Chapter 16
Ajax-Enabled Rich Internet Applications with XML and JSON

16.1 Introduction

- Usability of web applications has lagged behind that of desktop applications
- Rich Internet Applications (RIAs)
  - Web applications that approximate the look, feel and usability of desktop applications
  - Two key attributes—performance and rich GUI
- RIA performance
  - Comes from Ajax (Asynchronous JavaScript and XML), which uses client-side scripting to make web applications more responsive
- Ajax applications separate client-side user interaction and server communication, and run them in parallel, making the delays of server-side processing more transparent to the user
- "Raw" Ajax uses JavaScript to send asynchronous requests to the server, then updates the page using the DOM
- When writing "raw" Ajax you need to deal directly with cross-browser portability issues, making it impractical for developing large-scale applications

16.1 Introduction (Cont.)

- Portability issues
  - Hidden by Ajax toolkits, such as jQuery, ASP.NET Ajax and JSF's Ajax capabilities, which provide powerful ready-to-use controls and functions that enrich web applications and simplify JavaScript coding by making it cross-browser compatible.
- Achieve rich GUI in RIAs with
  - JavaScript toolkits, providing powerful ready-to-use controls and functions that enrich web applications.
- Client-side of Ajax applications
  - Written in HTML5 and CSS3
  - Uses JavaScript to add functionality to the user interface
- XML and JSON are used to structure the data passed between the server and the client
- `XMLHttpRequest`
  - The Ajax component that manages interaction with the server
  - Commonly abbreviated as XHR.

16.1.2 Traditional Web Applications

- Traditional web applications
  - User fills in the form's fields, then submits the form
  - Browser generates a request to the server, which receives the request and processes it
  - Server generates and sends a response containing the exact page that the browser will render
  - Browser loads the new page and temporarily makes the browser window blank
  - Client *waits* for the server to respond and *reloads the entire page* with the data from the response
- While a synchronous request is being processed on the server, the user cannot interact with the client web browser
- The synchronous model was originally designed for a web of hypertext documents
  - some people called it the "brochure web"
  - model yielded erratic application performance

