Beginning JavaScript®, Fifth Edition
By Jeremy McPeak and Paul Wilton

14

Ajax

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Making HTTP requests with the `XMLHttpRequest` object
- Writing a custom Ajax module
- Working with older Ajax techniques to preserve usability

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at `http://www.wiley.com/go/BeginningJavaScript5E` on the Download Code tab. You can also view all of the examples and related files at `http://beginningjs.com`.

Since its inception, the Internet has used a transaction-like communication model; a browser sends a request to a server, which sends a response back to the browser, which (re)loads the page. This is typical HTTP communication, and it was designed to be this way. But this model is rather cumbersome for developers, because it requires web applications to consist of several pages. The resulting user experience becomes disjointed and interrupted due to these separate page loads.

In the early 2000s, a movement began to look for and develop new techniques to enhance the user's experience; to make web applications behave more like conventional applications. These new techniques offered the performance and usability usually associated with conventional desktop applications. It wasn't long before developers began to refine these processes to offer richer functionality to the user.

At the heart of this movement was one language: JavaScript, and its ability to make HTTP requests transparent to the user.

WHAT IS AJAX?

Essentially, *Ajax* allows client-side JavaScript to request and receive data from a server without refreshing the web page. This technique enables the developer to create an application that is uninterrupted, making only portions of the page reload with new data.

The term Ajax was originally coined by Jesse James Garrett in 2005. He wrote an article entitled "Ajax: A New Approach to Web Applications" (`www.adaptivepath.com/publications/essays/archives/000385.php`). In it, Garrett states that the interactivity gap between web and desktop applications is becoming smaller, and he cites applications such as Google Maps and Google Suggest as proof of this. The term originally stood for Asynchronous JavaScript + XML (XML was the format in which the browser and server communicated with each other). Today, Ajax simply refers to the pattern of using JavaScript to send and receive data from the web server without reloading the entire page.

Although the term Ajax was derived in 2005, the underlying methodology was used years before. Early Ajax techniques consisted of using hidden frames/iframes, dynamically adding `<script/>` elements to the document, and/or using JavaScript to send HTTP requests to the server; the latter has become quite popular in the past few years. These new techniques refresh only portions of a page, both cutting the size of data sent to the browser and making the web page feel more like a conventional application.

What Can It Do?

Ajax opened the doors for advanced web applications—ones that mimic desktop applications in form and in function. A variety of commercial websites employ the use of Ajax. These sites look and behave more like desktop applications than websites. The most notable Ajax-enabled web applications come from the search giant Google: Google Maps and Google Suggest.

Google Maps

Designed to compete with existing commercial mapping sites (and using images from its Google Earth), Google Maps (`http://maps.google.com`) uses Ajax to dynamically add map images to the web page. When you enter a location, the main page does not reload at all; the images are dynamically loaded in the map area. Google Maps also enables you to drag the map to a new location, and once again, the map images are dynamically added to the map area (see Figure 14-1).

Google Suggest

The now commonplace Google Suggest is another Google innovation that employs the use of Ajax. Upon first glance, it appears to be a normal Google search page. When you start typing, however, a drop-down box displays suggestions for search terms that might interest you. Under the suggested word or phrase is the number of results the search term returns (see Figure 14-2).

Browser Support

In the early years of Ajax, browser support was mixed. Every browser supported the basics in some way, but support differed from browser to browser. Today, Ajax is a just another normal part of JavaScript development, and today's modern browsers unquestionably support Ajax.

