CHAPTER 7

Creating Responsive GUIs with Real-Time Validation

AJAX THE CLEANSER

You knew it was coming sooner or later, so why wait any longer? Perhaps the most known usage of the term Ajax is the all-purpose cleanser that was introduced by Colgate-Palmolive in 1947. The cleanser was named after the powerful Greek hero Ajax, which led to the Ajax product slogan: "Stronger than dirt!"

One of the truly killer applications of Ajax is real-time user input validation, which means that data entered by the user is checked for validity as the user enters it. This can dramatically improve the efficiency of user interfaces because errors are caught as they are made, thus eliminating the need to submit an entire form to find out whether a problem exists. Additionally, applications just feel smarter when they give immediate feedback to the user regarding data entry. As it turns out, Ajax isn't technically required for a lot of real-time validation tricks, but it still does play a role in some instances when it is necessary to load data from a server as part of the validation.

This chapter digs into real-time validation and explores when and where it makes sense to inject such functionality into your own applications. You also take a quick break from Ajax to learn how to validate popular data types such as phone numbers, dates, and email addresses. Ajax then comes back into play as you work through a practical example involving ZIP code validation that performs a live look-up of the respective city and state. This is an extremely handy Ajax trick you can use to significantly streamline the entry of location/address information.

The following files are used by the Validator application in this chapter and are located on the live Linux CD-ROM with the book in the `chap07` example code folder:

- `validator.html`—Main Web page
- `ziplookup.php`—Server script for looking up a city/state based upon a ZIP code
- `ajaxkit.js`—Core Ajax functions for the Ajax Toolkit
- `domkit.js`—DOM-related functions for the Ajax Toolkit
- `validatekit.js`—Validation functions for the Ajax Toolkit
- `wait.gif`—Animated "loading" icon that is displayed during an Ajax request

The Challenge: Checking User Input in Real Time

The challenge laid down in this chapter is quite broad—improve the retrieval of information from the user by eliminating input errors immediately at the point where they are made. If you use a word processor such as Microsoft Word, which has a real-time spell checker, then you understand how handy this feature can be. In many ways the auto-completion application in the previous chapter can be thought of as roughly similar to a real-time validator. However, auto-complete is more about improving efficiency than it is about correcting mistakes, even though the two tasks often go hand in hand.

To see a practical validator in action, check out Google's Gmail online email service at `http://mail.google.com/`. The invitation feature in Gmail allows you to send an invitation to a friend to join Gmail. The invitation request requires the email address of your friend, which is what you're supposed to enter in the text box. An improperly formatted email address results in Gmail displaying an "invalid address" message just below the email address text box.

Gmail isn't quite as aggressive with its validation as some Ajax applications, which is evident by the fact that you have to click a Send Invite button before Gmail performs the validation. A more aggressive approach would involve validating the address as soon as the user leaves the email text box. This latter approach is often employed in Ajax applications, and in most cases serves as an improvement because it can head off invalid data entry as soon as it is entered.

The general idea behind a modern approach to user input validation is to eliminate the need for a page reload in order to see if data is valid. In many cases you can eliminate the entire trip to the server and perform the validation entirely within

the client. Strictly speaking, if the server isn't involved, the validation isn't truly using Ajax. However, there is a notion of an Ajax "feel" to an application that can still be gained via client-side validation. The challenge in this chapter involves both client-only and true Ajax validation.

Speaking of the challenge, this chapter gets away from accomplishing a central task and instead focuses on demonstrating how to validate several different kinds of user input. Sure, it would've been possible to dream up some example involving numbers, dates, email addresses, phone numbers, and ZIP codes, but the additional overhead would just distract from the real emphasis, validating data. So the challenge in the Validator application is to simply present several user-input text boxes, each associated with a particular data type, and validate each one in real time as the user moves through the user interface.

Following are the data types targeted for validation by the Validator application:

- Integer
- Number
- Phone number
- Email address
- Date
- ZIP code

The key to validation isn't just popping up an annoying alert box to let the user know his or her input is bad. In fact, an alert box is the worst way you can go about notifying the user of an input problem. A better approach is to sneakily provide help fields on the form that are invisible unless a validation problem occurs, in which case you display help text for the user. For example, you might display the help text

```
"Please enter a date (for example, 01/14/1975)."
```

when the user enters an invalid date. The help text appears when the data is bad and then magically goes away once the data is fixed.