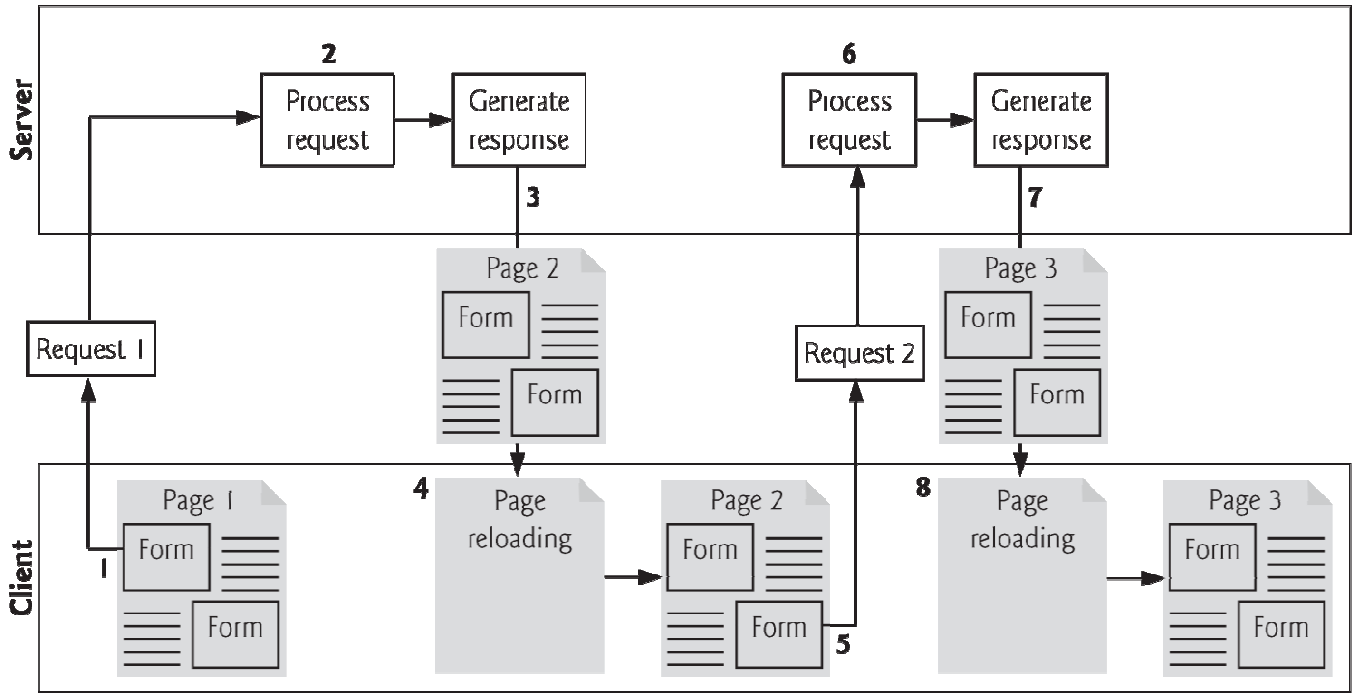


Fig. 16.1 | Classic web application reloading the page for every user interaction.

16.1.3 Ajax Web Applications

- In an Ajax application, when the user interacts with a page
  - Client creates an `XMLHttpRequest` object to manage a request
  - `XMLHttpRequest` object sends the request to and awaits the response from the server
  - Requests are asynchronous, allowing the user to continue interacting with the application while the server processes the request concurrently
  - When the server responds, the `XMLHttpRequest` object that issued the request invokes a callback function, which typically uses partial page updates to display the returned data in the existing web page *without reloading the entire page*
- Callback function updates only a designated part of the page
- Partial page updates help make web applications more responsive, making them feel more like desktop applications

