JavaScript, there is no need for an explicit constructor since the function definition acts as both the definition of the pseudo-class and its constructor.

adding Methods to the Object

One of the most common features one expects from a class is the ability to define behaviors with methods. In JavaScript this is relatively easy to do syntactically.

To define a method in an object's function one can either define it internally, or use a reference to a function defined outside the class. External definitions can quickly cause namespace conflict issues, since all method names must remain conflict free with all other methods for other classes. For this reason, one technique for adding a method inside of a class definition is by assigning an anonymous function to a variable, as shown in Listing 15.1.2.

With this method so defined, all dice objects can call the `randomRoll` function, which will return one of the six faces defined in the Die constructor.

```
function Die(col) {
    this.color=col;
    this.faces=[1,2,3,4,5,6];

    // define method randomRoll as an anonymous function
    this.randomRoll = function() {
        var randNum = Math.floor((Math.random() * this.faces.length)+ 1);
        return faces[randNum-1];
    };
}
```

Listing 15.1.2 Die pseudo-class with an internally defined method

```
var oneDie = new Die("0000FF");
console.log(oneDie.randomRoll() + " was rolled");
```

Although this mechanism for methods is effective, it is not a memory-efficient approach because each inline method is redefined for each new object. Unlike a PHP or Java class, an anonymous function in JavaScript is not defined once. Figure 15.1.1 illustrates how two instances of a Die object define two (identical) definitions of the randomRoll method.

Just imagine if you had 100 Die objects created; you would be redefining every method 100 times, which could have a noticeable effect on client execution speeds and browser responsiveness. To prevent this needless waste of memory, a better approach is to define the method just once using a *prototype* of the class.

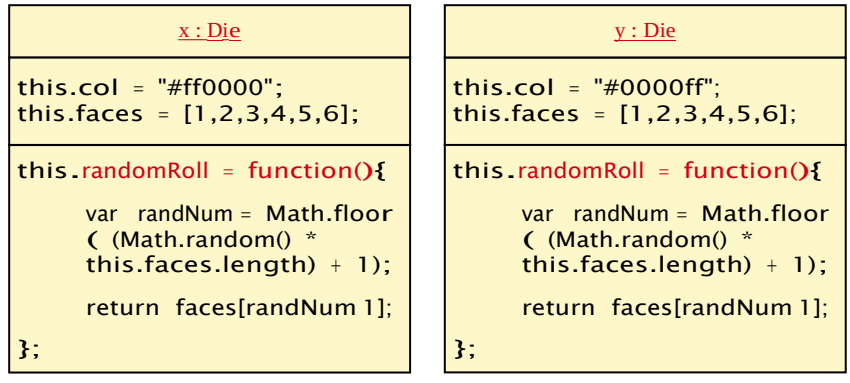

Figure 15.1.1 Illustrating duplicated method definition

15.1.1.3 Using prototypes

Prototypes are an essential syntax mechanism in JavaScript, and are used to make JavaScript behave more like an object-oriented language. The prototype properties and methods are defined *once* for all instances of an *object*. So now you can modify the definition of the randomRoll() method once again, by changing our *Die* in Listing 15.1.2 to that in Listing 15.1.3 by moving the randomRoll() method into the prototype.

```
// Start Die Class
function Die(col) {
   this.color=col;
   this.faces=[1,2,3,4,5,6];
}

Die.prototype.randomRoll = function() {
   var randNum = Math.floor((Math.random() * this.faces.length) + 1);
   return faces[randNum-1];
};
// End Die Class
```

Listing 15.1.3 The Die pseudo-class using the prototype object to define methods

This definition is better because it defines the method only once, no matter how many instances of Die are created. In contrast to the duplicated code in Figure 15.1.1, Figure 15.1.2 shows how the prototype object (not class) is updated to contain the method so that subsequent instantiations (x and y) reference that one-method

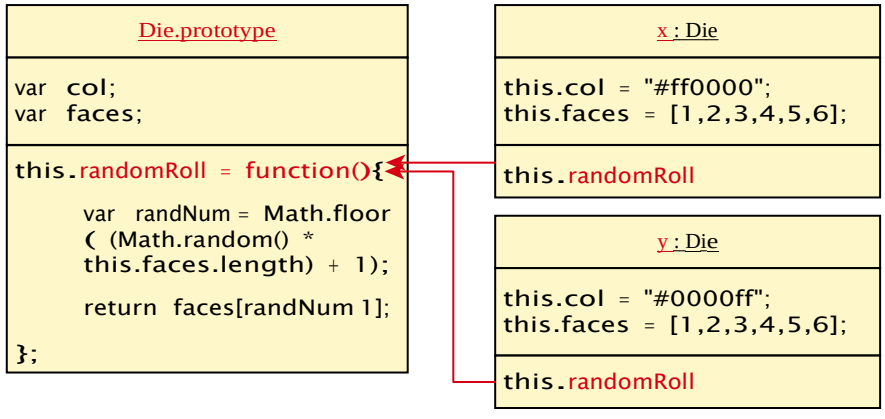

Figure 15.1.2 Illustration of JavaScript prototypes as pseudo-classes

definition. Since all instances of a Die share the same prototype object, the function declaration only happens one time and is shared with all Die instances.

More about prototypes

Even experienced JavaScript programmers sometimes struggle with the prototype concept. It should be known that every object (and method) in JavaScript has a prototype.

A prototype is an object from which other objects inherit.

The above definition sounds almost like a class in an object-oriented language, except that a prototype is itself an *object*, whereas in other oriented-oriented languages a class is an abstraction, not an object. Despite this distinction, you can make use of a function's prototype object, and assign properties or methods to it that are then available to any new objects that are created.

In addition to the obvious application of prototypes to our own pseudo-classes, prototypes enable you to *extend* existing classes by adding to their prototypes! Imagine a method added to the String object, which allows you to count instances of a character. Listing 15.1.4 defines just such a method, named countChars, that takes a character as a parameter.

```
String.prototype.countChars = function (c){
   var count=0;
   for (var i=0;i<this.length;i++){
      if (this.charAt(i) == c)
         count++;
   }
   return count;
}
```

listing 15.1.4 Adding a method named countChars to the String class

Now any new instances of String will have this method available to them (created using the new keyword), while existing strings will not. You could use the new method on any strings instantiated after the prototype definition was added. For instance the following example will output Hello World has 3 letter l's.

```
var hel = "Hello World";
console.log(hel + "has" + hel.countChars("l") + " letter l's");
```

This technique is also useful to assign properties to a pseudo-class that you want available to all instances. Imagine an array of all the *valid* characters attached to some custom string class. Again using prototype you could define such a list.

```
CustomString.prototype.validChars = ["A","B","C"];
```

Prototypes are certainly one of the hardest syntactic mechanisms to learn in JavaScript and are a poor choice for teaching object-oriented design to students. You must, however, understand and make use of them: even helpful frameworks like jQuery make extensive use of prototypes.

15.1.2 jQuery Foundations

A **library** or **framework** is software that you can utilize in your own software, which provides some common implementations of standard ideas. A web framework can be expected to have features related to the web including HTTP headers, AJAX, authentication, DOM manipulation, cross-browser implementations, and more.

jQuery's beginnings date back to August 2005, when jQuery founder John Resig was looking into how to better combine CSS selectors with succinct JavaScript notation.[1] Within a year, AJAX and animations were added, and the project has been improving ever since. Additional modules (like the popular jQuery UI extension and recent additions for mobile device support) have considerably extended jQuery's abilities. Many developers find that once they start using a framework like jQuery, there's no going back to "pure" JavaScript because the framework offers so many useful shortcuts and succinct ways of doing things. **jQuery** is now the most popular JavaScript library currently in use as supported by the statistics in Figure 15.1.3.

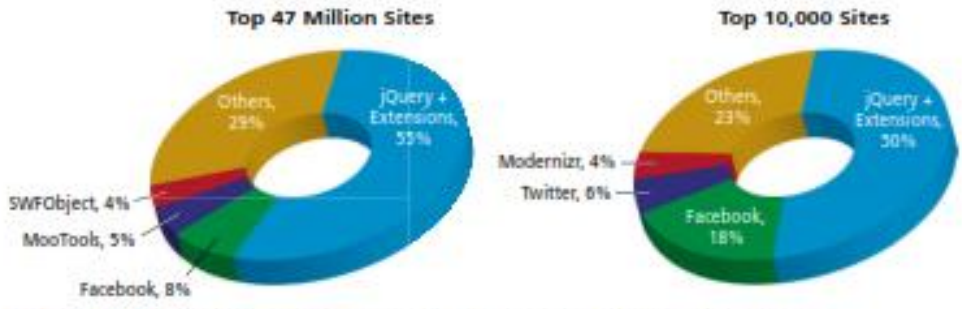

FIGURE 15.3 Comparison of the most popular JavaScript frameworks (data courtesy of BuiltWith.com)

jQuery bills itself as the *write less, do more* framework.[2] According to its website

> *jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.*

To really benefit from jQuery, you must understand how and why it replaces some JavaScript techniques and code regarding selectors, attributes, and AJAX with more succinct syntax that also includes improvements and enhancements. It should be noted that ideas and syntax learned in Chapter 6 will be used since jQuery is still JavaScript and must make use of the loops, conditionals, variables, and prototypes of that language.

15.1.2.1 including jQuery in Your page

Since the entire library exists as a source JavaScript file, importing jQuery for use in your application is as easy as including a link to a file in the `<head>` section of your HTML page. You must either link to a locally hosted version of the library or use an approved third-party host, such as Google, Microsoft, or jQuery itself.

Using a third-party **content delivery network (CDN)** is advantageous for several reasons. Firstly, the bandwidth of the file is offloaded to reduce the demand on your servers. Secondly, the user may already have cached the third-party file and thus not have to download it again, thereby reducing the total loading time. This probability is increased when using a CDN like Google rather than a developer-focused CDN like jQuery.

A disadvantage to the third-party CDN is that your jQuery will fail if the third-party host fails, although that is unlikely given the mission-critical demands of large companies like Google and Microsoft.

To achieve the benefits of the CDN and increase reliability on the rare occasion it might be down, you can write a small piece of code to check if the first attempt to load jQuery was successful. If not, you can load the locally hosted version. This setup should be included in the `<head>` section of your HTML page as shown in Listing 15.1.5.

```
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script type="text/javascript">
window.jQuery ||
document.write('<script src="/jquery-1.9.1.min.js"><\/script>');
</script>
```

Listing 15.1.5 jQuery loading using a CDN and a local fail-safe if the CDN is offline

15.1.2.2 jQuery selectors

Selectors were first covered in Chapter 6, when we introduced the getElement-ByID() and querySelector() functions in JavaScript (they were also covered back in Chapter 3, when CSS was introduced). Selectors offer the developer a way of accessing and modifying a DOM object from an HTML page in a simple way. Although the advanced querySelector() methods allow selection of DOM elements based on CSS selectors, it is only implemented in newest browsers. To address this issue jQuery introduces its own way to select an element, which under the hood supports a myriad of older browsers for you! jQuery builds on the CSS selectors and adds its own to let you access elements as you would in CSS or using new shortcut methods.