Of course, the "magic" is in the validation code that supports the help text feature. This code is part of the `validatekit.js` file included as part of the Ajax Toolkit on the CD-ROM with this book, and is also a part of the source code files for the Validator sample application. As was originally mentioned back in Chapter 2, "Inside an Ajax Application," the `validatekit.js` file offers several JavaScript functions you can use to validate different types of data. More specifically, the `validatekit.js` file exposes the following JavaScript functions that can be used for validating user input data:

- `validateNonEmpty()`—Is the data empty?
- `validateInteger()`—Is the data a valid integer?
- `validateNumber()`—Is the data a valid number?
- `validateZipCode()`—Is the data a valid ZIP code?
- `validatePhone()`—Is the data a valid phone number?
- `validateEmail()`—Is the data a valid email address?
- `validateDate()`—Is the data a valid date?

As you can see, these functions just so happen to match up perfectly with the data types targeted by the Validator application. This means that the challenge just got a whole lot easier because you now have standard JavaScript functions you can use to validate each data type. The challenge primarily becomes figuring out how to wire these functions into the HTML elements on the Validator page so that data within text boxes is properly routed to the JavaScript functions. Also, you'll need to create HTML elements for displaying the help text for each text box so that the validation functions can display help text.

At this point you may be wondering exactly where Ajax fits into the Validator equation, and the answer is that it doesn't… at least not yet. All of the data types targeted by the application can be validated perfectly fine within the client without any assistance from a server. Or at least the format of the data can be validated on the client. ZIP code data is a little unusual because you can validate the format of a ZIP code (exactly five integer numbers, for example) on the client but you won't know if the ZIP code is truly valid in the real world. In other words, is the ZIP code actually in use?

The only way to check for the validity of a ZIP code is to look it up in a database of real ZIP codes to see if it is indeed real. This database would be able to tell you that 90210 is indeed a real ZIP code and that it represents Beverly Hills, CA. As it turns out, such databases already exist and can be accessed via Ajax requests. Even better, most of them can provide you the data in an XML format, making it easy to sort out in your Ajax code. Ajax therefore enters the picture with the Validator application by way of looking up the city and state for a ZIP code to verify that the ZIP code is real.

So now you have a fairly complete validation challenge involving several client-side data validations as well as a true Ajax validation involving ZIP codes and their respective cities and states. You can now turn your attention to the design of the Validator application.

THE DESIGN: DESIGNING A VALIDATOR APPLICATION

The design of the Validator application is hopefully a little more obvious than some of the other applications in the book, primarily because discrete pieces of user input data are validated independently of one another. Your design job is basically just to call the appropriate `validatekit.js` validation function for each user-input text box. Then there is the Ajax request that provides an additional level of validation for ZIP codes by looking up the city and state from a Web service.

Figure 7.1 shows a rough interpretation of the user interface for the application, which reveals how the help fields are arranged next to each input text box.

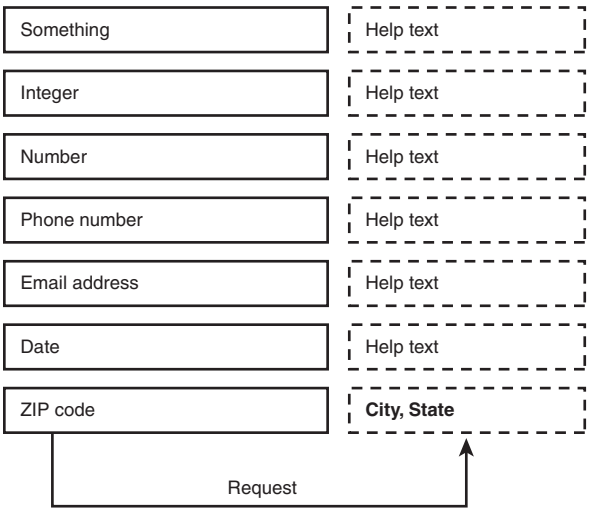

FIGURE 7.1 The user interface for the Validator application consists of several text boxes and their associated help text fields.

You can see in the figure how each text box is associated with a particular kind of data, such as a phone number, an email address, etc. The help field next to each text box is used to convey help information about the validity of the data in that text box. The only exception is the help field for the ZIP code, which serves double duty as a display field for the city and state retrieved via an Ajax request.

Speaking of the Ajax request for the ZIP field, it would be quite handy to display an animated "loading" image to let the user know that the application is busy looking up the city and state based upon the ZIP code. This image, `wait.gif`, is shown in Figure 7.2.

FIGURE 7.2 The Validator application makes use of an animated "loading" image (`wait.gif`) to let the user know the Ajax request is being processed.

The "loading" image is displayed in the same field that holds the help text for the ZIP code, which I suppose makes the field actually perform triple duty. The following three pieces of information are capable of appearing in the ZIP code's help field at different times throughout the process of entering a ZIP code:

- Help text is shown when the ZIP code is invalid.
- The "loading" image is shown while an Ajax request is being carried out.
- The dynamically loaded city and state are shown after a successful Ajax request.