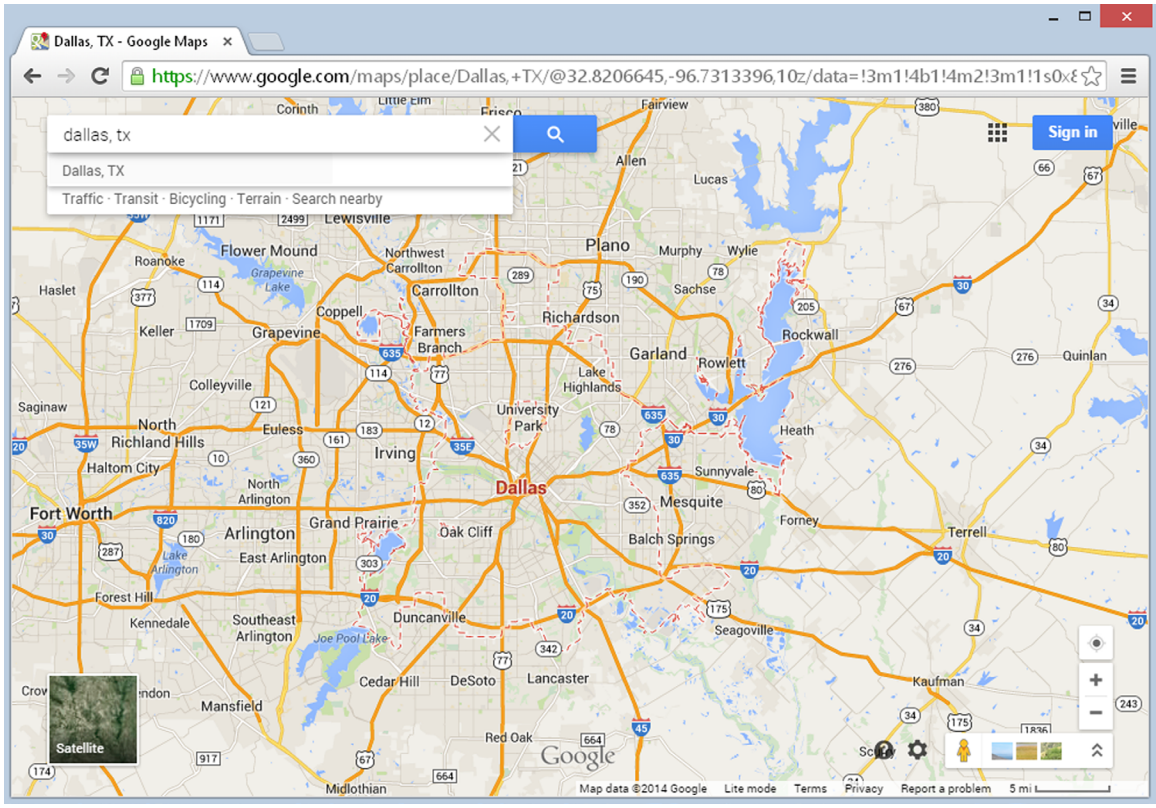

FIGURE 14-1

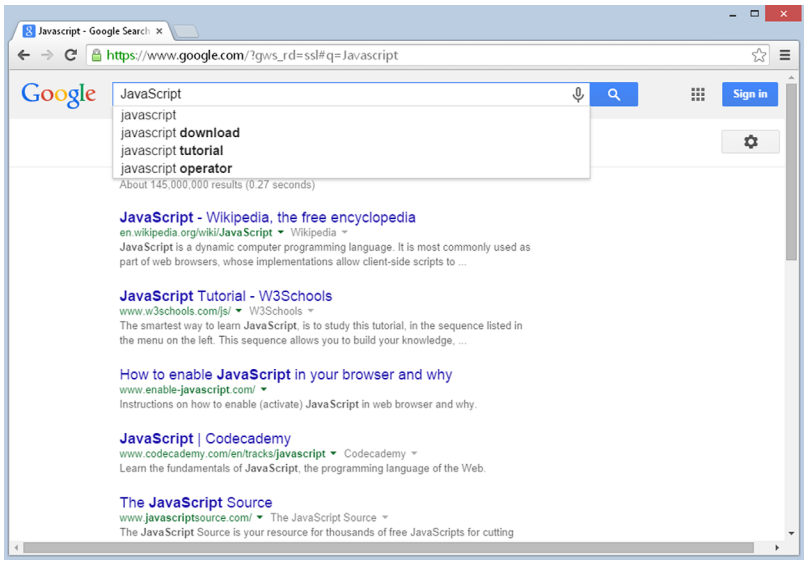

FIGURE 14-2

USING THE XMLHTTPREQUEST OBJECT

As stated before, you can create Ajax-enabled applications in a variety of ways. However, probably the most popular Ajax technique incorporates the JavaScript `XMLHttpRequest` object, which is present in all major browsers.

> ***NOTE*** *Despite its name, you can retrieve other types of data, like plaintext, with* `XMLHttpRequest`.

The `XMLHttpRequest` object originated as a Microsoft component, called `XmlHttp`, in the MSXML library first released with IE 5. It offered developers an easy way to open HTTP connections and retrieve XML data. Microsoft improved the component with each new version of MSXML, making it faster and more efficient.

As the popularity of the Microsoft `XMLHttpRequest` object grew, Mozilla decided to include its own version of the object with Firefox. The Mozilla version maintained the same properties and methods used in Microsoft's ActiveX component, making cross-browser usage possible. Soon after, Opera Software and Apple copied the Mozilla implementation, and Google naturally implemented it with Chrome's initial release. As for Internet Explorer, `XMLHttpRequest` is no longer an ActiveX component but a native object in the browser.

Creating an XMLHttpRequest Object

The `XMLHttpRequest` object is located in the `window` object. Creating an `XMLHttpRequest` object is as simple as calling its constructor:

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

This line creates an `XMLHttpRequest` object, which you can use to connect to, and request and receive data from, a server.

Using the XMLHttpRequest Object

Once you create the `XMLHttpRequest` object, you are ready to start requesting data with it. The first step in this process is to call the `open()` method to initialize the object:

```
request.open(requestType, url, async);
```

This method accepts three arguments. The first, `requestType`, is a string value consisting of the type of request to make. The value can be either `GET` or `POST`. The second argument is the URL to send the request to, and the third is an optional `true` or `false` value indicating whether the request should be made in asynchronous or synchronous mode.

Requests made in synchronous mode halt all JavaScript code from executing until a response is received from the server. This can slow down your application's execution time. In most cases, you want to use asynchronous mode, which lets the browser continue to execute your application's code

while the XMLHttpRequest object awaits a response from the server. Asynchronous mode is the default behavior of XMLHttpRequest, so you can usually omit the third argument to open().

> **NOTE** *In the past, it was considered best practice to pass* true *as the third argument.*

The next step is to send the request; do this with the send() method. This method accepts one argument, which is a string that contains the request body to send along with the request. GET requests do not contain any information, so pass null as the argument:

```
var request = new XMLHttpRequest();
request.open("GET", "http://localhost/myTextFile.txt", false);
request.send(null);
```

This code makes a GET request to retrieve a file called myTextFile.txt in synchronous mode. Calling the send() method sends the request to the server.

> **WARNING** *The* send() *method requires an argument to be passed, even if it is* null.

Each XMLHttpRequest object has a status property. This property contains the HTTP status code sent with the server's response. The server returns a status of 200 for a successful request, and one of 404 if it cannot find the requested file. With this in mind, consider the following example:

```
var request = new XMLHttpRequest();
request.open("GET", "http://localhost/myTextFile.txt", false);
request.send(null);

var status = request.status;

if (status == 200) {
    alert("The text file was found!");
} else if (status == 404) {
    alert("The text file could not be found!");
} else {
    alert("The server returned a status code of " + status);
}
```

This code checks the status property to determine what message to display to the user. If successful (a status of 200), an alert box tells the user the request file exists. If the file doesn't exist (status 404), the user sees a message stating that the server cannot find the file. Finally, an alert box tells the user the status code if it equals something other than 200 or 404.

Many different HTTP status codes exist, and checking for every code is not feasible. Most of the time, you should only be concerned with whether your request is successful. Therefore, you can cut the previous code down to this:

```
var request = new XMLHttpRequest();
request.open("GET", "http://localhost/myTextFile.txt", false);
```

```
request.send(null);

var status = request.status;

if (status == 200) {
    alert("The text file was found!");
} else {
    alert("The server returned a status code of " + status);
}
```

This code performs the same basic function, but it only checks for a status code of `200` and sends a generic message to alert the user for other status codes.

Asynchronous Requests

The previous code samples demonstrate the simplicity of synchronous requests. Asynchronous requests, on the other hand, add some complexity to your code because you have to handle the `readystatechange` event. In asynchronous requests, the `XMLHttpRequest` object exposes a `readyState` property, which holds a numeric value; each value refers to a specific state in a request's life span, as follows:

- `0`: The object has been created, but the `open()` method hasn't been called.
- `1`: The `open()` method has been called, but the request hasn't been sent.
- `2`: The request has been sent; headers and status are received and available.
- `3`: A response has been received from the server.
- `4`: The requested data has been fully received.

The `readystatechange` event fires every time the `readyState` property changes, calling the `onreadystatechange` event handler. The fourth and final state is the most important; it lets you know that the request completed.