The relationship between DOM objects and selectors is so important in JavaScript programming that the pseudo-class bearing the name of the framework, jQuery(), lets programmers easily access DOM objects using selectors passed as parameters. Because it is used so frequently, it has a shortcut notation and can be written as $(). This $() syntax can be confusing to PHP developers at first, since in PHP the $ symbol indicates a variable. Nonetheless jQuery uses this shorthand frequently, and we will use this shorthand notation throughout this book.

You can combine CSS selectors with the $() notation to select DOM objects that match CSS attributes. Pass in the string of a CSS selector to $() and the result will be the set of DOM objects matching the selector. You can use the basic selector syntax from CSS, as well as some additional ones defined within jQuery.

The selectors always return arrays of results, rather than a single object. This is easy to miss since we can apply operations to the set of DOM objects matched by the selector. For instance, sometimes in the examples you will see the *0*th element referenced using the familiar [0] syntax. This will access the first DOM object that matches the selector, which we can then drill down into to access other attributes and properties.

basic selectors

The four basic selectors were defined back in Chapter 3, and include the universal selector, class selectors, id selectors, and elements selectors. To review:

- $("*") **Universal selector** matches all elements (and is slow).
- $("tag") **Element selector** matches all elements with the given element name.
- $(".class") **Class selector** matches all elements with the given CSS class.
- $("#id") **Id selector** matches all elements with a given HTML id attribute.

For example, to select the single element with id="grab" you would write:

```
var singleElement = $("#grab");
```

To get a set of all the <a> elements the selector would be:

```
var allAs = $("a");
```

These selectors are powerful enough that they can replace the use of getElementById() entirely.

The implementation of selectors in jQuery purposefully mirrors the CSS specification, which is especially helpful since CSS is something you have learned and used throughout this book.

In addition to these basic selectors, you can use the other CSS selectors that were covered in Chapter 3: attribute selectors, pseudo-element selectors, and contextual selectors as illustrated in Figure 15.1.4. The remainder of this section reviews some of these selectors and how they are used with jQuery.

attribute selector

An **attribute selector** provides a way to select elements by either the presence of an element attribute or by the value of an attribute. Chapter 3 mentioned that not all

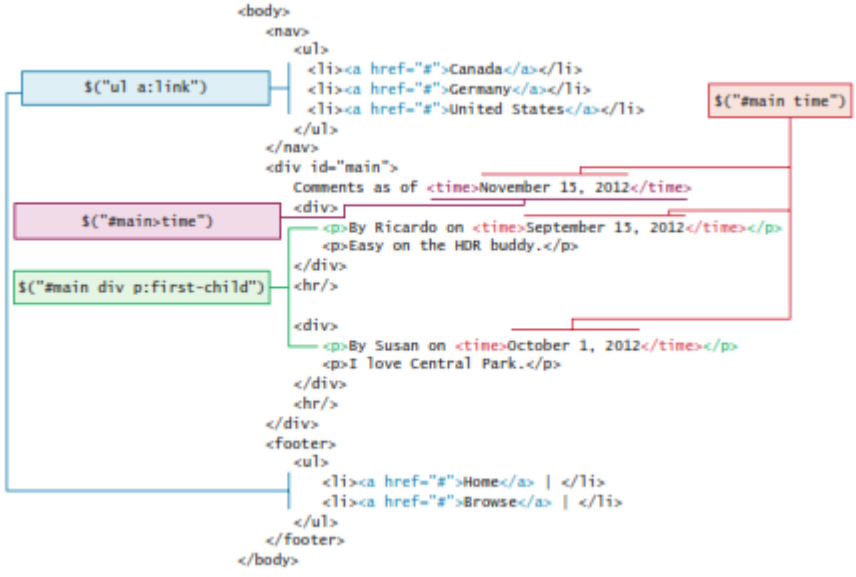

FIGURE 15.4 Illustration of some jQuery selectors and the HTML being selected

browsers implemented it. jQuery overcomes those browser limitations, providing the ability to select elements by attribute. A list of sample CSS attribute selectors was given in Chapter 3 (Table 3.4), but to jog your memory with an example, consider a selector to grab all <img> elements with an src attribute beginning with /artist/ as:

```
var artistImages = $("img[src^='/artist/']");
```

Recall that you can select by attribute with square brackets ([attribute]), specify a value with an equals sign ([attribute=value]) and search for a particular value in the beginning, end, or anywhere inside a string with ^, $, and * symbols respectively ([attribute^=value], [attribute$=value], [attribute*=value]).

pseudo-element selector

Pseudo-elements are special elements, which are special cases of regular ones. As you may recall from Chapter 3, these **pseudo-element selectors** allow you to append to any selector using the colon and one of :link, :visited, :focus, :hover, :active, :checked, :first-child, :first-line, and :first-letter.

These selectors can be used in combination with the selectors presented above, or alone. Selecting all links that have been visited, for example, would be specified with:

```
var visitedLinks = $("a:visited");
```

Since this chapter reviews and builds on CSS selectors, you are hopefully remembering some of the selectors you have used earlier and are making associations between those selectors and the ones in jQuery. As you already know from Chapter 6, once you have the ability to select an element, you can do many things to manipulate that element from changing its content or style all the way to removing it.

Contextual selector

Another powerful CSS selector included in jQuery's selection mechanism is the **contextual selectors** introduced in Chapter 3. These selectors allowed you to specify elements with certain relationships to one another in your CSS. These relationships included descendant (space), child (>), adjacent sibling (+), and general sibling (~).

To select all <p> elements inside of elements you would write

```
var para = $("div p");
```

Content Filters

The **content filter** is the only jQuery selector that allows you to append filters to all of the selectors you've used thus far and match a particular pattern. You can select

FIGURE 15.5 An illustration of jQuery's content filter selector

elements that have a particular child using :has(), have no children using :empty, or match a particular piece of text with :contains(). Consider the following example:

```
var allWarningText = $("body *:contains('warning')");
```

It will return a list of all the DOM elements with the word *warning* inside of them. You might imagine how we may want to highlight those DOM elements by coloring the background red as shown in Figure 15.1.5 with one line of code:

```
$("body *:contains('warning')").css("background-color", "#aa0000");
```

Form selectors

Since form HTML elements are well known and frequently used to collect and transmit data, there are jQuery selectors written especially for them. These selectors, listed in Table 15.1.1, allow for quick access to certain types of field as well as fields in certain states.

15.1.2.3 jQuery attributes

With all of the selectors described in this chapter, you can select any set of elements that you want from a web page. In order to understand how to fully manipulate the elements you now have access to, one must understand an element's *attributes* and *properties*.

htML attributes

The core set of attributes related to DOM elements are the ones specified in the HTML tags described in Chapter 2. You have by now integrated many of the key

selector	Css equivalent	Description
$(:button)	$("button, input[type='button']")	Selects all *buttons*.
$(:checkbox)	$('[type=checkbox]')	Selects all *checkboxes*.
$(:checked)	No equivalent	Selects elements that are checked. This includes radio buttons and checkboxes.
$(:disabled)	No equivalent	Selects form elements that are disabled. These could include <button>, <input>, <optgroup>, <option>, <select>, and <textarea>
$(:enabled)	No equivalent	Opposite of :disabled. It returns all elements where the disabled attribute=false as well as form elements with no disabled attribute.
$(:file)	$('[type=file]')	Selects all elements of type file.
$(:focus)	$(document.activeElement)	The element with focus.
$(:image)	$('[type=image]')	Selects all elements of type image.
$(:input)	No equivalent	Selects all <input>, <textarea>, <select>, and <button> elements.
$(:password)	$('[type=password]')	Selects all password fields.
$(:radio)	$('[type=radio]')	Selects all radio elements.
$(:reset)	$('[type=reset]')	Selects all the reset buttons.
$(:selected)	No equivalent	Selects all the elements that are currently selected of type <option>. It does not include checkboxes or radio buttons.
$(:submit)	$('[type=submit]')	Selects all submit input elements.
$(:text)	No equivalent	Selects all input elements of type text. $('[type=text]') is almost the same, except that $(:text) includes <input>. fields with no type specified.

table 15.1.1 jQuery form selectors and their CSS equivalents when applicable

attributes like the href attribute of an <a> tag, the src attribute of an <img>, or the class attribute of most elements.

In jQuery we can both set and get an attribute value by using the **attr()** method on any element from a selector. This function takes a parameter to specify which attribute, and the optional second parameter is the value to set it to. If no second parameter is passed, then the return value of the call is the current value of the attribute. Some example usages are:

```
// var link is assigned the href attribute of the first <a> tag
var link = $("a").attr("href");

// change all links in the page to http://funwebdev.com
$("a").attr("href","http://funwebdev.com");

// change the class for all images on the page to fancy
$("img").attr("class","fancy");
```

htML properties

Many HTML tags include properties as well as attributes, the most common being the *checked* property of a radio button or checkbox. In early versions of jQuery, HTML properties could be set using the **attr()** method. However, since properties are not technically attributes, this resulted in odd behavior. The **prop()** method is now the preferred way to retrieve and set the value of a property although, **attr()** may return some (less useful) values.

To illustrate this subtle difference, consider a DOM element defined by

```
<input class ="meh" type="checkbox" checked="checked">
```

The value of the **attr()** and **prop()** functions on that element differ as shown below.

```
var theBox = $(".meh"); theBox.prop("checked")
// evaluates to TRUE theBox.attr("checked") //
evaluates to "checked"
```

Changing Css

Changing a CSS style is syntactically very similar to changing attributes. jQuery provides the extremely intuitive **css()** methods. There are two versions of this method (with two different method signatures), one to get the value and another to set it. The first version takes a single parameter containing the CSS attribute whose value you want and returns the current value.

```
$color = $("#colourBox").css("background-color"); // get the color
```

To modify a CSS attribute you use the second version of **css()**, which takes two parameters: the first being the CSS attribute, and the second the value.

```
// set color to red
$("#colourBox").css("background-color", "#FF0000");
```

If you want to use classes instead of overriding particular CSS attributes individually, have a look at the additional shortcut methods described in the jQuery documentation.

shortcut Methods

jQuery allows the programmer to rely on foundational HTML attributes and properties exclusively as described above. However, as with selectors, there are additional functions that provide easier access to common operations such as changing an object's class or the text within an HTML tag.

The **html()** method is used to get the HTML contents of an element (the part between the <> and </> tags associated with the innerHTML property in JavaScript). If passed with a parameter, it updates the HTML of that element.

The **html()** method should be used with caution since the inner HTML of a DOM element can itself contain nested HTML elements! When replacing DOM with text, you may inadvertently introduce DOM errors since no validation is done on the new content (the browser wouldn't want to presume).

You can enforce the DOM by manipulating textNode objects and adding them as children to an element in the DOM tree rather than use **html()**. While this enforces the DOM structure, it does complicate code. To illustrate, consider that you could replace the content of every <p> element with "jQuery is fun," with the one line of code:

```
$("p").html("jQuery is fun");
```

The shortcut methods addClass(className) / removeClass(className) add or remove a CSS class to the element being worked on. The className used for these functions can contain a space-separated list of classnames to be added or removed.

The hasClass(classname) method returns true if the element has the className currently assigned. False, otherwise. The toggleClass(className) method will add or remove the class className, depending on whether it is currently present in the list of classes. The val() method returns the value of the element. This is typically used to retrieve values from input and select fields.