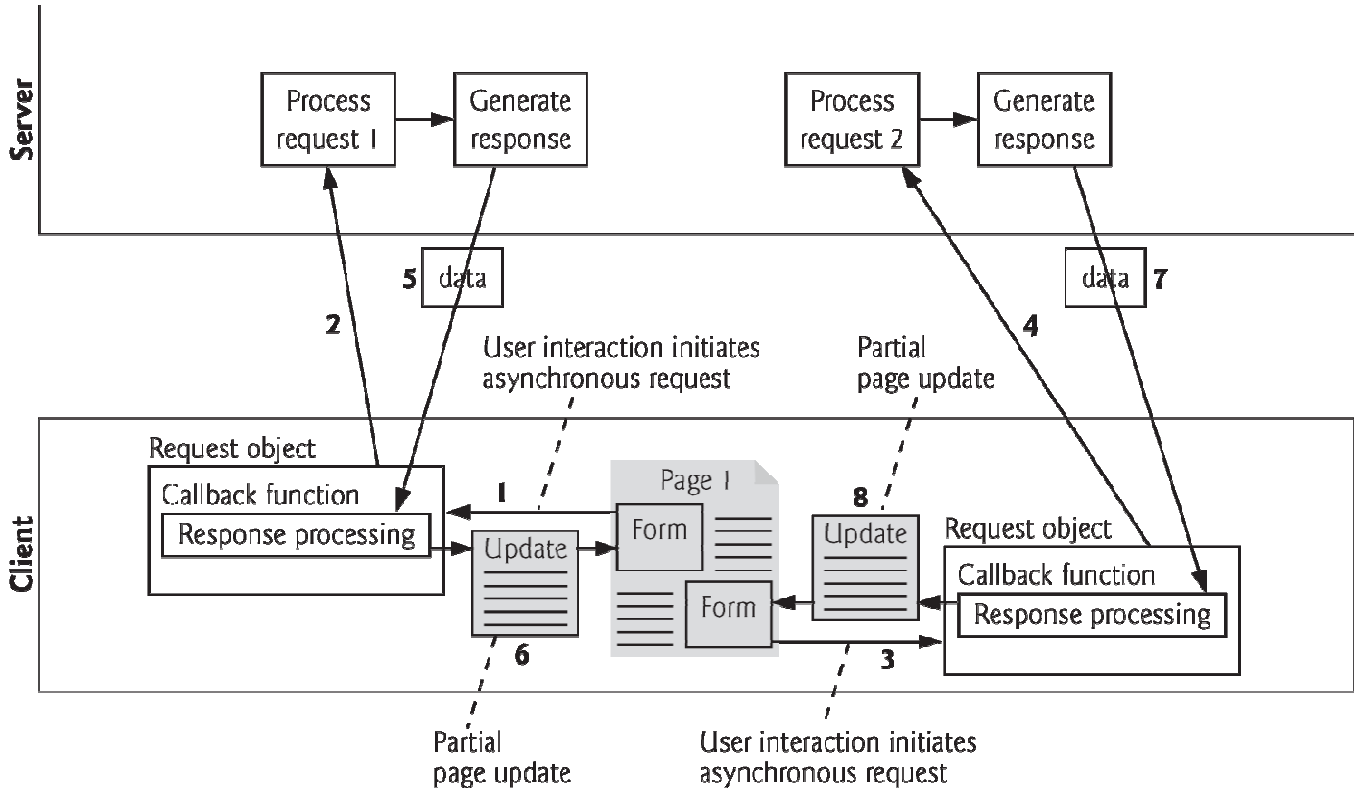


Fig. 16.2 | Ajax-enabled web application interacting with the server asynchronously.

16.2 Rich Internet Applications (RIAs) with Ajax

- Classic HTML registration form
  - Sends all of the data to be validated to the server when the user clicks the Register button
  - While the server is validating the data, the user cannot interact with the page
  - Server finds invalid data, generates a new page identifying the errors in the form and sends it back to the client—which renders the page in the browser
  - User fixes the errors and clicks the Register button again
  - Cycle repeats until no errors are found, then the data is stored on the server
  - Entire page reloads every time the user submits invalid data
- Ajax-enabled forms are more interactive
  - Entries are validated individually, dynamically as the user enters data into the fields
  - If a problem is found, the server sends an error message that is asynchronously displayed to inform the user of the problem
  - Sending each entry asynchronously allows the user to address invalid entries quickly, rather than making edits and resubmitting the entire form repeatedly until all entries are valid
  - Asynchronous requests could also be used to fill some fields based on previous fields' values (e.g., city based on zip code)

a) A sample registration form in which the user has not filled in the required fields, but attempts to submit the form anyway by clicking **Register**.