> ***NOTE*** *It is important to note that even if the request was successful, you may not have the information you wanted. An error may have occurred on the server's end of the request (a* `404`, `500`, *or some other error). Therefore, you still need to check the status code of the request.*

Code to handle the `readystatechange` event could look like this:

```
var request = new XMLHttpRequest();

function reqReadyStateChange() {
    if (request.readyState == 4) {
        var status = request.status;

        if (status == 200) {
            alert(request.responseText);
```

```
            } else {
                alert("The server returned a status code of " + status);
            }
        }
    }

    request.open("GET", "http://localhost/myTextFile.txt");
    request.onreadystatechange = reqReadyStateChange;

    request.send(null);
```

This code first defines the `reqReadyStateChange()` function, which handles the `readystatechange` event. It first checks if the request completed by comparing `readyState` to `4`. The function then checks the request's status to ensure the server returned the requested data. Once these two criteria are met, the code alerts the value of the `responseText` property (the actual requested data in plaintext format). Note the `open()` method's call; the third argument is omitted. This makes the `XMLHttpRequest` object request data asynchronously.

The benefits of using asynchronous communication are well worth the added complexity of the `readystatechange` event, because the browser can continue to load the page and execute your other JavaScript code while the request object sends and receives data. Perhaps a user-defined module that wraps an `XMLHttpRequest` object could make asynchronous requests easier to use and manage.

> ***NOTE*** *An* `XMLHttpRequest` *object also has a property called* `responseXML`*, which attempts to load the received data into an HTML DOM (whereas* `responseText` *returns plaintext).*

CREATING A SIMPLE AJAX MODULE

The concept of code reuse is important in programming; it is the reason why functions are defined to perform specific, common, and repetitive tasks. Chapter 5 introduced you to the object-oriented construct of code reuse: reference types. These constructs contain properties that contain data and/or methods that perform actions with that data.

In this section, you write your own Ajax module called `HttpRequest`, thereby making asynchronous requests easier to make and manage. Before getting into writing this module, let's go over the properties and methods the `HttpRequest` reference type exposes.

Planning the HttpRequest Module

There's only one piece of information that you need to keep track of: the underlying `XMLHttpRequest` object. Therefore, this module will have only one property, `request`, which contains the underlying `XMLHttpRequest` object.

The `HttpRequest` exposes a single method called `send()`. Its purpose is to send the request to the server.

Now let's begin to write the module.

The HttpRequest Constructor

A reference type's constructor defines its properties and performs any logic needed to function properly:

```
function HttpRequest(url, callback) {
    this.request = new XMLHttpRequest();

    //more code here
}
```

The constructor accepts two arguments. The first, `url`, is the URL the `XMLHttpRequest` object will request. The second, `callback`, is a callback function; it will be called when the server's response is received (when the request's `readyState` is `4` and its `status` is `200`). The first line of the constructor initializes the `request` property, assigning an `XMLHttpRequest` object to it.

With the `request` property created and ready to use, you prepare to send the request:

```
function HttpRequest(url, callback) {
    this.request = new XMLHttpRequest();
    this.request.open("GET", url);

    function reqReadyStateChange() {
        //more code here
    }

    this.request.onreadystatechange = reqReadyStateChange;
}
```

The first line of the new code uses the `XMLHttpRequest` object's `open()` method to initialize the request object. Set the request type to `GET`, and use the `url` parameter to specify the URL you want to request. Because you omit `open()`'s third argument, you set the request object to use asynchronous mode.

The next few lines define the `reqReadyStateChange()` function. Defining a function within a function may seem weird, but it is perfectly legal to do so. This inner function cannot be accessed outside the containing function (the constructor in this case), but it has access to the variables and parameters of the containing constructor function. As its name implies, the `reqReadyStateChange()` function handles the request object's `readystatechange` event, and you bind it to do so by assigning it to the `onreadystatechange` event handler:

```
function HttpRequest(url, callback) {
    this.request = new XMLHttpRequest();
    this.request.open("GET", url);

    var tempRequest = this.request;

    function reqReadyStateChange() {
        if (tempRequest.readyState == 4) {
            if (tempRequest.status == 200) {
                callback(tempRequest.responseText);
            } else {
                alert("An error occurred trying to contact the server.");
            }
```

```
        }
    }

    this.request.onreadystatechange = reqReadyStateChange;
}
```

The new lines of code may once again look a little strange, but it's actually a pattern you'll often see when looking at other people's code. The first new line creates the `tempRequest` variable. This variable is a pointer to the current object's `request` property, and it's used within the `reqReadyStateChange()` function. This is a technique to get around scoping issues. Ideally, you would use `this.request` inside the `reqReadyStateChange()` function. However, the `this` keyword points to the `reqReadyStateChange()` function instead of to the `XMLHttpRequest` object, which would cause the code to not function properly. So when you see `tempRequest`, think `this.request`.

Inside the `reqReadyStateChange()` function, you see the following line:

```
callback(tempRequest.responseText);
```

This line calls the callback function specified by the constructor's `callback` parameter, and you pass the `responseText` property to this function. This allows the callback function to use the information received from the server.

Creating the send() Method

There is one method in this reference type, and it enables you to send the request to the server. Sending a request to the server involves the `XMLHttpRequest` object's `send()` method. This `send()` is similar, with the difference being that it doesn't accept arguments:

```
HttpRequest.prototype.send = function () {
    this.request.send(null);
};
```

This version of `send()` is simple in that all you do is call the `XMLHttpRequest` object's `send()` method and pass it `null`.

The Full Code

Now that the code's been covered, open your text editor and type the following:

```
function HttpRequest(url, callback) {
    this.request = new XMLHttpRequest();
    this.request.open("GET", url);

    var tempRequest = this.request;

    function reqReadyStateChange() {
        if (tempRequest.readyState == 4) {
            if (tempRequest.status == 200) {
                callback(tempRequest.responseText);
            } else {
                alert("An error occurred trying to contact the server.");
            }
```

```
            }
        }

        this.request.onreadystatechange = reqReadyStateChange;
    }

    HttpRequest.prototype.send = function () {
        this.request.send(null);
    };
```

Save this file as `httprequest.js`. You'll use it later in the chapter.

The goal of this module was to make asynchronous requests easier to use, so let's look at a brief code-only example and see if that goal was accomplished.