15.1.2.4 jQuery Listeners

Just like JavaScript, jQuery supports creation and management of listeners/handlers for JavaScript events. The usage of these events is conceptually the same as with JavaScript with some minor syntactic differences.

set Up after page Load

In JavaScript, you learned why having your **listeners** set up inside of the window.onload() event was a good practice. Namely, it ensured the entire page and all DOM elements are loaded before trying to attach listeners to them. With jQuery we do the same thing but use the $(document).ready() event as shown in Listing 15.1.6.

```
$(document).ready(function(){
 //set up listeners on the change event for the file items.
 $("input[type=file]").change(function(){
     console.log("The file to upload is "+ this.value);
  });
});
```

listing 15.1.6 jQuery code to listen for file inputs changing, all inside the document's ready event

What is really happening is we are attaching our code to the **handler** for the document.ready event, which triggers when the page is fully downloaded and parsed into its DOM representation.

Listener Management

Setting up listeners for particular events is done in much the same way as JavaScript. While pure JavaScript uses the addEventListener() method, jQuery has on() and off() methods as well as shortcut methods to attach events. Modifying the code in Listing 15.1.6 to use listeners rather than one handler yields the more modular code in Listing 15.1.7. Note that the shortcut :file selector is used in place of the equiva- lent input[type=file].

```
$(document).ready(function(){
    $(":file").on("change",alertFileName); // add listener
});
// handler function using this
function alertFileName() {
    console.log("The file selected is: "+this.value);
}
```

listing 15.1.7 Using the listener technique in jQuery with on and off methods

Listeners in jQuery become especially necessary once we start using AJAX since the advanced handling of those requests and responses can get quite complicated, and well-structured code using listeners will help us better manage that complexity.

15.1.2.5 Modifying the DOM

jQuery comes with several useful methods to manipulate the DOM elements themselves. We have already seen how the `html()` function can be used to manipulate the inner contents of a DOM element and how `attr()` and `css()` methods can modify the internal attributes and styles of an existing DOM element.

Creating DOM and textNodes

If you decide to think about your page as a DOM object, then you will want to manipulate the tree structure rather than merely manipulate strings. Thankfully, jQuery is able to convert strings containing valid DOM syntax into DOM objects automatically.

Recall that the basic act of creating a DOM node in JavaScript uses the `createElement()` method:

```
var element = document.createElement('div'); //create a new DOM node
```

However, since the jQuery methods to manipulate the DOM take an HTML string, jQuery objects, or DOM objects as parameters, you might prefer to define your element as

```
var element = $("<div></div>"); //create new DOM node based on html
```

This way you can apply all the jQuery functions to the object, rather than rely on pure JavaScript, which has fewer shortcuts. If we consider creation of a simple `<a>` element with multiple attributes, you can see the comparison of the JavaScript and jQuery techniques in Listing 15.1.8.

```
// pure JavaScript way
var jsLink = document.createElement("a");
jsLink.href = "http://www.funwebdev.com";
jsLink.innerHTML = "Visit Us";
jsLink.title = "JS";

// jQuery way
var jQueryLink = $("<a href='http://funwebdev.com'
                       title = 'jQuery'>Visit Us</a>");

// jQuery long-form way
var jQueryVerboseLink = $("<a></a>");
jQueryVerboseLink.attr("href",'http://funwebdev.com');
jQueryVerboseLink.attr("title","jQuery verbose");
jQueryVerboseLink.html("Visit Us");
```

Listing 15.1.8 A comparison of node creation in JS and jQuery

prepending and appending DOM elements

When an element is defined in any of the ways described above, it must be inserted into the existing DOM tree. You can also insert the element into several places at once if you desire, since selectors can return an array of DOM elements.

The append() method takes as a parameter an HTML string, a DOM object, or a jQuery object. That object is then added as the last child to the element(s) being selected. In Figure 15.1.6 we can see the effect of an append() method call. Each ele- ment with a class of linkOut has the jsLink element defined in Listing 15.1.8 appended to it.

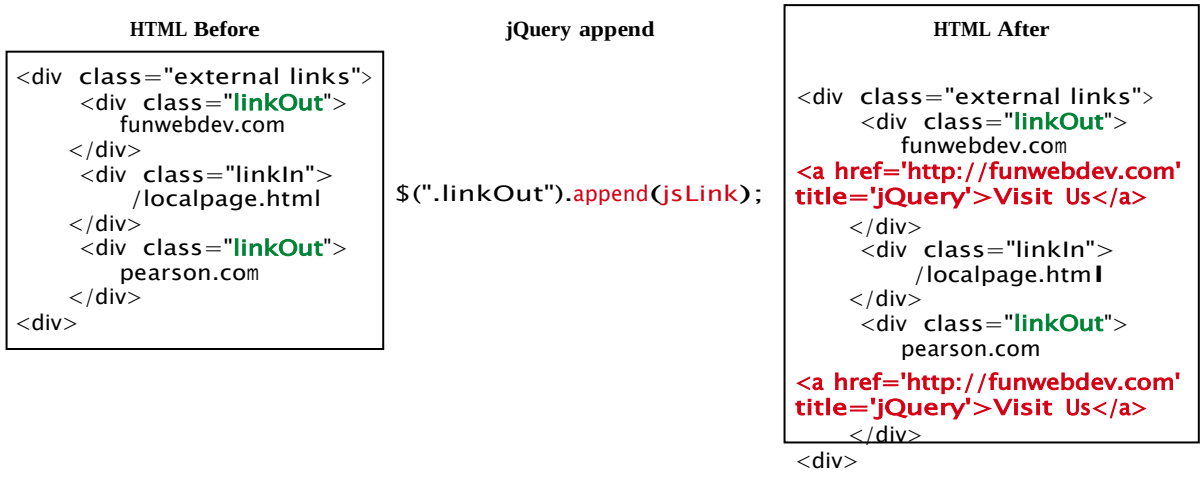

Figure 15.1.6 Illustration of where append adds a node

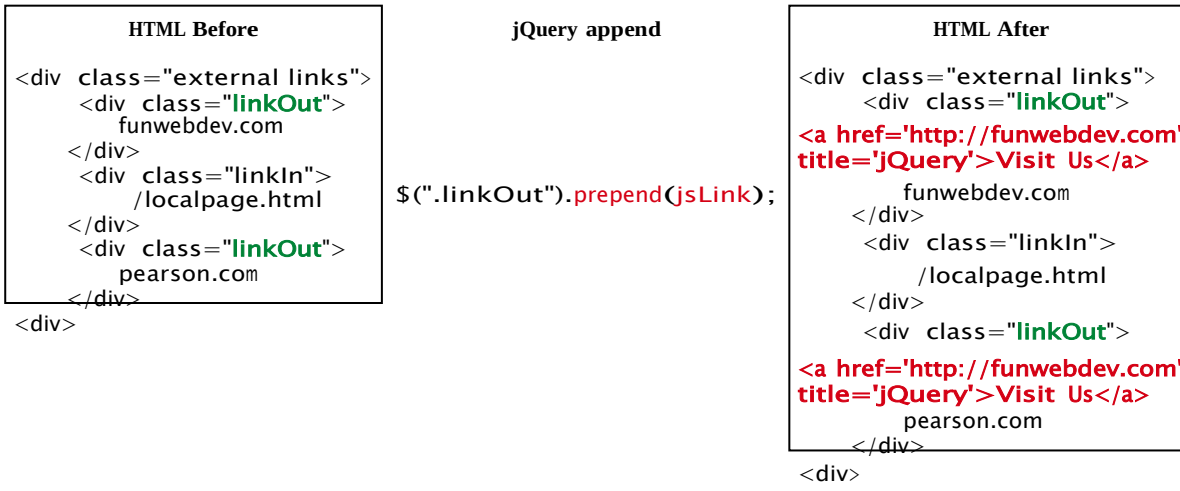

Figure 15.1.7 Illustration of prepend() adding a node

The appendTo() method is similar to append() but is used in the syntactically converse way. If we were to use appendTo(), we would have to switch the object making the call and the parameter to have the same effect as the previous code:

```
jsLink.appendTo($(".linkOut"));
```

The prepend() and prependTo() methods operate in a similar manner except that they add the new element as the first child rather than the last. See Figure 15.1.7 for an illustration of what happens with prepend().

Wrapping existing DOM in New tags

One of the most common ways you can enhance a website that supports JavaScript is to add new HTML tags as needed to support some jQuery functions. Imagine for illustration purposes our art galleries being listed alongside some external links as described by the HTML in Listing 15.1.9.

```
<div class="external-links">
  <div class="gallery">Uffuzi Museum</div>
  <div class="gallery">National Gallery</div>
  <div class="link-out">funwebdev.com</div>
</div>
```

Listing 15.1.9 HTML to illustrate DOM manipulation

If we wanted to wrap all the gallery items in the whole page inside, another (perhaps because we wish to programmatically manipulate these items later) with class galleryLink we could write:

```
$(".gallery").wrap('<div class="galleryLink"/>');
```

which modifies the HTML to that shown in Listing 15.1.10. Note how each and every link is wrapped in the correct opening and closing and uses the galleryLink class.

```
<div class="external-links">
  <div class="galleryLink">
      <div class="gallery">Uffuzi Museum</div>
  </div>
   <div class="galleryLink">
      <div class="gallery">National Gallery</div>
  </div>
   <div class="link-out">funwebdev.com</div>
</div>
```

Listing 15.1.10 HTML from Listing 15.1.9 modified by executing the wrap statement above

In a related demonstration of how succinctly jQuery can manipulate HTML, consider the situation where you wanted to add a **title** element to each element that reflected the unique contents inside. To achieve this more sophisticated manipulation, you must pass a function as a parameter rather than a tag to the wrap() method, and that function will return a dynamically created element as shown in Listing 15.1.11.

```
$(".contact").wrap(function(){
  return "<div class='galleryLink' title='Visit" + $(this).html() +
         "'></div>";
});
```

Listing 15.1.11 Using wrap() with a callback to create a unique div for every matched element

The wrap() method is a callback function, which is called for each element in a set (often an array). Each element then becomes **this** for the duration of one of the wrap() function's executions, allowing the unique **title** attributes as shown in Listing 15.1.12.

```
<div class="external-links">
  <div class="galleryLink" title="Visit Uffuzi Museum">
     <div class="gallery">Uffuzi Museum</div>
  </div>
  <div class="galleryLink" title="Visit National Gallery">
     <div class="gallery">National Gallery</div>
  </div>
  <div class="link-out">funwebdev.com</div>
</div>
```

listing 15.1.12 The modified HTML from Listing 15.1.9 after executing using wrap code from Listing 15.1.11

As with almost everything in jQuery, there is an inverse method to accomplish the opposite task. In this case, unwrap() is a method that does not take any parameters and whereas wrap() *added* a parent to the selected element(s), unwrap() *removes* the selected item's parent.