This may seem like a lot of stuff is going on with respect to the ZIP code and its helper field, but the Validator application handles it all without much complication. Keep in mind that the application is really only dealing with one Ajax request, which also helps to simplify things. You're now ready to learn a little more about the ZIP code Ajax request.

The Client Request

The client request for the Validator application is very basic in that it sends along the ZIP code when the ZIP code text-input control loses focus. This approach is very different from the Completer application in Chapter 6, "Reading the User's Mind with Auto-Complete," which issues an Ajax request after each individual character is typed. The character-by-character request technique doesn't make sense for Validator because it takes multiple keypresses to enter a meaningful piece of data in Validator—you must wait until the user is finished entering the data before attempting a validation. Your cue that the user is finished entering data in the ZIP code input control is when he or she leaves the control by tabbing away from it or clicking off of it.

Once the user has left the ZIP code text box, you're free to send along the input as part of an Ajax request. This user input is just raw characters of text without any special formatting. The integer validation that has already been performed does ensure that the ZIP code text consists of five integer characters, but the text is still considered unformatted in a sense that it isn't encoded in XML or any other standard data format.

NOTE

I realize there is a longer ZIP code format of the form #####-#### that could feasibly enter the picture here. To keep things simple, I've opted to stick with the basic five-digit ZIP code, which has the form #####.

That's all you need to know about the minimal amount of data sent from the client to the server for a ZIP code lookup. Let's now address what the server sends back.

The Server Response

The Ajax server response in the Validator application returns information about the ZIP code sent over in the request. This response is indirectly received from a third party Web service, webserviceX.net, that offers several handy data feeds in XML format. In this case, the webserviceX.net service uses a ZIP code to look up and return a matching city, state, area code, and time zone. Since all you need in the Validator application is the city and state, you can ignore the area code and time zone. The fact that the response data is formatted as XML code makes it easy to access only the specific data you need.

You may recall from some of the earlier applications that you can't make an Ajax request on a third-party domain. Therefore, the trick is to alleviate this security limitation by making the third-party data request from a server script, and then using Ajax to get the data from that script, which resides on your server. This works because there isn't a security issue with requesting data from a server script. So the Validator application needs to include a PHP server script that serves as a proxy for making ZIP code requests to the webserviceX.net service.

Making Sense of the Client-to-Server Conversation

The Ajax request within the Validator application is used to shuttle a user-entered ZIP code to the server and then receive a city and state in return. Figure 7.3 helps to clarify this transfer of data visually.

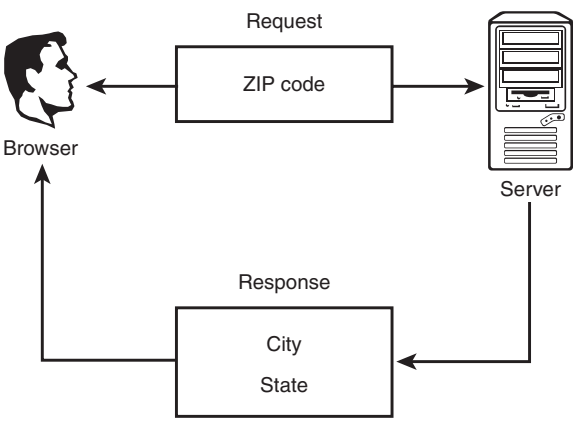

FIGURE 7.3 The Validator application involves an unformatted ZIP code and formatted city and state data flowing back and forth between the client and server.

The data sent to the server as a parameter of the Ajax request is a ZIP code that has been entered by the user. The ZIP code has already been validated in terms of its format but has not been checked to make sure it really matches up with a city and state. ZIP code requests are handled by the `ziplookup.php` server script. Following is an example of an Ajax request URL that sends along the ZIP code 85250 as the only parameter to the server script:

```
ziplookup.php?zipCode=85250
```

This code reveals how the ZIP code is packaged into a URL as a single parameter (`zipCode`). This entire URL is used as the basis for an Ajax request, in which case your server script, `ziplookup.php`, will take the `zipCode` parameter and use the URL it contains to obtain the city and state information.

The city and state data is returned as a compact XML document that actually contains the city, state, area code, and time zone for the ZIP code, assuming the ZIP code is successfully matched. Following is an example of what such an XML document actually looks like:

```
<NewDataSet>
  <Table>
    <CITY>Scottsdale</CITY>
    <STATE>AZ</STATE>
    <ZIP>85250</ZIP>
    <AREA_CODE>602</AREA_CODE>
    <TIME_ZONE>M</TIME_ZONE>
  </Table>
</NewDataSet>
```

Even if you aren't an XML guru, this code isn't too terribly hard to figure out. You're really only interested in the `<CITY>` and `<STATE>` tags because they contain the city and state for the ZIP code. The remaining task is then to extract the city and state from this data on the client.