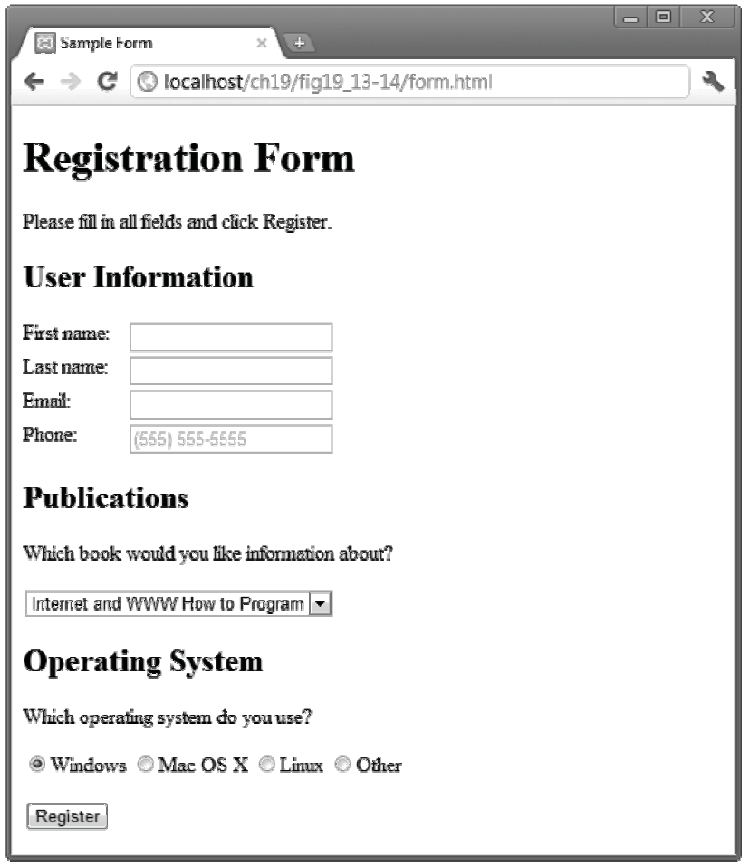


Fig. 16.3 | Classic HTML5 form: The user submits the form to the server, which validates the data (if any). Server responds indicating any fields with invalid or missing data. (Part 1 of 2.)

b) The server responds by indicating all the form fields with missing or invalid data. The user must correct the problems and resubmit the *entire* form repeatedly until *all* errors are corrected.

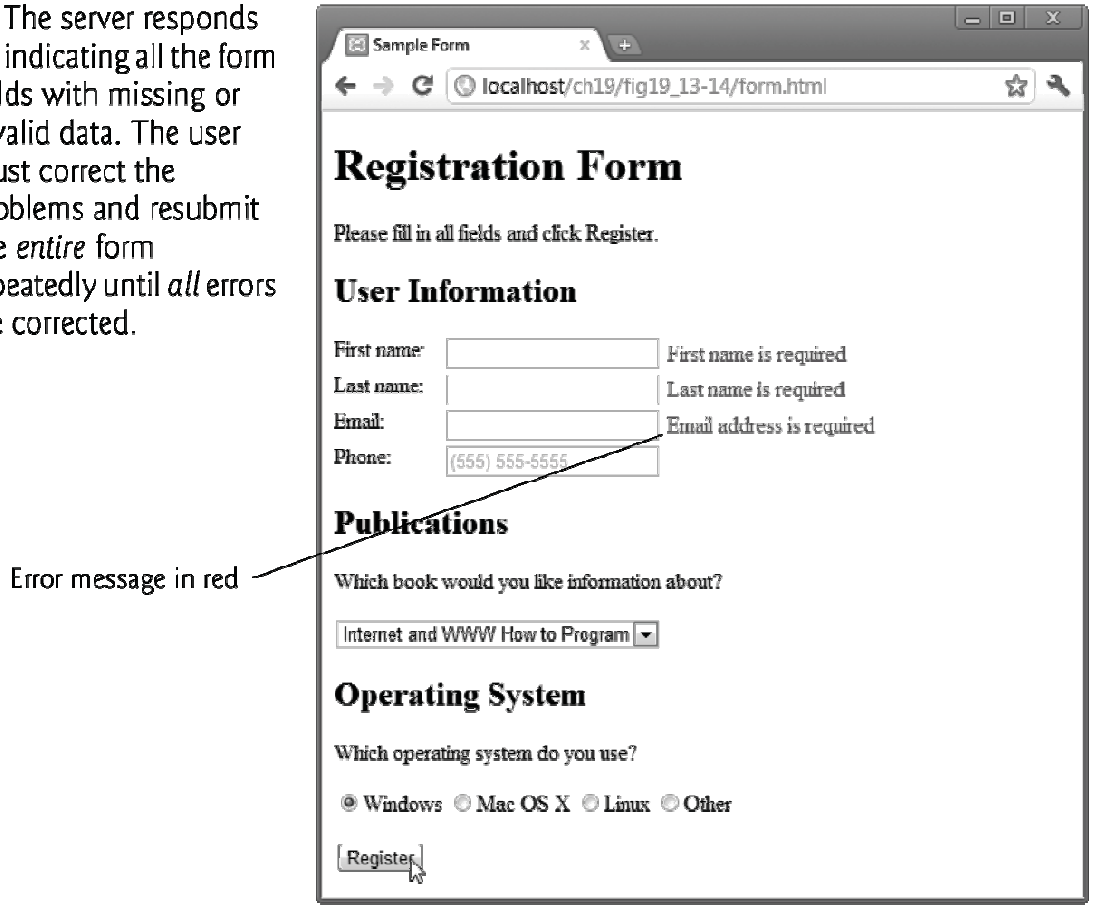


Fig. 16.3 | Classic HTML5 form: The user submits the form to the server, which validates the data (if any). Server responds indicating