Other methods such as wrapAll() and wrapInner() provide additional controls over wrapping DOM elements. The details of those methods can be found in the online jQuery documentation.[3]

15.1.3 ajax

Asynchronous JavaScript with XML (AJAX) is a term used to describe a paradigm that allows a web browser to send messages back to the server without interrupting the flow of what's being shown in the browser. This makes use of a browser's multi-threaded design and lets one thread handle the browser and interactions while other threads wait for responses to asynchronous requests.

Figure 15.1.8 annotates a UML sequence diagram where the white activity bars illustrate where computation is taking place. Between the request being sent and the response being received, the system can continue to process other requests from the client, so it does not appear to be waiting in a loading state.

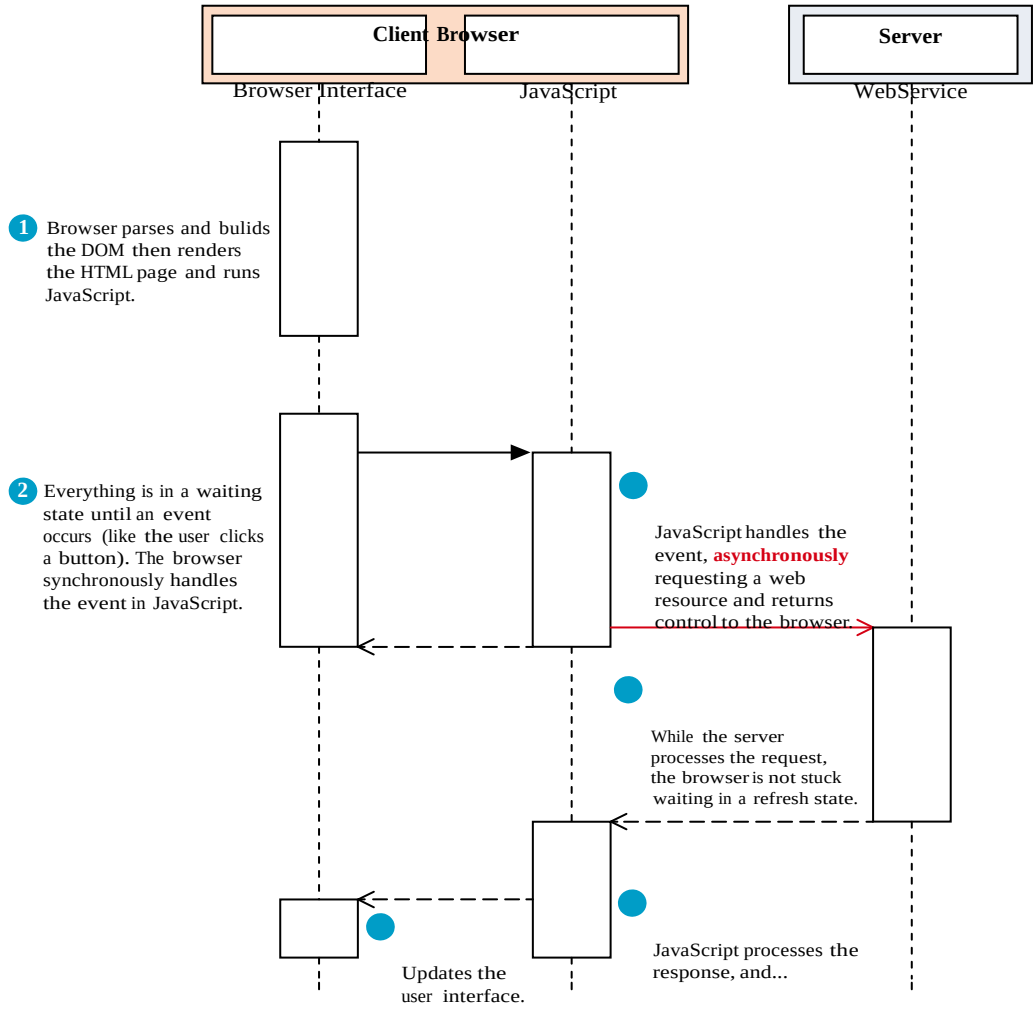

Figure 15.1.8 UML sequence diagram of an AJAX request

Responses to asynchronous requests are caught in JavaScript as events. The events can subsequently trigger changes in the user interface or make additional requests. This differs from the typical synchronous requests we have seen thus far, which require the entire web page to refresh in response to a request.

Another way to contrast AJAX and synchronous JavaScript is to consider a web page that displays the current server time as illustrated in Figure 15.1.9. If imple- mented synchronously, the entire page has to be refreshed from the server just to

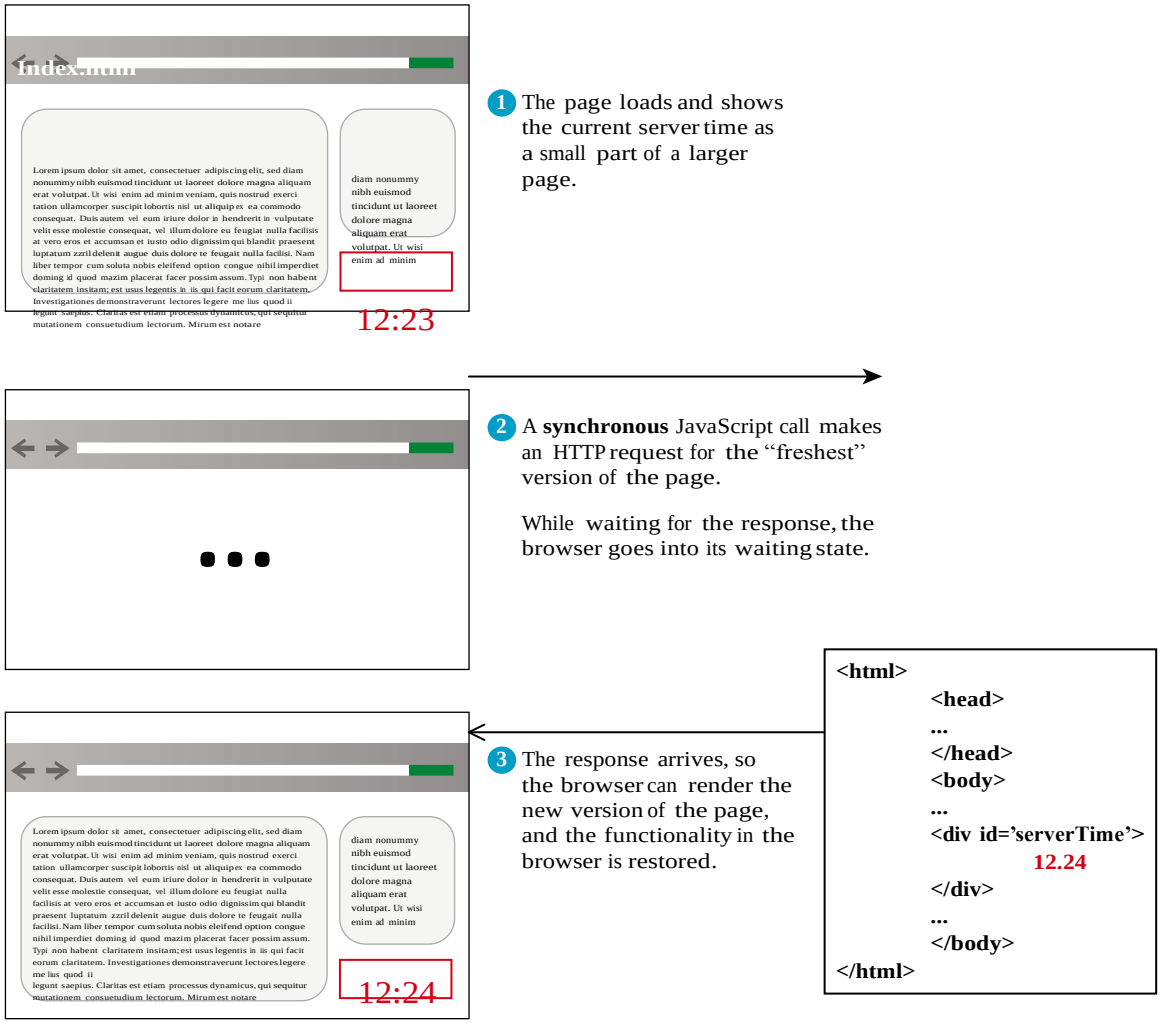

Figure 15.1.9 Illustration of a synchronous implementation of the server time web page.

update the displayed time. During that refresh, the browser enters a waiting state, so the user experience is interrupted (yes, you could implement a refreshing time using pure JavaScript, but for illustrative purposes, imagine it's essential to see the server's time).

In contrast, consider the very simple asynchronous implementation of the server time, where an AJAX request updates the server time in the background as illustrated in Figure 15.1.10.

In pure JavaScript it is possible to make asynchronous requests, but it's tricky and it differs greatly between browsers with Mozilla's `XMLHttpRequest` object and

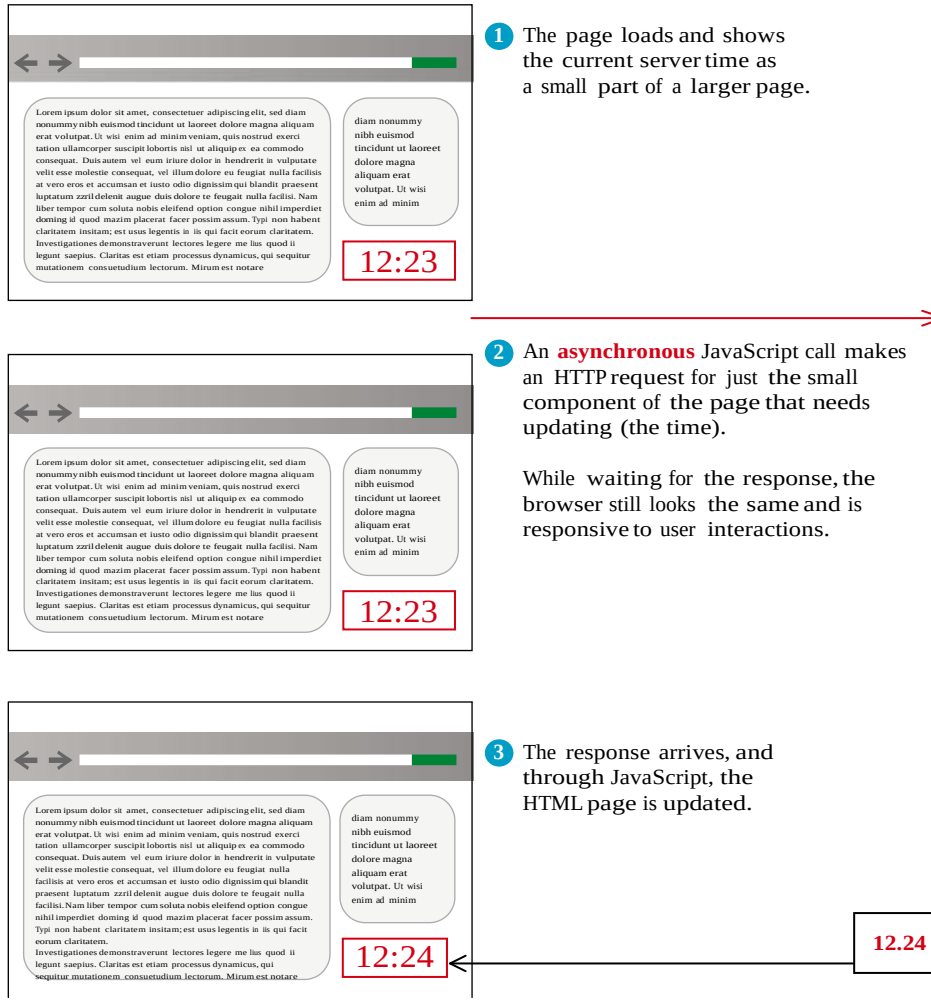

Figure 15.1.10 Illustration of an AJAX implementation of the server time widget

Internet Explorer's ActiveX wrapper. jQuery simplifies making asynchronous requests in different browsers by defining high-level methods that can work on any browser (and hiding the implementation details from the developer).