Putting Together the Validator Application

Aside from the standard toolkit JavaScript files, including `validatekit.js`, the Validator application consists of only two components:

- HTML Web page (`validator.html`)
- PHP server script (`ziplookup.php`)

Of course, the other major ingredient in the application is the third party location Web service that is responsible for providing the city/state data based upon a

ZIP code. Although this service is an important part of the application, it doesn't require a distinct component other than code within the HTML Web page that initiates an Ajax request and code within the PHP server script that handles relaying the response back to the page. This arrangement is necessary because Ajax requests aren't allowed direct access to resources located on other domains.

The next couple of sections explore the Validator Web page and server script in detail in order to fully understand how the application works.

The Validator Web Page

The HTML Web page for the Validator application, `validator.html`, consists of a series of text box input fields, each with a non-input text field next to it that serves as a display area for help messages. The text boxes are created as `input` elements, while the help message fields are created as `span` elements. Figure 7.4 shows the Validator application open in a Web browser with the blank text fields ready for user input and all of the help messages indicating that the fields are missing data.

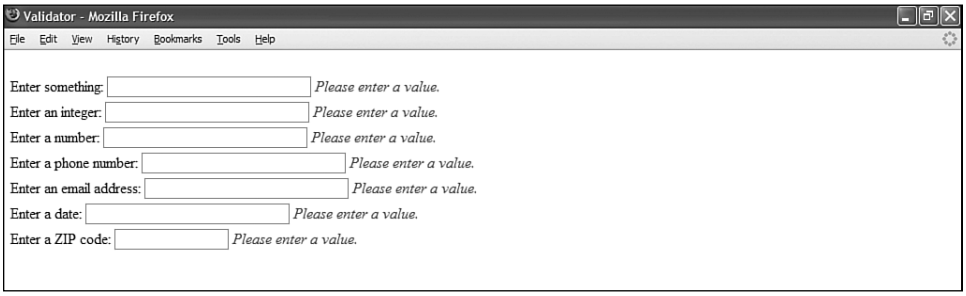

FIGURE 7.4 The Validator Web page consists of a series of text box input controls followed by text display areas.

NOTE

The application doesn't automatically display help messages initially as shown in the figure. Instead, the "Please enter a value." help message doesn't appear until you actually navigate away from an input control without entering anything.

With the layout of the Validator application in mind, as revealed in the figure, check out the HTML code for the body of the Web page:

```
<body onload="document.getElementById('something').focus()">
  <div id="ajaxState" style="display:none; font-style:italic">
  </div>
```

```
  <br />
  <div>
    Enter something:
    <input id="something" type="text" size="32" onblur=
      "validateNonEmpty(this,
document.getElementById('something_help'))" />
    <span id="something_help" class="help"></span>
  </div>
  <div>
    Enter an integer:
    <input id="integer" type="text" size="32" onblur=
      "validateInteger(this,
      document.getElementById('integer_help'))" />
    <span id="integer_help" class="help"></span>
  </div>
  <div>
    Enter a number:
    <input id="number" type="text" size="32" onblur=
      "validateNumber(this,
      document.getElementById('number_help'))" />
    <span id="number_help" class="help"></span>
  </div>
  <div>
    Enter a phone number:
    <input id="phone" type="text" size="32" onblur=
      "validatePhone(this,
      document.getElementById('phone_help'))" />
    <span id="phone_help" class="help"></span>
  </div>
  <div>
    Enter an email address:
    <input id="email" type="text" size="32" onblur=
      "validateEmail(this,
      document.getElementById('email_help'))" />
    <span id="email_help" class="help"></span>
  </div>
  <div>
    Enter a date:
    <input id="date" type="text" size="32" onblur=
      "validateDate(this,
      document.getElementById('date_help'))" />
    <span id="date_help" class="help"></span>
  </div>
  <div>
    Enter a ZIP code:
    <input id="zipcode" type="text" size="16"
      onblur="if (validateZipCode(this,
        document.getElementById('zipcode_help')))
        getCityState(this.value);" />
    <span id="zipcode_help" class="help"></span>
  </div>
</body>
```

After the focus is set to the first text box on the page via the `onload` event handler, the body of the page moves on to creating each of the text box `input` and associated help message `span` pairs. There isn't much particularly unusual about any of this code except for the `onblur` event handler, which gets called when the input focus leaves an input control. So, when the user tabs away from a text box or clicks somewhere else on the page, the text box receives an `onblur` event notification. This is where you call a function to validate the input value in the text box.