The first thing you need is a function to handle the data received from the request; this function gets passed to the `HttpRequest` constructor:

```
function handleData(text) {
    alert(text);
}
```

This code defines a function called `handleData()` that accepts one argument called `text`. When executed, the function merely alerts the data passed to it. Now create an `HttpRequest` object and send the request:

```
var request = new HttpRequest(
        "http://localhost/myTextFile.txt", handleData);

request.send();
```

Pass the text file's location and a pointer of the `handleData()` function to the constructor, and send the request with the `send()` method. The `handleData()` function is called in the event of a successful request.

This module encapsulates the code related to asynchronous `XMLHttpRequest` requests nicely. You don't have to worry about creating the request object, handling the `readystatechange` event, or checking the request's `status`; the `HttpRequest` module does it all for you.

VALIDATING FORM FIELDS WITH AJAX

You've probably seen it many times: registering as a new user on a website's forum or signing up for web-based e-mail, only to find that your desired username is taken. Of course, you don't find this out until after you've filled out the entire form, submitted it, and watched the page reload with new data (not to mention that you've lost some of the data you entered). As you can attest, form validation can be a frustrating experience. Thankfully, Ajax can soften this experience by sending data to the server before submitting the form—allowing the server to validate the data, and letting the user know the outcome of the validation without reloading the page!

In this section, you create a form that uses Ajax techniques to validate form fields. It's possible to approach building such a form in a variety of ways; the easiest of which to implement provides a link that initiates an HTTP request to the server application to check whether the user's desired information is available to use.

The form you build resembles typical forms used today; it will contain the following fields:

- **Username (validated):** The field where the user types his or her desired username
- **Email (validated):** The field where the user types his or her e-mail
- **Password (not validated):** The field where the user types his or her password
- **Verify Password (not validated):** The field where the user verifies his or her password

Note that the Password and Verify Password fields are just for show in this example. Verifying a password is certainly something the server application can do; however, it is far more efficient to let JavaScript perform that verification. Doing so adds more complexity to this example, and we want to keep this as simple as possible to help you get a grasp of using Ajax.

Next to the Username and Email fields will be a hyperlink that calls a JavaScript function to query the server with your `HttpRequest` module from the previous section.

As mentioned earlier, Ajax is communication between the browser and server. So this example needs a simple server application to validate the form fields. PHP programming is beyond the scope of this book. However, we should discuss how to request data from the PHP application, as well as look at the response the application sends back to JavaScript.

Requesting Information

The PHP application looks for one of two arguments in the query string: `username` and `email`.

To check the availability of a username, use the `username` argument. The URL to do this looks like the following:

```
http://localhost/formvalidator.php?username=[usernameToSearchFor]
```

When searching for a username, replace `[usernameToSearchFor]` with the actual name.

Searching for an e-mail follows the same pattern. The e-mail URL looks like this, where you replace `[emailToSearchFor]` with the actual name:

```
http://localhost/formvalidator.php?email=[emailToSearchFor]
```

The Received Data

A successful request results in a simple JSON structure that defines two members called `searchTerm` and `available`, like this:

```
{
    "searchTerm": "jmcpeak",
    "available" : true
}
```

As its name implies, the `searchTerm` item contains the string used in the username or e-mail search. The `available` item is a boolean value. If `true`, the requested username and/or e-mail is available for use. If `false`, the username and/or e-mail is in use and therefore not available.

Before You Begin

This is a live-code Ajax example; therefore, your system must meet a few requirements if you want to run this example from your computer.

A Web Server

First, you need a web server. If you are using Windows, you have Microsoft's web server software, Internet Information Services (IIS), freely available to you. To install it on Windows, open Programs and Features in the Control Panel and click Turn Windows features on or off. Figure 14-3 shows the Windows Features dialog box in Windows 8.

Expand Internet Information Services and check the features you want to install. You must check World Wide Web Services (Figure 14-4). You may need your operating system's installation CD to complete the installation.

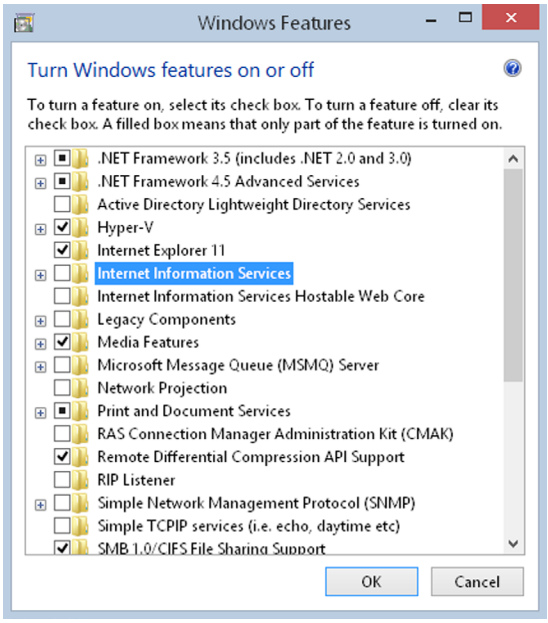

FIGURE 14-3

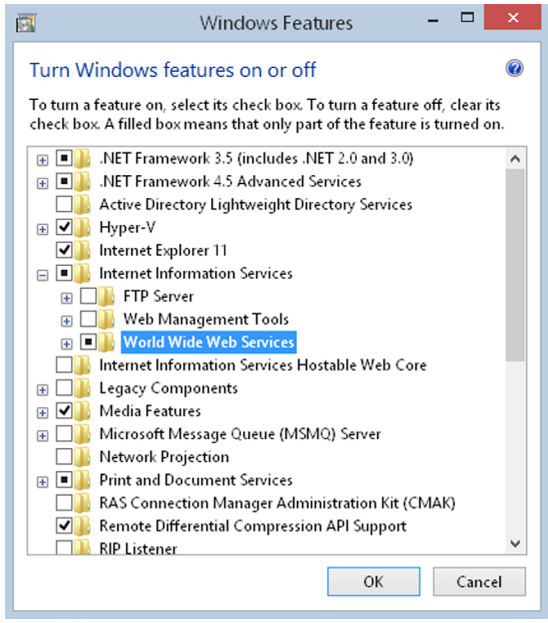

FIGURE 14-4

If you use another operating system, or you want to use another web server application, you can install Apache HTTP Server (`www.apache.org`). This is an open source web server and can run on a variety of operating systems, such as Linux, Unix, and Windows, to list only a few. Most websites run on Apache, so don't feel nervous about installing it on your computer. It is extremely stable.