15.1.3.1 Making asynchronous requests

jQuery provides a family of methods to make asynchronous requests. We will start with the simplest GET requests, and work our way up to the more complex usage of AJAX where all variety of control can be exerted.

Consider for instance the very simple server time page described above. If the URL currentTime.php returns a single string and you want to load that value asynchronously into the element, you could write:

```
$("#timeDiv").load("currentTime.php");
```

Get requests

To illustrate the more powerful features of jQuery and AJAX, consider the more complicated scenario of a web poll where the user must choose one of the four options as illustrated in Figure 15.1.11.

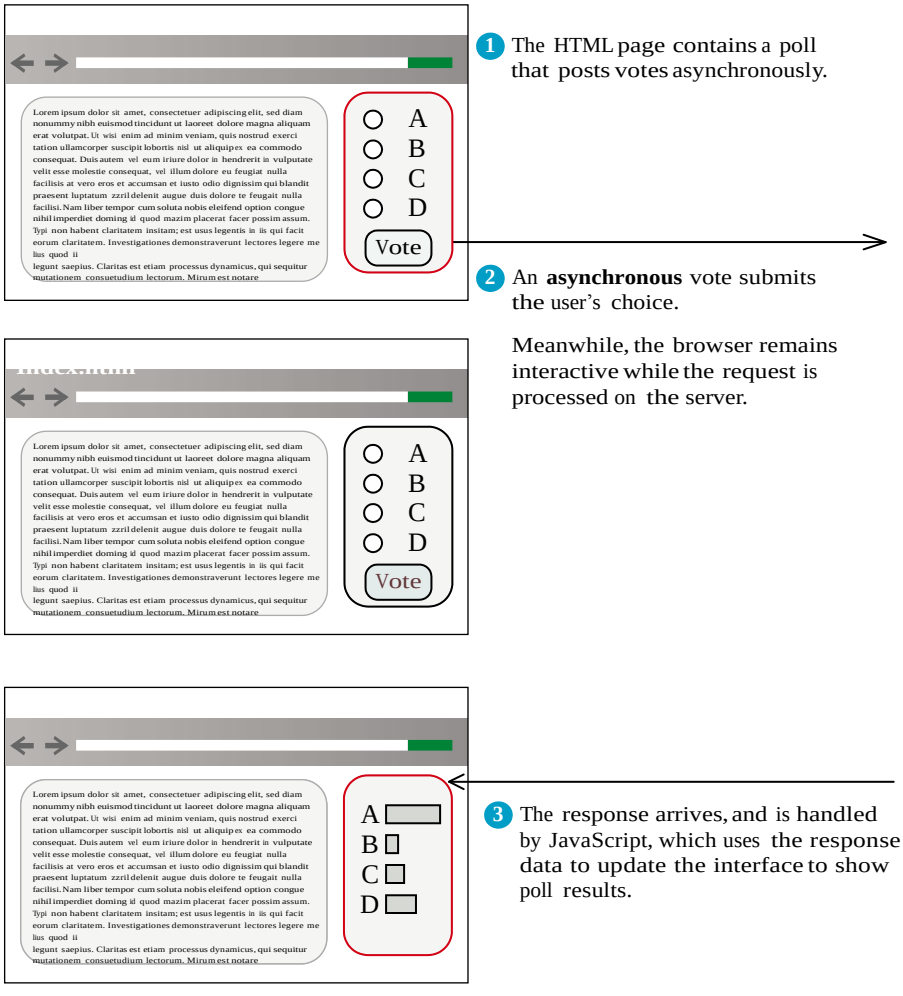

Figure 15.1.11 Illustration of a simple asynchronous web poll

Making a request to vote for option C in a poll could easily be encoded as a URL request `GET /vote.php?option=C`. However, rather than submit the whole page just to vote in the poll, jQuery's `$.get()` method sends that `GET` request asynchronously as follows:

```
$.get("/vote.php?option=C");
```

Note that the $ symbol is followed by a dot. Recall that since $ is actually shorthand for `jQuery()`, the above method call is equivalent to

```
jQuery().("/vote.php?option=C");
```

Attaching that function call to the form's submit event allows the form's default behavior to be replaced with an asynchronous `GET` request.

Although a `get()` method can request a resource very easily, handling the response from the request requires that we revisit the notion of the handler and listener.

The event handlers used in jQuery are no different than those we've seen in JavaScript, except that they are attached to the event triggered by a request completing rather than a mouse move or key press. The formal definition of the `get()` method lists one required parameter `url` and three optional ones: `data`, a callback to a `success()` method, and a `dataType`.

```
jQuery.get( url [, data ] [, success(data, textStatus, jqXHR) ]
            [, dataType ] )
```

- `url` is a string that holds the location to send the request.
- `data` is an optional parameter that is a query string or a *Plain Object*.
- `success(data,textStatus,jqXHR)` is an optional *callback* function that executes when the response is received. Callbacks are the programming term

given to placeholders for functions so that a function can be passed into another function and then called from there (called back). This callback function can take three optional parameters

- data holding the body of the response as a string.
- textStatus holding the status of the request (i.e., "success").
- jqXHR holding a jqXHR object, described shortly.

- dataType is an optional parameter to hold the type of data expected from the server. By default jQuery makes an intelligent guess between **xml**, **json**, **script,** or **html**.

In Listing 15.1.13, the callback function is passed as the second parameter to the get() method and uses the textStatus parameter to distinguish between a success- ful post and an error. The data parameter contains plain text and is echoed out to the user in an alert. Passing a function as a parameter can be an odd syntax for newcomers to jQuery.

```
$.get("/vote.php?option=C", function(data,textStatus,jsXHR) {
  if (textStatus=="success") {
     console.log("success! response is:" + data);
  }
  else {
     console.log("There was an error code"+jsXHR.status);
  }
  console.log("all done");
});
```

listing 15.1.13 jQuery to asynchronously get a URL and outputs when the response arrives

Unfortunately, if the page requested (vote.php, in this case) does not exist on the server, then the callback function does not execute at all, so the code announcing an error will never be reached. To address this we can make use of the jqXHR object to build a more complete solution.

the jqXhr Object

All of the $.get() requests made by jQuery return a **jqXHR** object to encapsulate the response from the server. In practice that means the data being referred to in the callback from Listing 15.1.13 is actually an object with backward compatibility with XMLHttpRequest. The following properties and methods are provided to conform to the XMLHttpRequest definition.

- abort() stops execution and prevents any callback or handlers from receiving the trigger to execute.
- getResponseHeader() takes a parameter and gets the current value of that header.

- readyState **is** an integer from 1 to 4 representing the state of the request. The values include 1: sending, 3: response being processed, and 4: completed.
- responseXML and/or responseText the main response to the request.
- setRequestHeader(name, value) when used before actually instantiating the request allows headers to be changed for the request.
- status is the HTTP request status codes described back in Chapter 1. (200 = ok)
- statusText is the associated description of the status code.

jqXHR objects have methods, done(), fail(), and always(), which allow us to structure our code in a more modular way than the inline callback. Figure 15.1.12 shows a representation of the various paths a request could take, and which methods are called.

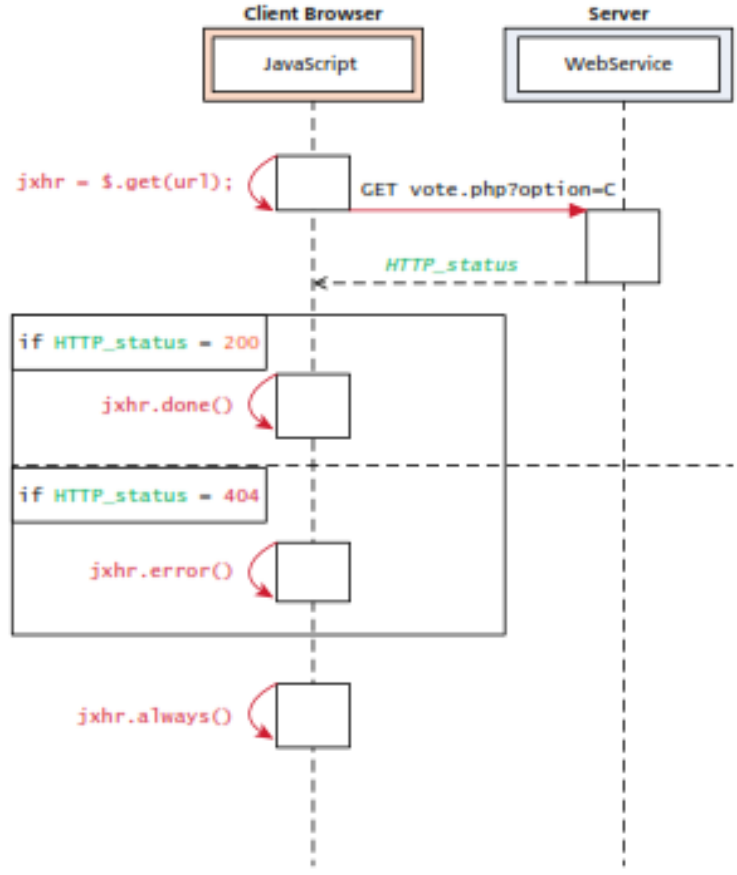

FIGURE 15.12 Sequence diagram depicting how the jqXHR object reacts to different response codes

By using these methods, the messy and incomplete code from Listing 15.1.13 becomes the more modular code in Listing 15.1.14, which also happens to work if the file was missing from the server.

```
var jqxhr = $.get("/vote.php?option=C");

jqxhr.done(function(data) { console.log(data);});
jqxhr.fail(function(jqXHR) { console.log("Error: "+jqXHR.status); })
jqxhr.always(function() { console.log("all done"); });
```

listing 15.1.14 Modular jQuery code using the jqXHR object

As we progress with AJAX in jQuery you will see that the jqXHR object is used extensively and that knowledge of it will help you develop more effective, complete code.

pOst requests

POST requests are often preferred to GET requests because one can post an unlimited amount of data, and because they do not generate viewable URLs for each action. GET requests are typically not used when we have forms because of the messy URLs and that limitation on how much data we can transmit. Finally, with POST it is possible to transmit files, something which is not possible with GET.