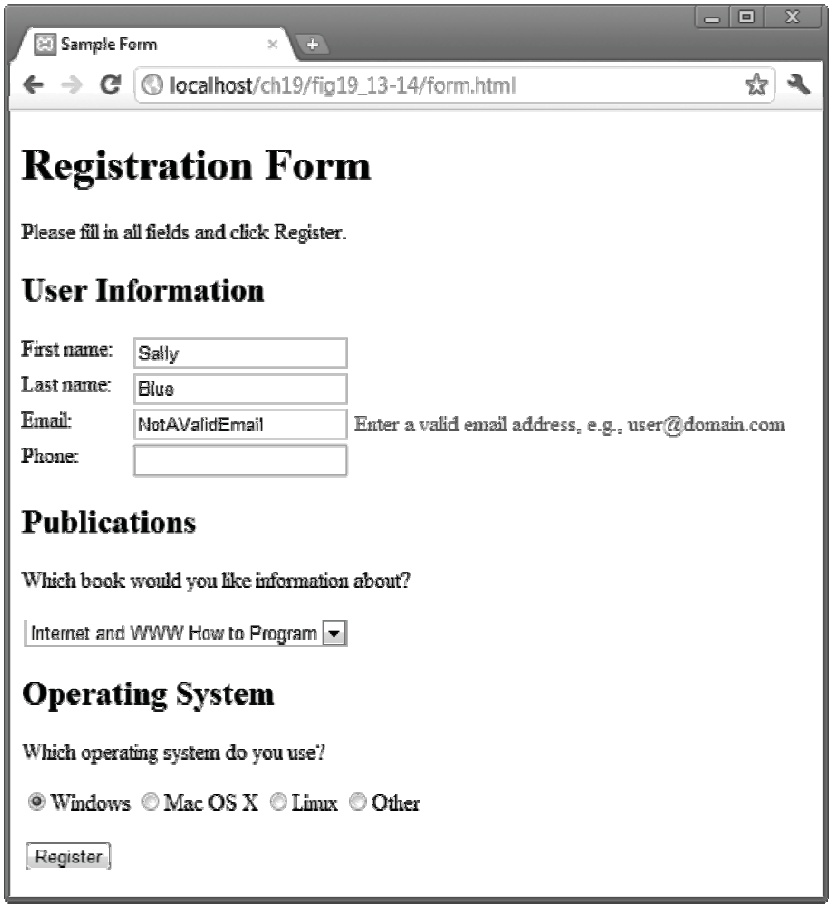


Fig. 16.4 | Ajax-enabled form shows errors asynchronously when user moves to another field.

16.3 History of Ajax

- The term Ajax was coined by Jesse James Garrett of Adaptive Path in February 2005, when he was presenting the previously unnamed technology to a client
- Ajax technologies (HTML, JavaScript, CSS, dynamic HTML, the DOM and XML) had existed for many years prior to 2005
- In 1998, Microsoft introduced the `XMLHttpRequest` object to create and manage asynchronous requests and responses
- Popular applications like Flickr, Google's Gmail and Google Maps use the `XMLHttpRequest` object to update pages dynamically
- Ajax has enabled "webtop" applications to challenge the dominance of established desktop applications
- This has become increasingly significant as more and more computing moves to "the cloud."