If you do choose to use Apache, don't download and install it just yet; there are different versions of Apache. Instead, download PHP first because PHP's website gives you accurate information on which Apache version you should download and install.

PHP

PHP is a popular open source server-side scripting language and must be installed on your computer if you want to run PHP scripts. You can download PHP in a variety of forms (binaries, Windows installation wizards, and source code) at `www.php.net`. The PHP code used in this example was written in PHP 5.

TRY IT OUT XMLHttpRequest Smart Form

In this Try It Out, you will use Ajax to validate form fields. Open your text editor and type the following:

```
<!DOCTYPE html>

<html lang="en">
<head>
    <title>Chapter 14: Example 1</title>
    <style>
        .fieldname {
            text-align: right;
        }

        .submit {
            text-align: right;
        }
    </style>
</head>
<body>
    <form>
        <table>
            <tr>
                <td class="fieldname">
                    Username:
                </td>
                <td>
                    <input type="text" id="username" />
                </td>
                <td>
                    <a id="usernameAvailability" href="#">Check Availability</a>
                </td>
            </tr>
            <tr>
                <td class="fieldname">
                    Email:
                </td>
                <td>
                    <input type="text" id="email" />
                </td>
                <td>
                    <a id="emailAvailability" href="#">Check Availability</a>
                </td>
            </tr>
            <tr>
                <td class="fieldname">
```

```
                Password:
            </td>
            <td>
                <input type="text" id="password" />
            </td>
            <td />
        </tr>
        <tr>
            <td class="fieldname">
                Verify Password:
            </td>
            <td>
                <input type="text" id="password2" />
            </td>
            <td />
        </tr>
        <tr>
            <td colspan="2" class="submit">
                <input type="submit" value="Submit" />
            </td>
            <td />
        </tr>
    </table>
</form>
<script src="httprequest.js"></script>
<script>
    function checkUsername(e) {
        e.preventDefault();

        var userValue = document.getElementById("username").value;

        if (!userValue) {
            alert("Please enter a user name to check!");
            return;
        }

        var url = "ch14_formvalidator.php?username=" + userValue;

        var request = new HttpRequest(url, handleResponse);
        request.send();
    }

    function checkEmail(e) {
        e.preventDefault();

        var emailValue = document.getElementById("email").value;

        if (!emailValue) {
            alert("Please enter an email address to check!");
            return;
        }

        var url = "ch14_formvalidator.php?email=" + emailValue;

        var request = new HttpRequest(url, handleResponse);
```

```
            request.send();
        }

        function handleResponse(responseText) {
            var response = JSON.parse(responseText);

            if (response.available) {
                alert(response.searchTerm + " is available!");
            } else {
                alert("We're sorry, but " + response.searchTerm +
                      " is not available.");
            }
        }

        document.getElementById("usernameAvailability")
                .addEventListener("click", checkUsername);

        document.getElementById("emailAvailability")
                .addEventListener("click", checkEmail);
    </script>
</body>

</html>
```

Save this file in your web server's root directory. If you're using IIS for your web server, save it as `c:\inetpub\wwwroot\ch14_example1.html`. If you're using Apache, you'll want to save it inside the `htdocs` folder: `path_to_htdocs\htdocs\ch14_example1.html`.

You also need to place `httprequest.js` (the `HttpRequest` module) and the `ch14_formvalidator.php` file (from the code download) into the same directory as `ch14_example1.html`.

Now open your browser and navigate to `http://localhost/ch14_formvalidator.php`. If everything is working properly, you should see the text "PHP is working correctly. Congratulations!" as in Figure 14-5.

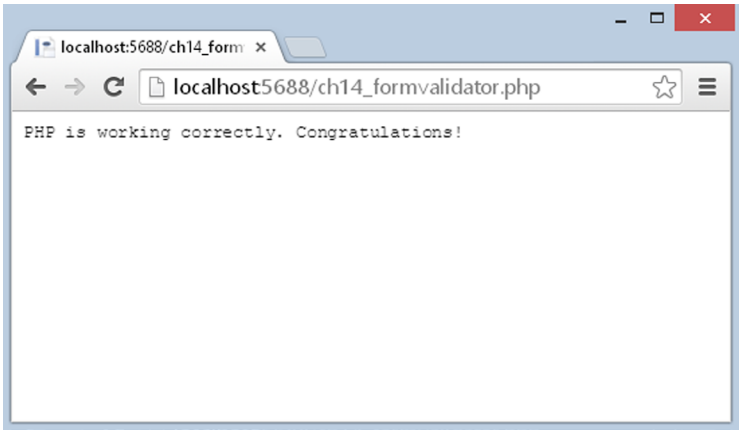

FIGURE 14-5

Now point your browser to `http://localhost/ ch14_example1.html`, and you should see something like Figure 14-6.

Type **jmcpeak** into the `Username` field and click the Check Availability link next to it. You'll see an `alert` box like the one shown in Figure 14-7.

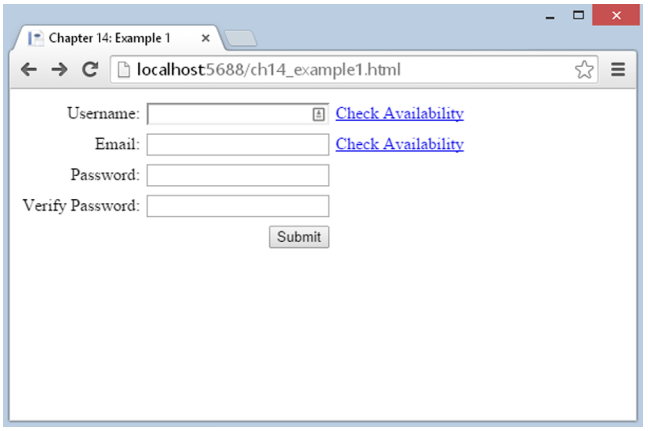

FIGURE 14-6

FIGURE 14-7

Now type **someone@xyz.com** in the e-mail field and click the Check Availability link next to it. Again, you'll be greeted with an `alert` box stating that the e-mail's already in use. Now input your own username and e-mail into these fields and click the appropriate links. Chances are an `alert` box will tell you that your username and/or e-mail is available (the usernames `jmcpeak` and `pwilton` and the e-mails `someone@xyz.com` and `someone@zyx.com` are the only ones used by the application).