Although the differences between a GET and POST request are relatively minor, the HTTP 1.1 definition describes GET as a **safe method** meaning that they should not change anything, and should only read data. POSTs on the other hand are not safe, and should be used whenever we are changing the state of our system (like casting a vote). Although Listing 15.1.13 used a GET request to send our vote, it really should have used a POST to adhere to the notion of *safe* GET requests.

jQuery handles POST almost as easily as GET, with the need for an added field to hold our data. The formal definition of a jQuery `post()` request is identical to the `get()` request, aside from the method name.

```
jQuery.post( url [, data ] [, success(data, textStatus, jqXHR) ]
             [, dataType ] )
```

The main difference between a POST and a GET http request is where the data is transmitted. The data parameter, if present in the function call, will be put into the body of the request. Interestingly, it can be passed as a string (with each name=value pair separated with a "&" character) like a GET request or as a Plain Object, as with the get() method.

If we were to convert our vote casting code from Listing 15.1.14 to a POST request,

it would simply change the first line from

```
var jqxhr = $.get("/vote.php?option=C");
```

to

```
var jqxhr = $.post("/vote.php", "option=C");
```

Since jQuery can be used to submit a form, you may be interested in the short-cut method serialize(), which can be called on any form object to return its current key-value pairing as an & separated string, suitable for use with post().

Consider our simple vote-casting example. Since the poll's form has a single field, it's easy to understand the ease of creating a short query string on the fly. However, as forms increase in size this becomes more difficult, which is why jQuery includes a helper function to serialize an entire form in one step. The serialize() method can be called on a DOM form element as follows:

```
var postData = $("#voteForm").serialize();
```

With the form's data now encoded into a query string (in the postData variable), you can transmit that data through an asynchronous POST using the $.post() method as follows:

15.1.3.2 Complete Control over ajaX

It turns out both the $.get() and $.post() methods are actually shorthand forms for the jQuery().ajax() method, which allows fine-grained control over HTTP

requests. This method allows us to control many more aspects of our asynchronous JavaScript requests including the modification of headers and use of cache controls.

The `ajax()` method has two versions. In the first it takes two parameters: a URL and a Plain Object (also known as an object literal), containing any of over 30 fields. A second version with only one parameter is more commonly used, where the URL is but one of the key-value pairs in the Plain Object. The one line required to post our form using `get()` becomes the more verbose code in Listing 15.1.15.1.

```
$.ajax({ url: "vote.php",
         data: $("#voteForm").serialize(),
         async: true,
         type: post
});
```

listing 15.1.15.1 A raw AJAX method code to make a post

A complete listing of the 33 options available to you would require a chapter in itself. Some of the more interesting things you can do are send login credentials with the username and password fields. You can also modify headers using the header field, which brings us full circle to the HTTP protocol first explored in Chapter 1.

To pass HTTP headers to the `ajax()` method, you enclose as many as you would like in a Plain Object. To illustrate how you could override `User-Agent` and `Referer` headers in the `POST`, see Listing 15.1.16.

```
$.ajax({ url: "vote.php",
  data: $("#voteForm").serialize(),
  async:true,
  type: post,
  headers: {"User-Agent" : "Homebrew JavaScript Vote Engine agent",
            "Referer": "http://funwebdev.com"
           }
});
```

listing 15.1.16 Adding headers to an AJAX post in jQuery

15.1.3.3 Cross-Origin resource sharing (COrs)

As you will see when we get to Chapter 16 on security, cross-origin resource sharing (also known as cross-origin scripting) is a way by which some malicious software can gain access to the content of other web pages you are surfing despite the scripts being hosted on another domain. Since modern browsers prevent cross-origin requests by default (which is good for security), sharing content legitimately between two domains

becomes harder. By default, JavaScript requests for images on **images.funwebdev.com** from the domain **www.funwebdev.com** will result in denied requests because subdomains are considered different origins.

Cross-origin resource sharing (CORS) uses new headers in the HTML5 standard implemented in most new browsers. If a site wants to allow any domain to access its content through JavaScript, it would add the following header to all of its responses.

```
Access-Control-Allow-Origin: *
```

The browser, seeing the header, permits any cross-origin request to proceed (since * is a wildcard) thus allowing requests that would be denied otherwise (by default).

A better usage is to specify specific domains that are allowed, rather than cast the gates open to each and every domain. In our example the more precise header

```
Access-Control-Allow-Origin: www.funwebdev.com
```

will prevent all cross-site requests, except those originating from **www.funwebdev.com**, allowing content to be shared between domains as needed.

15.1.4 asynchronous File transmission

Asynchronous file transmission is one of the most powerful tools for modern web applications. In the days of old, transmitting a large file could require your user to wait idly by while the file uploaded, unable to do anything within the web interface. Since file upload speeds are almost always slower than download speeds, these transmissions can take minutes or even hours, destroying the feeling of a "real" application. Unfortunately jQuery alone does not permit asynchronous file uploads! However, using clever tricks and HTML5 additions, you too can use asynchronous file uploads.

For the following examples consider a simple file-uploading HTML form defined in Listing 15.1.17.

```
<form name="fileUpload" id="fileUpload" enctype="multipart/form-data"
      method="post" action="upload.php">
<input name="images" id="images" type="file" multiple />
<input type="submit" name="submit" value="Upload files!"/>
</form>
```

listing 15.1.17 Simple file upload form

15.1.4.1 Old iframe Workarounds

The original workaround to allow the asynchronous posting of files was to use a hidden <iframe> element to receive the posted files. Given that jQuery still does not

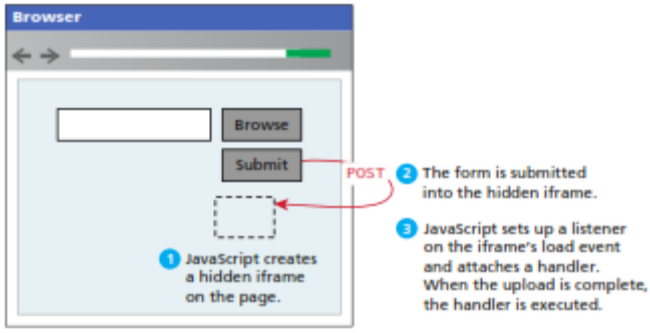

FIGURE 15.13 Illustration of posting to a hidden iframe

natively support the asynchronous uploading of files, this technique persists to this day and may be found in older code you have to maintain. As illustrated in Figure 15.1.13 and Listing 15.1.18, a hidden <iframe> allows one to post synchronously to another URL in another window. If JavaScript is enabled, you can also hide the upload button and use the change event instead to trigger a file upload. You then use the <iframe> element's onload event to trigger an action when it is done loading. When the window is done loading, the file has been received and we use the return message to update our interface much like we do with AJAX normally.

```
$(document).ready(function() {
  // set up listener when the file changes
  $(":file").on("change",uploadFile);
  // hide the submit buttons
  $("input[type=submit]").css("display","none");
});

// function called when the file being chosen changes
function uploadFile () {
  // create a hidden iframe
  var hidName = "hiddenIFrame";
  $("#fileUpload").append("<iframe id='"+hidName+"' name='"+hidName+"'
  style='display:none' src='#' ></iframe>");

  // set form's target to iframe
  $("#fileUpload").prop("target",hidName);
  // submit the form, now that an image is in it.
  $("#fileUpload").submit();
```

(*continued*)

```
    // Now register the load event of the iframe to give feedback
    $('#'+hidName).load(function() {
    var link = $(this).contents().find('body')[0].innerHTML;
    // add an image dynamically to the page from the file just uploaded
    $("#fileUpload").append("<img src='"+link+"' />");
    });
}
```

Listing 15.1.18 Hidden iFrame technique to upload files

This technique exploits the fact that browsers treat each <iframe> element as a separate window with its own thread. By forcing the post to be handled in another window, we don't lose control of our user interface while the file is uploading.

Although it works, it's a workaround using the fact that every browser can post a file synchronously. A more modular and "pure" technique would be to somehow serialize the data in the file being uploaded with JavaScript and then post it in the body of a post request asynchronously.

Thankfully, the newly redefined XMLHttpRequest Level 2 (XHR2) specification allows us to get access to file data and more through the FormData interface[4] so we can post a file as illustrated in Figure 15.1.14.

15.1.4.2 the FormData interface

Using the **FormData** interface and File API, which is part of HTML5, you no longer have to trick the browser into posting your file data asynchronously. However, you are limited to modern browsers that implement the new specification.

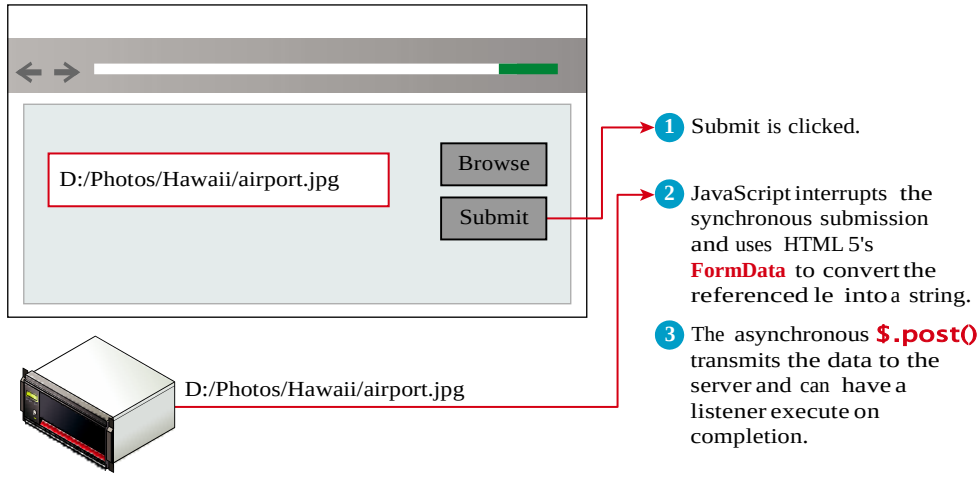

Figure 15.1.14 Posting a file using FormData

The FormData interface provides a mechanism for JavaScript to read a file from the user's computer (once they choose the file) and encode it for upload. You can use this mechanism to upload a file asynchronously. Intuitively the browser is already able to do this, since it can access file data for transmission in synchronous posts. The FormData interface simply exposes this functionality to the developer, so you can turn a file into a string when you need to.

The <iframe> method uploadFile() from Listing 15.1.18 can be replaced with the more elegant and straightforward code in Listing 15.1.19. In this pure AJAX technique the form object is passed to a FormData constructor, which is then used in the call to send() the XHR2 object. This code attaches listeners for various events that may occur.

```
function uploadFile () {
  // get the file as a string
  var formData = new FormData($("#fileUpload")[0]);

  var xhr = new XMLHttpRequest();
  xhr.addEventListener("load", transferComplete, false);
  xhr.addEventListener("error", transferFailed, false);
  xhr.addEventListener("abort", transferCanceled, false);

  xhr.open('POST', 'upload.php', true);
  xhr.send(formData);                    // actually send the form data

  function transferComplete(evt) {      // stylized upload complete
     $("#progress").css("width","100%");
     $("#progress").html("100%");
  }

  function transferFailed(evt) {
     alert("An error occurred while transferring the file.");
  }

  function transferCanceled(evt) {
     alert("The transfer has been canceled by the user.");
  }
}
```

listing 15.1.19 Using the new FormData interface from the XHR2 Specification to post files asychronously

While the code in Listing 15.1.19 works whenever the browser supports the specification, it always posts the entire form.