The functions that take care of the validating in the Validator application are all part of the `validatekit.js` Ajax Toolkit file, which must be imported into the Web page with the following line of code (placed in the head of the page):

```
<script type="text/javascript" src="validatekit.js"> </script>
```

Each of the functions provided by this file accept the same two function parameters:

- The input control containing the value to be validated
- The help control used to display help text

To see how these parameters get passed into a validation function, take a look at the code that calls the integer validation function:

```
validateInteger(this, document.getElementById('integer_help'))
```

Because the function is called from within the context of the integer text box, you can pass `this` as the first parameter to the function. The second parameter is then specified using the standard `getElementById()` function to grab the integer help control, whose ID is `integer_help`.

If you pay close attention to the help controls on the page, you may notice that each of the `span` elements is styled to a CSS style class named `help`. This style class simply changes the color of the font to red (`#FF0000`) and the style of the font to `italic`, which makes the help messages jump out a little more on the page. Following is the code for the Validator Web page's internal style sheet, which includes the `span.help` style class:

```
<style type="text/css">
  div { padding-bottom:5px; }
  span.help { color:#AA0000; font-style:italic; }
</style>
```

A global `div` style is included in the style sheet purely to help provide a bit of vertical spacing between the input controls on the page.

Although the Validator internal style sheet is helpful in making the page look a little better, it doesn't do much to help on the Ajax front. The code that initiates Ajax requests for the Validator application is contained in the `onblur` event handler for the ZIP code text box. Although this code is in the listing you just saw, it's easier to follow if you break it out of the `onblur` attribute:

```
if (validateZipCode(this,
  document.getElementById('zipcode_help')))
  getCityState(this.value);
```

This code first validates the ZIP code input data to see if it is indeed a five-digit number. This is accomplished with the call to the `validateZipCode()` function. The return value of the function indicates whether the data is valid (`true`) or invalid (`false`). If the data is valid, the `getCityState()` function is called to initiate an Ajax request. Here's the code for this function:

```
function getCityState(zipCode) {
  // Display the wait image
  document.getElementById("zipcode_help").innerHTML =
    "<img src='wait.gif'
    alt='Looking up ZIP code...' />";

  // Send the Ajax request to load the city/state
  ajaxSendRequest("GET", "ziplookup.php?zipCode=" + zipCode,
    handleCityStateRequest);
}
```

Ah, you finally see some of the magic that makes this application look cool. The first task in the `getCityState()` function is to display the animated "loading" image, `wait.gif`, in the ZIP code help control. This image will remain displayed until the Ajax request is completed, thereby giving the user a visual cue that something is taking place behind the scenes.

The second task in the `getCityState()` function is to actually submit the Ajax request, which involves packaging the ZIP code into a URL for the `ziplookup.php` server script. The `handleCityStateRequest()` function is specified as the request handler for the ZIP code Ajax request. Here's the code for it:

```
function handleCityStateRequest() {
  if (request.readyState == 4 && request.status == 200) {
    // Store the XML response data
    var xmlData = request.responseXML;

    // Display the city/state results
    if (xmlData != null &&
      getText(xmlData.getElementsByTagName("CITY")[0]) != "")
      document.getElementById("zipcode_help").innerHTML =
        "<span style='font-weight:bold'>" +
```

```
        getText(xmlData.getElementsByTagName("CITY")[0]) +
        ", " +
        getText(xmlData.getElementsByTagName("STATE")[0]) +
        "</span>";
    else
      document.getElementById("zipcode_help").innerHTML =
        "Could not find the ZIP code.";
  }
  ajaxUpdateState();
}
```

The job of this function, which is called when the Ajax request completes, is to extract the city and state from the response data and display them on the page in place of the "loading" image. The city/state data is extracted with a little help from the `getText()` Ajax Toolkit function, which is located in the `domkit.js` file. Notice that the XML element names `CITY` and `STATE` are used as the basis for accessing the city and state data. Not only is this data extracted, but it is carefully reformatted using HTML code constructed on the fly. If all goes well, the resulting HTML code looks something like this:

```
<span style='font-weight:bold'>Scottsdale, AZ</span>
```

Notice that an inline style is applied to the city and state so that they appear in a bold font, which helps to make them distinguishable from normal help text. If there is a problem with the XML data returned from the server, a help message is displayed indicating that the ZIP code could not be found.

That wraps up the code for the Validator Web page. All that's left is to take a quick look at the PHP server script responsible for passing along the ZIP code request to a remote server.