The body of this HTML page contains a simple form whose fields are contained within a table. Each form field exists in its own row. The first two rows contain the fields you're most interested in: the Username and Email fields:

```
<form>
    <table>
        <tr>
            <td class="fieldname">
                Username:
            </td>
            <td>
                <input type="text" id="username" />
            </td>
            <td>
                <a id="usernameAvailability" href="#">Check Availability</a>
            </td>
        </tr>
        <tr>
            <td class="fieldname">
                Email:
            </td>
            <td>
```

```
                <input type="text" id="email" />
            </td>
            <td>
                <a id="emailAvailability" href="#">Check Availability</a>
            </td>
        </tr>
        <!-- HTML to be continued later -->
```

The first column contains text identifiers for the fields. The second column contains the `<input/>` elements themselves. Each of these tags has an `id` attribute: `username` for the Username field and `email` for the Email field. This enables you to easily find the `<input/>` elements and get the text entered into them.

The third column contains an `<a/>` element. These hyperlinks exist for the sole purpose of kicking off Ajax requests. As such, they have a hash (`#`) in their `href` attributes, thus preventing the browser from navigating to a different page (to be considered a valid, clickable hyperlink, an `<a/>` element must have an `href` value). Each of these links has an `id` attribute that you'll use later in your JavaScript code.

The remaining three rows in the table contain two password fields and the Submit button (the smart form currently does not use these fields):

```
        <!-- HTML continued from earlier -->
        <tr>
            <td class="fieldname">
                Password:
            </td>
            <td>
                <input type="text" id="password" />
            </td>
            <td />
        </tr>
        <tr>
            <td class="fieldname">
                Verify Password:
            </td>
            <td>
                <input type="text" id="password2" />
            </td>
            <td />
        </tr>
        <tr>
            <td colspan="2" class="submit">
                <input type="submit" value="Submit" />
            </td>
            <td />
        </tr>
    </table>
</form>
```

The CSS in this HTML page consists of only a couple of CSS rules:

```
.fieldname {
    text-align: right;
```

```
}

.submit {
    text-align: right;
}
```

These rules align the fields to give the form a clean and unified look.

As stated earlier, the hyperlinks are key to the Ajax functionality, because they call JavaScript functions when clicked. The first function, `checkUsername()`, retrieves the text the user entered into the `Username` field and issues an HTTP request to the server.

This function executes because the user clicked a link. Therefore, you want to prevent the browser from navigating to the URL specified in its `href` attribute. Even though the URL is the hash (`#`), you still want to call `preventDefault()`:

```
function checkUsername(e) {
    e.preventDefault();

    var userValue = document.getElementById("username").value;
```

Use the `document.getElementById()` method to find the `<input id="FileName_username"/>` element and use its `value` property to retrieve the text typed into the text box. You then check to see if the user typed any text:

```
    if (!userValue) {
        alert("Please enter a user name to check!");
        return;
    }
```

If the text box is empty, the function alerts the user to input a username and stops the function from further processing. The application would make unnecessary requests to the server if the code didn't do this.

Next construct the URL to make the request to the PHP application and assign it to the `url` variable. Then create an `HttpRequest` object by passing the URL and the `handleResponse()` callback function to the constructor, and send the request by calling `send()`:

```
    var url = "ch14_formvalidator.php?username=" + userValue;

    var request = new HttpRequest(url, handleResponse);
    request.send();
}
```

You look at the `handleResponse()` function later. For now, let's examine the `checkEmail()` function.

Checking the e-mail address availability is almost identical to the username process. The `checkEmail()` function retrieves the text typed in the Email field and sends that information to the server application:

```
function checkEmail(e) {
    e.preventDefault();

    var emailValue = document.getElementById("email").value;

    if (!emailValue) {
```

```
            alert("Please enter an email address to check!");
            return;
        }

        var url = "ch14_formvalidator.php?email=" + emailValue;

        var request = new HttpRequest(url, handleResponse);
        request.send();
    }
```

This function also uses `handleResponse()` to handle the server's response. The `handleResponse()` function executes when the `HttpRequest` object receives a complete response from the server. This function uses the requested information to tell the user whether the username or e-mail address is available. Remember, the response from the server is JSON-formatted data. So, you need to first parse the data into a JavaScript object:

```
function handleResponse(responseText) {
    var response = JSON.parse(responseText);
```

The server's response is parsed into an object that is stored in the `response` variable. You then use this object's `available` property to display the appropriate message to the user:

```
    if (response.available) {
        alert(response.searchTerm + " is available!");
    } else {
        alert("We're sorry, but " + response.searchTerm + " is not available.");
    }
}
```

If `available` is `true`, the function tells the user that his desired username or e-mail address is okay to use. If not, the `alert` box says that the user's desired username or e-mail address is taken.

Finally, you need to set up the event listeners for your two links:

```
document.getElementById("usernameAvailability")
        .addEventListener("click", checkUsername);

document.getElementById("emailAvailability")
        .addEventListener("click", checkEmail);
```

You do this by simply retrieving the `<a/>` elements by their respective `id` values and listening for the `click` event.

THINGS TO WATCH OUT FOR

Using JavaScript to communicate between server and client adds tremendous power to the language's abilities. However, this power does not come without its share of caveats. The two most important issues are security and usability.

Security Issues

Security is a hot topic in today's Internet, and as a web developer you must consider the security restrictions placed on Ajax. Knowing the security issues surrounding Ajax can save you development and debugging time.

The Same-Origin Policy

Since the early days of Netscape Navigator 2.0, JavaScript cannot access scripts or documents from a different origin. This is a security measure that browser makers adhere to; otherwise, malicious coders could execute code wherever they wanted. The same-origin policy dictates that two pages are of the same origin only if the protocol (HTTP), port (the default is 80), and host are the same.

Consider the following two pages:

- Page 1 is located at `http://www.site.com/folder/mypage1.htm`.
- Page 2 is located at `http://www.site.com/folder10/mypage2.htm`.

According to the same-origin policy, these two pages are of the same origin. They share the same host (`www.site.com`), use the same protocol (HTTP), and are accessed on the same port (none is specified; therefore, they both use 80). Because they are of the same origin, JavaScript on one page can access the other page.

Now consider the next two pages:

- Page 1 is located at `http://www.site.com/folder/mypage1.htm`.
- Page 2 is located at `https://www.site.com/folder/mypage2.htm`.