15.1.4.3 appending Files to a pOst

When we consider uploading multiple files, you may want to upload a single file, rather than the entire form every time. To support that pattern, you can access a single file and post it by appending the raw file to a FormData object as shown in Listing 15.1.20. The advantage of this technique is that you submit each file to the server asynchronously as the user changes it; and it allows multiple files to be trans- mitted at once.

```
var xhr = new XMLHttpRequest();
// reference to the 1st file input field
var theFile = $(":file")[0].files[0];
var formData = new FormData();
formData.append('images', theFile);
```

Listing 15.1.20 Posting a single file from a form

It should be noted that back in Listing 15.1.17 the file input is marked as **multi- ple**, and so, if supported by the browser, the user can select many files to upload at once. To support uploading multiple files in our JavaScript code, we must loop through all the files rather than only hard-code the first one. Listing 15.1.21 shows a better script than Listing 15.1.20, since it handles multiple files being selected and uploaded at once.

```
var allFiles = $(":file")[0].files;
for (var i=0;i<allFiles.length;i++){
  formData.append('images[]', allFiles[i]);
}
```

Listing 15.1.21 Looping through multiple files in a file input and appending the data for posting

The main challenge of asynchronous file upload is that your implementation must consider the range of browsers being used by your users. While the new XHR2 specification and FormData interfaces are "pure" and easy to use, they are not widely supported yet across multiple platforms and browsers, making reliance on them bad practice. Conversely the <iframe> workaround works well on more browsers, but simply feels inelegant and perhaps not worthy of your support and investment of time.

Recall, from Chapter 6, the principles of *graceful degradation* and *progressive enhancement*. These strategies guide how you design your site and regard JavaScript. How you implement features like asynchronous file upload will depend on the par- ticular strategy you've adopted for your website.

15.1.5 animation

When developers first learn to use jQuery, they are often initially attracted to the easy-to-use **animation** features. When used appropriately, these animation features can make your web applications appear more professional and engaging.

15.1.5.1 animation shortcuts

By now you've seen how jQuery provides complex (and complete) methods as well as shortcuts. Animation is no different with a raw `animate()` method and many more easy-to-use shortcuts like `fadeIn()`/`fadeOut()`, `slideUp()`/`slideDown()`. We introduce jQuery animation using the shortcuts first, then we learn about `animate()` afterward.

One of the common things done in a dynamic web page is to show and hide an element. Modifying the visibility of an element can be done using `css()`, but that causes an element to change instantaneously, which can be visually jarring. To provide a more natural transition from hiding to showing, the `hide()` and `show()` methods allow developers to easily hide elements gradually, rather than through an immediate change.

The `hide()` and `show()` methods can be called with no arguments to perform a default animation. Another version allows two parameters: the duration of the animation (in milliseconds) and a callback method to execute on completion. Using the callback is a great way to chain animations together, or just ensure elements are fully visible before changing their contents.

Listing 15.1.22 describes a simple contact form and script that builds and shows a clickable email link when you click the email icon. Hiding an email link is a com- mon way to avoid being targeted by spam bots that search for `mailto`: links in your `<a>` tags.

```
<div class="contact">
  <p>Randy Connolly</p>
  <div class="email">Show email</div>
</div>
<div class="contact">
  <p>Ricardo Hoar</p>
  <div class="email">Show email</div>
</div>
<script type='text/javascript'>
$(".email").click(function() {
    // Build email from 1st letter of first name + lastname
    // @ mtroyal.ca
    var fullName = $(this).prev().html();
    var firstName = fullName.split(" ")[0];
```

(*continued*)

```
        var address = firstName.charAt(0) + fullName.split(" ")[1] +
                      "@mtroyal.ca";

        $(this).hide();          // hide the clicked icon.
        $(this).html("<a href='mailto:"+address+"'>Mail Us</a>");
        $(this).show(1000);      // slowly show the email address.
    });
    </script>
```

Listing 15.1.22 jQuery code to build an email link based on page content and animate its appearance

A visualization of the show() method is illustrated in Figure 15.1.15.1.[5] Note that both the size and opacity are changing during the animation. Although using the very straightforward hide() and show() methods works, you should be aware of some more advanced shortcuts that give you more control.

fadein()/fadeOut()

The fadeIn() and fadeOut() shortcut methods control the opacity of an element. The parameters passed are the duration and the callback, just like hide() and show(). Unlike hide() and show(), there is no scaling of the element, just strictly control over the transparency. Figure 15.1.16 shows a span during its animation using fadeIn().

It should be noted that there is another method, fadeTo(), that takes two parameters: a duration in milliseconds and the opacity to fade to (between 0 and 1).

slideDown()/slideUp()

The final shortcut methods we will talk about are slideUp() and slideDown(). These methods do not touch the opacity of an element, but rather gradually change its height. Figure 15.1.17 shows a slideDown() animation using an email icon from **http://openiconlibrary.sourceforge.net.**

FIGURE 15.15 Illustration of the show() animation using the icon from openiconlibrary.sourceforge.net

FIGURE 15.16 Illustration of a fadeIn() animation

FIGURE 15.17 Illustration of the slideDown() animation

Y

ou should note at this point that hide() and show() are in fact a combination of both the fade and slide animations. However, different browsers may interpret these animations in slightly different ways.

toggle Methods

As you may have seen, the shortcut methods come in pairs, which make them ideal for toggling between a shown and hidden state. jQuery has gone ahead and written multiple toggle methods to facilitate exactly that. For instance, to toggle between the visible and hidden states (i.e., between using the hide() and show() methods), you can use the toggle() methods. To toggle between fading in and fading out, use the fadeToggle() method; toggling between the two sliding states can be achieved using the slideToggle() method.

Using a toggle method means you don't have to check the current state and then conditionally call one of the two methods; the toggle methods handle those aspects of the logic for you.

15.1.5.2 raw animation

Just like $.get() and $.post() methods are shortcuts for the complete $.ajax() method, the animations shown this far are all specific versions of the generic animate() method. When you want to do animation that differs from the prepackaged animations, you will need to make use of animate.

The animate() method has several versions, but the one we will look at has the following form:

```
.animate( properties, options );
```

The properties parameter contains a Plain Object with all the CSS styles of the final state of the animation. The options parameter contains another Plain Object with any of the options below set.

- always is the function to be called when the animation completes or stops with a fail condition. This function will always be called (hence the name).
- done is a function to be called when the animation completes.
- duration is a number controlling the duration of the animation.
- fail is the function called if the animation does not complete.
- progress is a function to be called after each step of the animation.

- queue is a Boolean value telling the animation whether to wait in the queue of animations or not. If false, the animation begins immediately.
- step is a function you can define that will be called periodically while the animation is still going. It takes two parameters: a now element, with the current numerical value of a CSS property, and an fx object, which is a temporary object with useful properties like the CSS attribute it represents (called *tween* in jQuery). See Listing 15.1.23 for example usage to do rotation.
- Advanced options called easing and specialEasing allow for advanced control over the speed of animation.

Movement rarely occurs in a linear fashion in nature. A ball thrown in the air slows down as it reaches the apex then accelerates toward the ground. In web development, **easing functions** are used to simulate that natural type of movement. They are mathematical equations that describe how fast or slow the transitions occur at various points during the animation.

Included in jQuery are linear and swing easing functions. Linear is a straight line and so animation occurs at the same rate throughout while swing starts slowly and ends slowly. Figure 15.1.18 shows graphs for both the linear and swing easing functions.

Easing functions are just mathematical definitions. For example, the function defining swing for values of time *t* between 0 and 1 is

$$\text{swing}(t) = \frac{1}{2} - \cos(t\pi) \div 0.5$$

The jQuery UI extension provides over 30 easing functions, including cubic functions and bouncing effects, so you should not have to define your own.

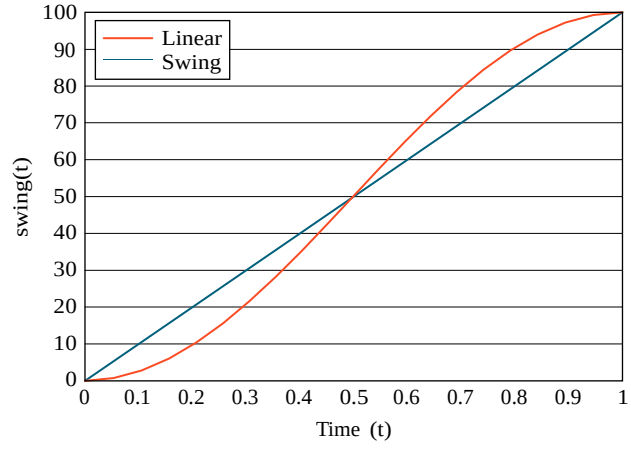

Figure 15.1.18 Visualization of the linear and swing easing functions

FIGURE 15.19 Illustration of rotation using the animate() method in Listing 15.23

An example usage of animate() is shown in Listing 15.1.23 where we apply sev- eral transformations (changes in CSS properties), including one for the text size, opacity, and a CSS3 style rotation, resulting in the animation illustrated in Figure 15.1.19.

It should be noted that step() callbacks are only made for CSS values that are numerical. This is why you often see a dummy CSS value used to control an unrelated CSS option like rotation (which has string values, not numeric values). In Listing 15.1.23 we add a margin-right CSS attribute with a value of 100 so that whenever the callback for that CSS property occurs, we can figure what percentage of the animation we are on by dividing now/100. We then use that percentage to apply the appropriate rotation (3360). If we had added the transform elements as CSS attributes, no automatic values would be calculated, no animated rotation would occur. Figure 15.1.20 illustrates the step function callbacks for our example with two calls to step functions shown for illustrative purposes. The actual number of calls to step will depend on your hardware and software configuration.

```
$(this).animate(
  // parameter one: Plain Object with CSS options.

  {opacity:"show","fontSize":"120%","marginRight":"100px"},
  // parameter 2: Plain Object with other options including a
  // step function
  {step: function(now, fx) {
        // if the method was called for the margin property
        if (fx.prop=="marginRight") {
           var angle=(now/100)*360; //percentage of a full circle
           // Multiple rotation methods to work in multiple browsers
           $(this).css("transform","rotate("+angle+"deg)");
           $(this).css("-webkit-transform","rotate("+angle+"deg)");
           $(this).css("-ms-transform","rotate("+angle+"deg)");
        }
     },
     duration:5000, "easing":"linear"
  }
);
```