The ZIP Lookup Server Script

Similar to the Picker application from Chapter 4, "Using Ajax to Dynamically Populate Lists: A Stock Picker," Validator requires a script on the server side to sidestep the limitation of not being able to make a direct Ajax client request on a third-party domain. The third-party domain is important because the Validator application relies on it for looking up ZIP codes. The Validator PHP script serves as somewhat of a security proxy by accepting an Ajax request on behalf of the client, retrieving the city/state data from the third-party server, and then responding to the client with the data. In this way, the Ajax request itself is made on your own domain, which is perfectly acceptable given the security constraints of Ajax.

The third-party Web service I'm talking about is webserviceX.net, which offers several XML-based data feeds. The Picker application in Chapter 4 uses this

service to obtain live stock quotes, whereas Validator uses it to obtain detailed information about a ZIP code. In this case you specify a ZIP code, and the server responds with an XML document containing detailed information about it, including the city and state associated with it. You've already seen the code within this XML document, but you haven't seen the PHP script that makes it possible to receive the code via an Ajax request. Following is the code for the `ziplookup.php` server script:

```
<?php
header('Content-Type: text/xml');

// Load the XML location data from the server
$xml = html_entity_decode(file_get_contents(
  "http://www.webservicex.net/uszip.asmx/GetInfoByZIP?USZip=" .
  $_REQUEST['zipCode']));

// Return the XML location data
echo $xml;
?>
```

Keep in mind that the job of this script is to retrieve an XML document from the ZIP code Web service at webserviceX.net, and then simply pass that data along to the client without any modifications. It is then up to the client to do something useful with the XML code, such as display the city and state on the page.

The server script begins by making sure that the response gets treated as XML. It then loads the XML document from the ZIP code Web service, making sure to pass along the `zipCode` parameter so that the Web service knows which ZIP code to look up. The script concludes by returning the XML data from the script, which serves as the response to the Ajax ZIP code request.

Although I'm sure you're itching to test out the application, let's take a second to test the PHP script by itself. You saw how this is done in earlier chapters by opening PHP scripts directly in a browser—here's a sample URL for testing the ZIPLookUp script:

```
http://www.yourdomain.com/ziplookup.php?zipCode=94965
```

Assuming the `ziplookup.php` file is stored on your server, change the domain to match yours in this line of code and then enter the URL into your Web browser. Figure 7.5 shows the XML response data generated by the PHP script.

It's definitely worth experimenting with the ZIP codes passed into the script via the URL to see how they impact the XML data that is returned. You should always test PHP scripts directly in this manner to make sure their results are consistent with what you expected—it's much tougher to diagnose server script problems when all you're looking at is the client Web page.

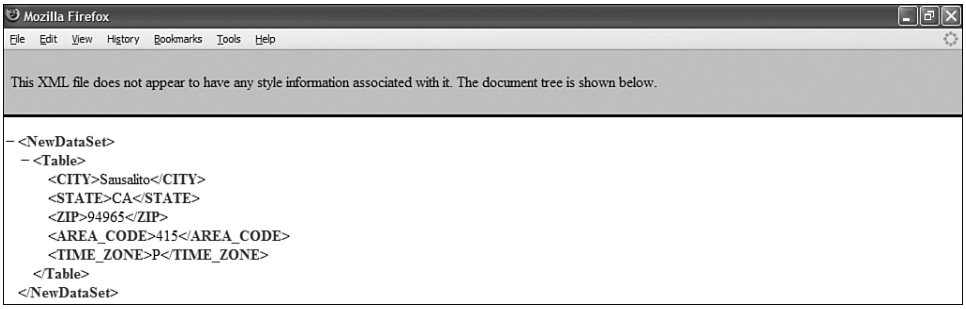

FIGURE 7.5 You can examine the XML ZIP code information returned by the `ZIPLookUp` PHP script by opening the script directly in a browser.

TESTING VALIDATOR

The Validator application presents an interesting testing opportunity because you have several different data fields to tinker with as you check the effectiveness of the validation. You'll notice that the data types tend to progress in complexity as you move down the page. It's easy to generate a help message by either entering nothing or entering invalid data into any of the input controls, and then tabbing off of the control. Figure 7.6 shows a help message being displayed next to invalid integer data.

FIGURE 7.6 Entering invalid data and then leaving an input control results in the display of a help message next to the control.

Of course, the text "more data" doesn't exactly constitute a valid integer, which explains the help message in the figure.

Things get more interesting as you experiment with entering data into more complex types, such as the date input control. Figure 7.7 shows an invalid date message displayed in response to a nearly valid date.