These two pages are not of the same origin. The host is the same, but their protocols and ports are different. Page 1 uses `HTTP` (port 80), whereas Page 2 uses `HTTPS` (port 443). This difference, though slight, is enough to give the two pages two separate origins. Therefore, JavaScript on one of these pages cannot access the other page.

So what does this have to do with Ajax? Everything, because a large part of Ajax is JavaScript. For example, because of this policy, an `XMLHttpRequest` object cannot retrieve any file or document from a different origin by default. There is, however, a legitimate need for cross-origin requests, and the W3C responded with the Cross-Origin Resource Sharing (CORS) specification.

CORS

The CORS specification defines how browsers and servers communicate with one another when sending requests across origins. For CORS to work, the browser must send a custom HTTP header called `Origin` that contains the protocol, domain name, and port of the page making the request. For example, if the JavaScript on the page `http://www.abc.com/xyz.html` used `XMLHttpRequest` to issue a request to `http://beginningjs.com`, the `Origin` header would look like this:

```
Origin: http://www.abc.com
```

When the server responds to a CORS request, it must also send a custom header called `Access-Control-Allow-Origin`, and it must contain the same origin specified in the request's `Origin` header. Continuing from the previous example, the server's response must contain the following `Access-Control-Allow-Origin` header for CORS to work:

```
Access-Control-Allow-Origin: http://www.abc.com
```

If this header is missing, or if the origins don't match, the browser doesn't process the request.

Alternatively, the server can include the `Access-Control-Allow-Origin` header with a value of `*`, signifying that all origins are accepted. This is primarily used by publicly available web services.

> ***NOTE*** *These custom headers are automatically handled by the browser. You do not need to set your own* `Origin` *header, and you do not have to manually check the* `Access-Control-Allow-Origin`*.*

Usability Concerns

Ajax breaks the mold of traditional web applications and pages. It enables developers to build applications that behave in a more conventional, non-"webbish" way. This, however, is also a drawback, because the Internet has been around for many, many years, and users are accustomed to traditional web pages.

Therefore, it is up to developers to ensure that users can use their web pages, and use them as they expect to, without causing frustration.

The Browser's Back Button

One of the advantages of `XMLHttpRequest` is its ease of use. You simply create the object, send the request, and await the server's response. Unfortunately, this object does have a downside: Most browsers do not log a history of requests made with the object. Therefore, `XMLHttpRequest` essentially breaks the browser's Back button. This might be a desired side-effect for some Ajax-enabled applications or components, but it can cause serious usability problems for the user.

Creating a Back/Forward-Capable Form with an IFrame

It's possible to avoid breaking the browser's navigational buttons by using an older but reliable Ajax technique: using hidden frames/iframes to facilitate client-server communication. You must use two frames for this method to work properly. One must be hidden, and one must be visible.

> ***NOTE*** *Note that when you are using an iframe, the document that contains the iframe is the visible frame.*

The hidden-frame technique consists of a four-step process:

1. The user initiates a JavaScript call to the hidden frame by clicking a link in the visible frame or performing some other type of user interaction. This call is usually nothing more

complicated that redirecting the hidden frame to a different web page. This redirection automatically triggers the second step.

2. The request is sent to the server, which processes the data.
3. The server sends its response (a web page) back to the hidden frame.
4. The browser loads the web page in the hidden frame and executes any JavaScript code to contact the visible frame.

The example in this section is based on the form validator built earlier in the chapter, but you'll use a hidden iframe to facilitate the communication between the browser and the server instead of an `XMLHttpRequest` object. Before getting into the code, you should first know about the data received from the server.

The Server Response

You expected a JSON data structure as the server's response when using `XMLHttpRequest` to get data from the server. The response in this example is different and must consist of two things:

- The data, which must be in HTML format
- A mechanism to contact the parent document when the iframe receives the HTML response

The following code is an example of the response HTML page:

```
<!DOCTYPE html>

<html lang="en">
<head>
    <title>Returned Data</title>
</head>
<body>
    <script>
        //more code here
    </script>
</body>
</html>
```

This simple HTML page contains a single `<script/>` element in the body of the document. The JavaScript code contained in this script block is generated by the PHP application, calling `handleResponse()` in the visible frame and passing it the expected JSON.

The JSON data structure has a new member: the `value` field. It contains the username or e-mail that was sent in the request. Therefore, the following HTML document is a valid response from the PHP application:

```
<!DOCTYPE html>

<html lang="en">
<head>
    <title>Returned Data</title>
</head>
<body>
```

```
        <script>
            top.handleResponse('{"available":false, "value":"jmcpeak"}');
        </script>
    </body>
    </html>
```

The HTML page calls the `handleResponse()` function in the parent window and passes the JSON structure signifying that the username or e-mail address is available. With the response in this format, you can keep a good portion of the JavaScript code identical to Example 1.

TRY IT OUT Iframe Smart Form

The code for this revised smart form is very similar to the code used previously with the `XMLHttpRequest` example. There are, however, a few changes. Open up your text editor and type the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Chapter 14: Example 2</title>
    <style>
        .fieldname {
            text-align: right;
        }

        .submit {
            text-align: right;
        }

        #hiddenFrame {
            display: none;
        }
    </style>
</head>
<body>
    <form>
        <table>
            <tr>
                <td class="fieldname">
                    Username:
                </td>
                <td>
                    <input type="text" id="username" />
                </td>
                <td>
                    <a id="usernameAvailability" href="#">Check Availability</a>
                </td>
            </tr>
            <tr>
                <td class="fieldname">
                    Email:
                </td>
                <td>
                    <input type="text" id="email" />
```

```
            </td>
            <td>
                <a id="emailAvailability" href="#">Check Availability</a>
            </td>
        </tr>
        <tr>
            <td class="fieldname">
                Password:
            </td>
            <td>
                <input type="text" id="password" />
            </td>
            <td></td>
        </tr>
        <tr>
            <td class="fieldname">
                Verify Password:
            </td>
            <td>
                <input type="text" id="password2" />
            </td>
            <td></td>
        </tr>
        <tr>
            <td colspan="2" class="submit">
                <input type="submit" value="Submit" />
            </td>
            <td></td>
        </tr>
    </table>
</form>
<iframe src="about:blank" id="hiddenFrame" name="hiddenFrame"></iframe>
<script>
    function checkUsername(e) {
        e.preventDefault();

        var userValue = document.getElementById("username").value;

        if (!userValue) {
            alert("Please enter a user name to check!");
            return;
        }

        var url = "ch14_iframevalidator.php?username=" + userValue;

        frames["hiddenFrame"].location = url;
    }

    function checkEmail(e) {
        e.preventDefault();

        var emailValue = document.getElementById("email").value;

        if (!emailValue) {
            alert("Please enter an email address to check!");
```