16.4 "Raw" Ajax Example using the `XMLHttpRequest` Object

- `XMLHttpRequest` object
  - Resides on the client
  - Is the layer between the client and the server that manages asynchronous requests in Ajax applications
  - Supported on most browsers, though they may implement it differently
- To initiate an asynchronous request
  - Create an instance of the `XMLHttpRequest` object
  - Use its open method to set up the request, and its send method to initiate the request
- When an Ajax application requests a file from a server, the browser typically caches that file
  - Subsequent requests for the same file can load it from the browser's cache

16.4.1 Asynchronous Requests

- When the third argument to `XMLHttpRequest` method open is `true`, the request is *asynchronous*

```
<!DOCTYPE html>

<!-- Fig. 16.5: SwitchContent.html -->
<!-- Asynchronously display content without reloading the page. -->
<html>
<head>
   <meta charset = "utf-8">
   <style type = "text/css">
      .box { border: 1px solid black; padding: 10px }
   </style>
   <title>Switch Content Asynchronously</title>
   <script>
      var asyncRequest; // variable to hold XMLHttpRequest object

      // set up event handlers
      function registerListeners()
      {
         var img;
         img = document.getElementById( "cpphtp" );
         img.addEventListener( "mouseover",
            function() { getContent( "cpphtp8.html" ); }, false );
         img.addEventListener( "mouseout", clearContent, false );
         img = document.getElementById( "iw3htp" );
```

Fig. 16.5 | Asynchronously display content without reloading the page. (Part 1 of 7.)

```
24          img.addEventListener( "mouseover",
25             function() { getContent( "iw3htp.html" ); }, false );
26          img.addEventListener( "mouseout", clearContent, false );
27          img = document.getElementById( "jhtp" );
28          img.addEventListener( "mouseover",
29             function() { getContent( "jhtp.html" ); }, false );
30          img.addEventListener( "mouseout", clearContent, false );
31          img = document.getElementById( "vbhtp" );
32          img.addEventListener( "mouseover",
33             function() { getContent( "vbhtp.html" ); }, false );
34          img.addEventListener( "mouseout", clearContent, false );
35          img = document.getElementById( "vcshtp" );
36          img.addEventListener( "mouseover",
37             function() { getContent( "vcshtp.html" ); }, false );
38          img.addEventListener( "mouseout", clearContent, false );
39          img = document.getElementById( "javafp" );
40          img.addEventListener( "mouseover",
41             function() { getContent( "javafp.html" ); }, false );
42          img.addEventListener( "mouseout", clearContent, false );
43       } // end function registerListeners
44
```

Fig. 16.5 | Asynchronously display content without reloading the page. (Part 2 of 7.)

```
45         // set up and send the asynchronous request.
46         function getContent( url )
47         {
48            // attempt to create XMLHttpRequest object and make the request
49            try
50            {
51               asyncRequest = new XMLHttpRequest(); // create request object
52
53               // register event handler
54               asyncRequest.addEventListener(
55                  "readystatechange", stateChange, false);
56               asyncRequest.open( "GET", url, true ); // prepare the request
57               asyncRequest.send( null ); // send the request
58            } // end try
59            catch ( exception )
60            {
61               alert( "Request failed." );
62            } // end catch
63         } // end function getContent
64
```

Fig. 16.5 | Asynchronously display content without reloading the page. (Part 3 of 7.)

```
65          // displays the response data on the page
66          function stateChange()
67          {
68             if ( asyncRequest.readyState == 4 && asyncRequest.status == 200 )
69             {
70                document.getElementById( "contentArea" ).innerHTML =
71                   asyncRequest.responseText; // places text in contentArea
72             } // end if
73          } // end function stateChange
74
75          // clear the content of the box
76          function clearContent()
77          {
78             document.getElementById( "contentArea" ).innerHTML = "";
79          } // end function clearContent
80
81          window.addEventListener( "load", registerListeners, false );
82       </script>
83    </head>
```

Fig. 16.5 | Asynchronously display content without reloading the page. (Part 4 of 7.)

```
84   <body>
85      <h1>Mouse over a book for more information.</h1>
86      <img id = "cpphtp" alt = "C++ How to Program book cover"
87         src = "http://test.deitel.com/images/thumbs/cpphtp8.jpg">
88      <img id = "iw3htp" alt = "Internet & WWW How to Program book cover"
89         src = "http://test.deitel.com/images/thumbs/iw3htp5.jpg">
90      <img id = "jhtp" alt = "Java How to Program book cover"
91         src = "http://test.deitel.com/images/thumbs/jhtp9.jpg">
92      <img id = "vbhtp" alt = "Visual Basic 2010 How to Program book cover"
93         src = "http://test.deitel.com/images/thumbs/vb2010htp.jpg">
94      <img id = "vcshtp" alt = "Visual C# 2010 How to Program book cover"
95         src = "http://test.deitel.com/images/thumbs/vcsharp2010htp.jpg">
96      <img id = "javafp" alt = "Java for Programmers book cover"
97         src = "http://test.deitel.com/images/thumbs/javafp.jpg">
98      <div class = "box" id = "contentArea"></div>
99   </body>
100  </html>
```