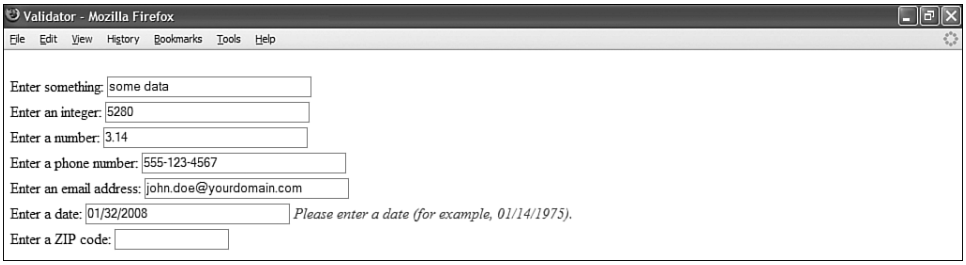

FIGURE 7.7 The date input control is pretty good at detecting bad dates.

Obviously the month of January only has 31 days, so a date involving the 32nd day of the month is invalid in this example. The Validator application catches the input error and displays a help message.

Although date validation is handy, the most intriguing part of the Validator application is the lookup of ZIP codes, which takes place when you enter a ZIP code into the ZIP code input control and then leave the control. Figure 7.8 shows the animated "loading" image displayed as the ZIP code Ajax request is being carried out.

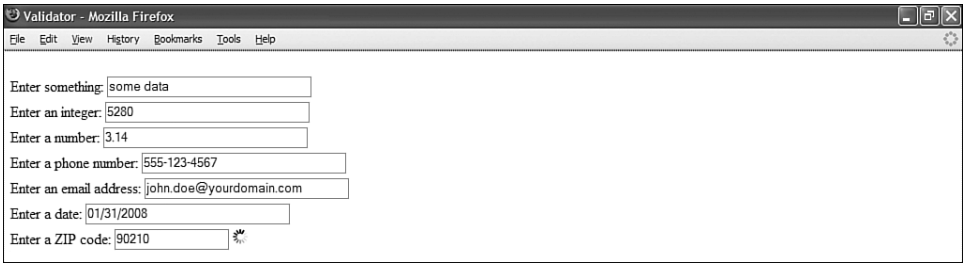

FIGURE 7.8 The "loading" image is displayed in lieu of a help message to let the user know that the ZIP code is being looked up.

Yeah, I know, I couldn't resist using the most popular ZIP code on Earth to test out the application. If the ZIP code data is successfully retrieved via the Ajax request, the resulting city and state are displayed in place of the "loading" image, as shown in Figure 7.9.

Success is great, but what happens when a ZIP code isn't found? If you recall in the code for the application, a help message is displayed to indicate this. Figure 7.10 shows the help message displayed in response to a ZIP code that isn't associated with a real city and state.

FIGURE 7.9 A successful ZIP code Ajax request results in the display of the associated city and state next to the input control.

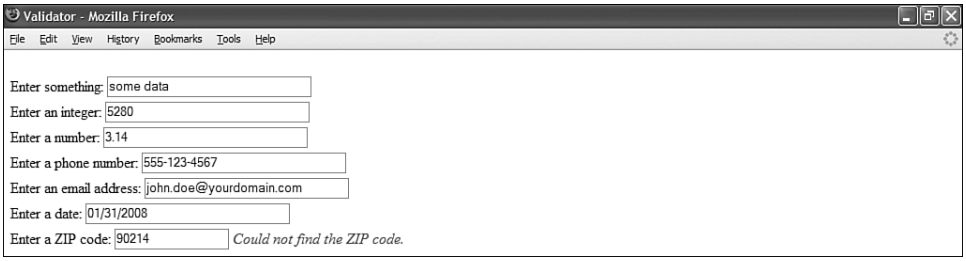

FIGURE 7.10 A ZIP code with no real-world city and state results in a help message indicating that the ZIP code couldn't be found.

It's important to note that a ZIP code that isn't found isn't necessarily invalid in a strict data entry sense. Yes, it isn't in use in the real world, which is a problem if you're entering a shipping address, for example, but that doesn't mean it won't be real someday. ZIP codes are not etched in stone, and in fact they are created anew on a fairly regular basis in fast-growing areas. That's the beauty of relying on a third-party service to handle the ZIP code lookup—it is responsible for keeping its ZIP code database up to date, not you.

Regardless of how you handle the issue of a ZIP code not being used in the real world, hopefully you can see how the Validator application lays some important groundwork in data validation that you can reuse and repurpose in all kinds of useful ways.