Listing 15.1.23 Use of animate() with a step function to do CSS3 rotation

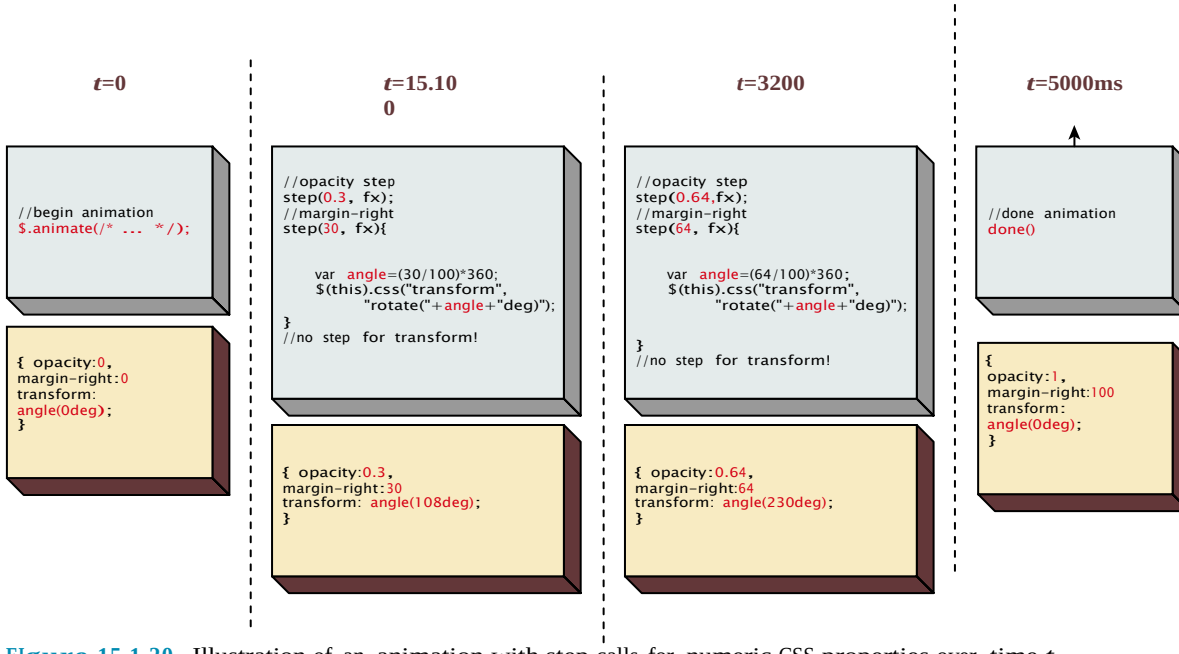

Figure 15.1.20 Illustration of an animation with step calls for numeric CSS properties over time *t*

You may ask, why use a margin-right property instead of the opacity that already goes from 0 to 1? The answer is that by attaching the rotation animation to another CSS property you care about, the two become coupled, so that modifying the opacity would modify the rotation. Decoupling the rotation from the opacity ensures both are independently controlled.

15.1.6 backbone MvC Frameworks

In working with jQuery thus far we have seen how easily we can make great animations and modularize our UI into asynchronous components. As our model gets more and more complex, it becomes challenging to keep our data decoupled from our view and the DOM. We end up with many listeners, callbacks, and server-side scripts, and it can be challenging to coordinate all the different parts together. In response, frameworks have been created that enforce a strict MVC pattern, and if used correctly can speed up development and result in maintainable modular code.

MVC frameworks (and frameworks in general) are not silver bullets that solve all your development challenges. These frameworks are overkill for small applications, where a small amount of jQuery would suffice. You will learn about the basics of Backbone so that you can consider this library or one like it before designing a large-scale web application.

15.1.6.1 Getting started with backbone.js

Backbone is an MVC framework that further abstracts JavaScript with libraries intended to adhere more closely to the MVC model as described in Chapter 14. This library is available from **http://backbonejs.org** and relies on the underscore library, available from **http://underscorejs.org/**.

In Backbone, you build your client scripts around the concept of **models**. These models are often related to rows in the site's database and can be loaded, updated, and eventually saved back to the database using a REST interface, described in Chapter 17. Rather than writing the code to connect listeners and event handlers, Backbone allows user interface components to be notified of changes so they can update themselves just by setting everything up correctly.

You must download the source for these libraries to your server, and reference them just as we've done with jQuery. Remember that the underscore library is also required, so a basic inclusion will look like:

```
<script src="underscore-min.js"></script>
<script src="backbone-min.js"></script>
```

To illustrate the application of Backbone, consider our travel website discussed throughout earlier chapters. In particular consider the management of albums, and an interface to select and publish particular albums.

The HTML shown in Listing 15.1.24 will serve as the basis for the example, with a form to create new albums and an unordered list to display them.

```
<form id="publishAlbums" method="post" action="publish.php">
  <h1>Publish Albums</h1>
  <ul id="albums">
      <!-- The albums will appear here -->
  </ul>
  <p id="totalAlbums">Count: <span>0</span></p>
  <input type="submit" id="publish" value="Publish" />
</form>
```

listing 15.1.24 HTML for an album publishing interface

The MVC pattern in Backbone will use a `Model` object to represent the `TravelAlbum`, a `Collection` object to manage multiple albums, and a `View` to render the HTML for the model, and instantiate and render the entire application as illustrated in the class diagram in Figure 15.1.21.

15.1.6.2 backbone Models

The term models can be a challenging one to apply, since authors of several frameworks and software engineering patterns already use the term.

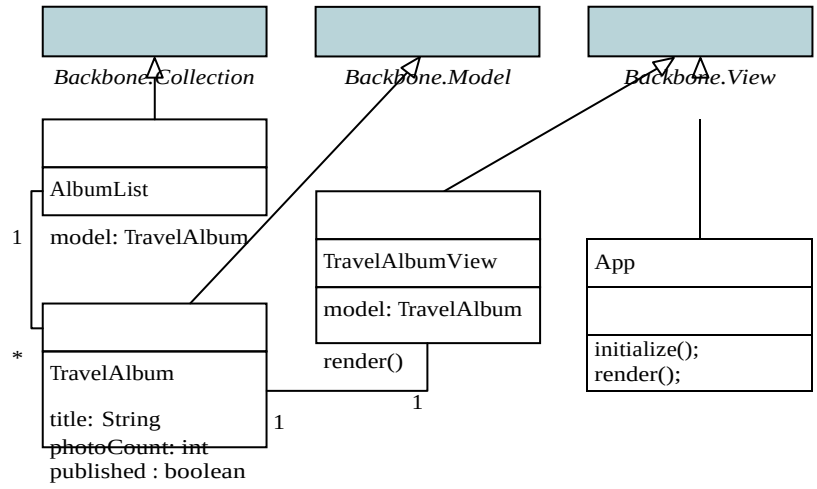

Figure 15.1.21 Illustration of Backbone Model, Collections, and Views for a Photo Album example

Backbone.js defines **models** as

> *the heart of any JavaScript application, containing the interactive data as well as a large part of the logic surrounding it: conversions, validations, computed properties, and access control.*

When using Backbone, you therefore begin by abstracting the elements you want to create models for. In our case, TravelAlbum will consist of a **title**, an image photoCount, and a Boolean value controlling whether it is published or not. The Models you define using Backbone must *extend* Backbone.Model, adding methods in the process as shown in Listing 15.1.25.

```
// Create a model for the albums
var TravelAlbum = Backbone.Model.extend({
  defaults:{
          title: 'NewAlbum',
          photoCount: 0,
          published: false
      },

      // Function to publish/unpublish
      toggle: function(){
          this.set('checked', !this.get('checked'));
      }
  });
```

Listing 15.1.25 A PhotoAlbum Model extending from Backbone.Model

15.1.6.3 Collections

In addition to models, Backbone introduces the concept of **Collections**, which are normally used to contain lists of Model objects. These collections have advanced features and like a database can have indexes to improve search performance. In Listing 15.1.26, a collection of Albums, AlbumList, is defined by extending from Backbone's Collection object. In addition an initial list of TravelAlbums, named albums, is instantiated to illustrate the creation of some model objects inside a Collection.

```
// Create a collection of albums
var AlbumList = Backbone.Collection.extend({

  // Set the model type for objects in this Collection
  model: TravelAlbum,

  // Return an array only with the published albums
  GetChecked: function(){
     return this.where({checked:true});
  }
});

// Prefill the collection with some albums.
var albums = new AlbumList([
  new TravelAlbum({ title: 'Banff, Canada', photoCount: 42}),
  new TravelAlbum({ title: 'Santorini, Greece', photoCount: 102}),
]);
```

listing 15.1.26 Demonstration of a Backbone.js Collection defined to hold PhotoAlbums

Although we now have the capacity to model albums and a collection of albums, we still have not rendered anything to the user! To facilitate this, Backbone adheres closely to a pure MVC pattern, and requires that you define a View for any DOM element you want to be auto-refreshed on certain occurrences.

15.1.6.4 views

Views allow you to translate your models into the HTML that is seen by the users. They attach themselves to methods and properties of the Collection and define methods that will be called whenever Backbone determines the view needs refreshing.

For our example we extend a View as shown in Listing 15.1.27. In that code we attach our view to a particular tagName (in our case the <li> element) and then

```
var TravelAlbumView = Backbone.View.extend({
  TagName: 'li',

  events:{
     'click': 'toggleAlbum'
  },

  initialize: function(){
      // Set up event listeners attached to change
     this.listenTo(this.model, 'change', this.render);
  },

  render: function(){
  // Create the HTML
  this.$el.html('<input type="checkbox" value="1" name="' +
                this.model.get('title') + '" /> ' +
                this.model.get('title') + '<span> ' +
                this.model.get('photoCount') + ' images</span>');
  this.$('input').prop('checked', this.model.get('checked'));

  // Returning the object is a good practice
     return this;
  },

  toggleAlbum: function() {
     this.model.toggle();
  }
});
```

listing 15.1.27 Deriving custom View objects for our model and Collection

associate the click event with a new method named toggleAlbum(). You must always override the render() method since it defines the HTML that is output.

Finally, to make this code work you must also override the render of the main application. In our case we will base it initially on the entire <body> tag, and output our content based entirely on the models in our collection as shown in Listing 15.1.28.

Notice that you are making use of several methods, _.each(), elem.get(), and this.total.text(), that have not yet been defined. Some of these methods replace jQuery functionality, while others have no analog in that framework. The _ is defined by the underscore library in much the same way as $ is defined for jQuery. If you are interested in learning more about Backbone, a complete listing of func- tions is available online.[6]

These models also allow us to save data temporarily on the client's machine as JavaScript and then post it back to the server when needed. You can leverage the

```
// The main view of the entire Backbone application
var App = Backbone.View.extend({
  // Base the view on an existing element
  el: $('body'),

  initialize: function() {
    // Define required selectors
    this.total = $('#totalAlbums span');
    this.list = $('#albums');

    // Listen for the change event on the collection.
    this.listenTo(albums, 'change', this. render);

    // Create views for every one of the albums in the collection
    albums.each(function(album) {
      var view = new TravelAlbumView({ model: album });
          this.list.append(view.render().el);
      }, this);  // "this" is the context in the callback
    },

    render: function(){

      // Calculate the count of published albums and photos
        var total = 0; var photos = 0;

        _.each(albums.getChecked(), function(elem) {
          total++;
          photos+= elem.get("photoCount");
        });

      // Update the total price
      this.total.text(total+' Albums ('+photos+' images)');
      return this;
    }
});

new App(); // create the main app
```

listing 15.1.28 Defining the main app's view and making use of the Collections and models defined earlier

jQuery().ajax() method already described since Backbone is designed to be used alongside jQuery. With Backbone's structured models and collections coupled with jQuery's visual flourishes and helper function, you have all the tools to build advanced client-side scripts.