```
                return;
            }

            var url = "ch14_iframevalidator.php?email=" + emailValue;

            frames["hiddenFrame"].location = url;
        }

        function handleResponse(responseText) {
            var response = JSON.parse(responseText);

            if (response.available) {
                alert(response.searchTerm + " is available!");
            } else {
                alert("We're sorry, but " + response.searchTerm +
                      " is not available.");
            }
        }

        document.getElementById("usernameAvailability")
                .addEventListener("click", checkUsername);

        document.getElementById("emailAvailability")
                .addEventListener("click", checkEmail);
    </script>
</body>
</html>
```

Save this file as `ch14_example2.html`, and save it in your web server's root directory. Also locate the `ch14_iframevalidator.php` file from the code download and place it in the same directory.

Open your web browser and navigate to `http://localhost/ch14_example2.html`. You should see a page similar to Example 1.

Check for three usernames and e-mail addresses. After you clear the final `alert` box, click the browser's Back button a few times. You'll notice that it is cycling through the information you previously entered. The text in the text box will not change; however, the `alert` box will display the names and e-mails you entered. You can do the same thing with the Forward button.

The HTML in the body of the page remains unchanged except for the addition of the `<iframe/>` tag after the closing `<form/>` tag:

```
<iframe src="about:blank" id="hiddenFrame" name="hiddenFrame" />
```

This frame is initialized to have a blank HTML page loaded. Its `name` and `id` attributes contain the value of `hiddenFrame`. You use the value of the `name` attribute later to retrieve this frame from the `frames` collection in the BOM. Next, you set the CSS for the frame:

```
#hiddenFrame {
    display: none;
}
```

This rule contains one style declaration to hide the iframe from view.

> ***NOTE*** *Hiding an iframe through CSS enables you to easily show it if you need to debug the server-side application.*

Next up, the JavaScript:

```
function checkUsername(e) {
    e.preventDefault();

    var userValue = document.getElementById("username").value;

    if (!userValue) {
        alert("Please enter a user name to check!");
        return;
    }

    var url = "ch14_iframevalidator.php?username=" + userValue;

    frames["hiddenFrame"].location = url;
}
```

This `checkUsername()` function is almost identical to Example 1. The value of the `url` variable is changed to the new `ch14_iframvalidator.php` file. The actual request is made by accessing the `<iframe/>` element using the `frames` collection and setting its `location` property to the new URL.

The `checkEmail()` function has the same modifications:

```
function checkEmail(e) {
    e.preventDefault();

    var emailValue = document.getElementById("email").value;

    if (!emailValue) {
        alert("Please enter an email address to check!");
        return;
    }

    var url = "ch14_iframevalidator.php?email=" + emailValue;

    frames["hiddenFrame"].location = url;
}
```

As before, the `checkEmail()` function retrieves the text box's value and checks to see if the user entered data. It then constructs the URL using `ch14_iframevalidator.php` and loads the URL into the `<iframe/>`.

Dealing with Delays

The web browser is just like any other conventional application in that user interface (UI) cues tell the user that something is going on. For example, when a user clicks a link, the throbber animation may run or the cursor might change to display a "busy" animation.

This is another area in which Ajax solutions, and `XMLHttpRequest` specifically, miss the mark. However, this problem is simple to overcome: Simply add UI elements to tell the user something is going on and remove them when the action is completed. Consider the following code:

```
function requestComplete(responseText) {

    //do something with the data here

    document.getElementById("divLoading").style.display = "none";
}

var myRequest = new HttpRequest("http://localhost/myfile.txt",
                                requestComplete);

//show that we're loading
document.getElementById("divLoading").style.display = "block";

myRequest.send();
```

This code uses the `HttpRequest` module to request a text file. Before sending the request, it retrieves an HTML element in the document with an `id` of `divLoading`. This `<div/>` element tells the user that data is loading. The code then hides the element when the request completes, thus letting the user know that the process completed.

Offering this information to your users lets them know the application is doing something. Without such visual cues, users are left to wonder if the application is working on whatever they requested.

Degrade Gracefully When Ajax Fails

In a perfect world, the code you write would work every time it runs. Unfortunately, you have to face the fact that many times Ajax-enabled web pages will not use the Ajax-enabled goodness because a user turned off JavaScript in his browser.

The only real answer to this problem is to build an old-fashioned web page with old-fashioned forms, links, and other HTML elements. Then, using JavaScript, you can disable the default behavior of those HTML elements and add Ajax functionality. Consider this hyperlink as an example:

```
<a href="http://www.wrox.com" title="Wrox Publishing">Wrox Publishing</a>
```

This is a normal, run-of-the-mill hyperlink. When the user clicks it, the browser will take him to `http://www.wrox.com`. By using JavaScript, you of course can prevent this behavior by using the `Event` object's `preventDefault()` method. Simply register a `click` event handler for the `<a/>` element and call `preventDefault()`. Both Examples 1 and 2 demonstrated this technique.

As a rule of thumb, build your web page first and add Ajax later.

SUMMARY

This chapter introduced you to Ajax, and it barely scratched the surface of Ajax and its many uses:

- You looked at the `XMLHttpRequest` object, and learned how to make both synchronous and asynchronous requests to the server and how to use the `onreadystatechange` event handler.
- You built your own Ajax module to make asynchronous HTTP requests easier for you to code.
- You used your new Ajax module in a smarter form, one that checks usernames and e-mails to see if they are already in use.
- You saw how `XMLHttpRequest` breaks the browser's Back and Forward buttons, and addressed this problem by rebuilding the same form using a hidden iframe to make requests.
- You looked at some of the downsides to Ajax, including the security issues and the gotchas.

EXERCISES

You can find suggested solutions for these questions in Appendix A.

1. Extend the `HttpRequest` module to include synchronous requests in addition to the asynchronous requests the module already makes. You'll have to make some adjustments to your code to incorporate this functionality. (Hint: Create an `async` property for the module.)

2. It was mentioned earlier in the chapter that you could modify the smart form to not use hyperlinks. Change the form that uses the `HttpRequest` module so that the Username and Email fields are checked when the user submits the form. Listen for the form's `submit` event and cancel the submission if a username or e-mail is taken.