Giving Validator an Extreme Ajax Makeover

The Validator application is one of the more immediately applicable examples in the book because intelligent data entry is a common need. The `validatekit.js` file

offers functions for common data types that are often used in data entry forms. The first way to expand upon the Validator example is to apply the validation functions to your own user-input forms. This is as easy as importing the `validatekit.js` file into your pages via the following `<script>` tag:

```
<script type="text/javascript" src="validatekit.js"> </script>
```

Then code the HTML elements for validation by calling the appropriate validation function and passing along the IDs of the HTML elements. As an example, if you wanted to validate the age of a person to make sure it is an integer number, you might create the following text box and help field:

```
<input id="age" type="text" size="4"
  onblur="validateInteger(this,
  document.getElementById('age_help'))" />
<span id="age_help" class="help"></span>
```

You don't have to associate a style class with the help text, but it does make the user interface look a little better. Here's an example of a help style that you can use to dress up the help fields:

```
<style type="text/css">
  span.help { color:#AA0000; font-style:italic; }
</style>
```

Keep in mind that this style must be placed in the head of the Web page, which is also where the `<script>` tag you saw a moment ago must be placed. This style just makes the help text italicized and red. You can easily add on more styles or tweak these to get a different look.

The one problem with this age validator is that it doesn't account for the user entering an age that doesn't make sense. For example, because the validation only checks to see if the input is an integer, no help text is displayed if the user enters an age of -5 or 212. And until someone uncovers the fountain of youth, an age much over 100 can probably be considered invalid. You could account for negative ages by possibly creating a whole new validation function that only allows positive integers, but in this particular case a range validation makes more sense.

The age range validation is responsible for first checking that an input value is an integer, and then further checking to see if the integer is within an acceptable range, such as 1 to 120. Following is how you might put together a validation function to carry out this task:

```
function validateAge(inputControl, helpControl) {
  // First see if the input value is an integer
  if (!validateInteger(inputControl, helpControl))
    return false;
```

```
  // Then see if the input value is in the prescribed range
  if (inputControl.value < 1 || inputControl.value > 120) {
    if (helpControl != null)
      helpControl.innerHTML = "Please enter an age in the
      range 1 - 120.";
    return false;
  }
  else {
    if (helpControl != null)
      helpControl.innerHTML = "";
    return true;
  }
}
```

Notice that this function first calls the `validateInteger()` function to check and make sure the input is indeed a valid integer. There's no sense in moving on to the range check if the input isn't an integer. If the input is an integer, a range check is performed. The result of the range check is to simply set or clear text in the help control. The help control is checked for a `null` value in the event that you forget or decide not to include a help control, in which case you don't want an error to occur.

Even if your JavaScript experience is shaky or nonexistent, you should be able to isolate the key part of the `validateAge()` function and modify it to fit your needs. For example, you can change the range of the validation check by simply changing this line of code:

```
if (inputControl.value < 1 || inputControl.value > 120) {
```

Let's say you want to validate an NFL (National Football League) quarterback rating, which must be in the range 0 to 158. Here's how you would change the range-testing line of code:

```
if (inputControl.value < 0 || inputControl.value > 158) {
```

NOTE

Please don't ask why an NFL quarterback rating is in the range 0 to 158 (actually 158.33), as opposed to something more logical such as 0 to 10. It's a tricky calculation that factors in variables such as pass completion percentage, yards per attempt, touchdowns per attempt, etc. Unless you're a total (American) football nerd, just accept the range and move on. Otherwise, feel free to learn more at http://en.wikipedia.org/wiki/Passer_rating.

Hopefully you can see that crafting your own validation functions out of the base set I've provided isn't terribly difficult.

It's also possible to extend the Ajax-style validation employed by the Validator application in looking up the city and state of a ZIP code. For example, a popular Ajax validation is to check a database of user names to ensure uniqueness when registering a new user. In this case, you would issue an Ajax request to your server script that passes along the user name entered, and the server script then checks the name against a database of existing user names to see if it is unique. The response could be as simple as "yes" or "no" for the uniqueness validation. So the client just needs to look at this result and display a help message accordingly.

The point to this makeover discussion is that I've really only exposed the tip of the iceberg in terms of data validation and Ajax. It's up to you to take what you've learned and the code I've provided to further expand on the validation techniques as needed.

Summary

Although this book presents lots of interesting Ajax applications, some with impressive visuals and live data feeds, from a purely practical perspective few of them are as immediately handy as the Validator application. The application has a very basic user interface, but it's the usability that shines here. The help text that appears when invalid data is entered is subtle but compelling. And consider that this help information appears immediately without you having to submit any data—this is the "Ajax" way of doing things, even when an Ajax server request isn't explicitly being made. Of course, the Validator application does involve an Ajax server request for obtaining the city and state for a ZIP code, but the fact remains that the application promotes an "Ajax" feel throughout, even when the validation is taking place solely on the client.