Fig. 16.5 | Asynchronously display content without reloading the page. (Part 5 of 7.)

a) User hovers over *C++ How to Program* book-cover image, causing an asynchronous request to the server to obtain the book's description. When the response is received, the application performs a *partial page update* to display the description.

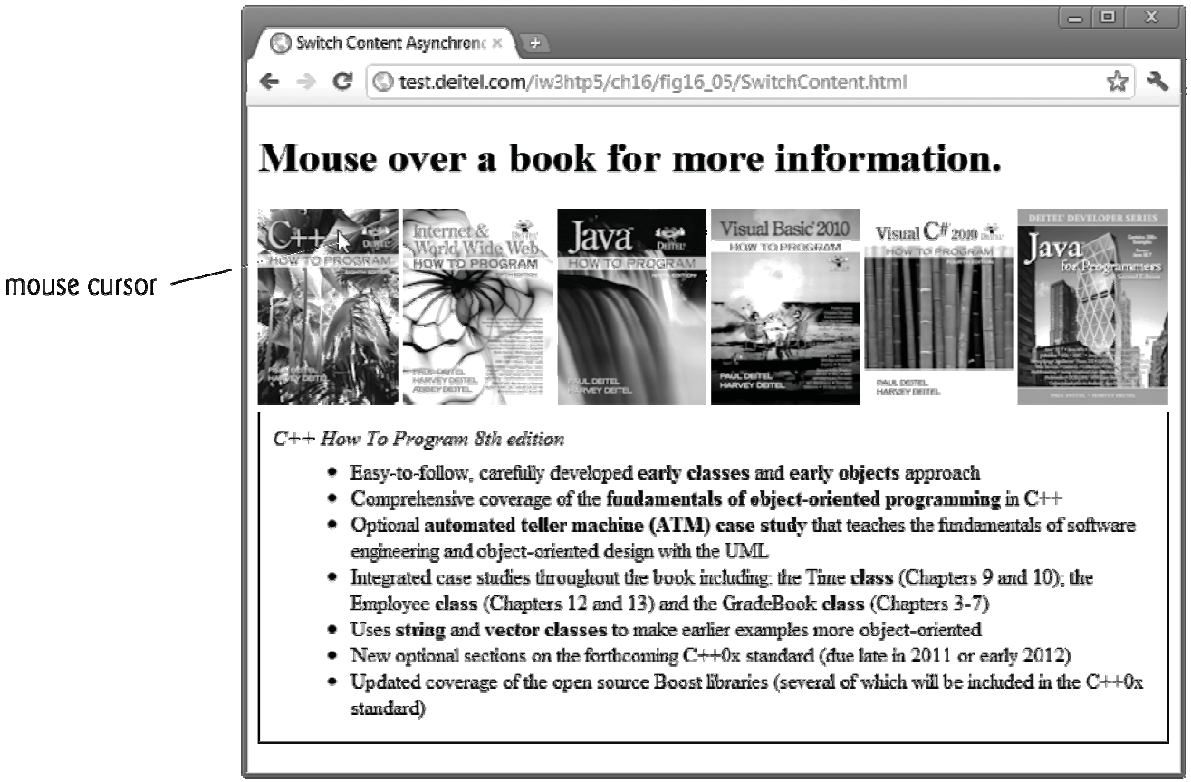


Fig. 16.5 | Asynchronously display content without reloading the page. (Part 6 of 7.)

b) User hovers over *Internet & World Wide Web How to Program* book-cover image, causing the process to repeat.

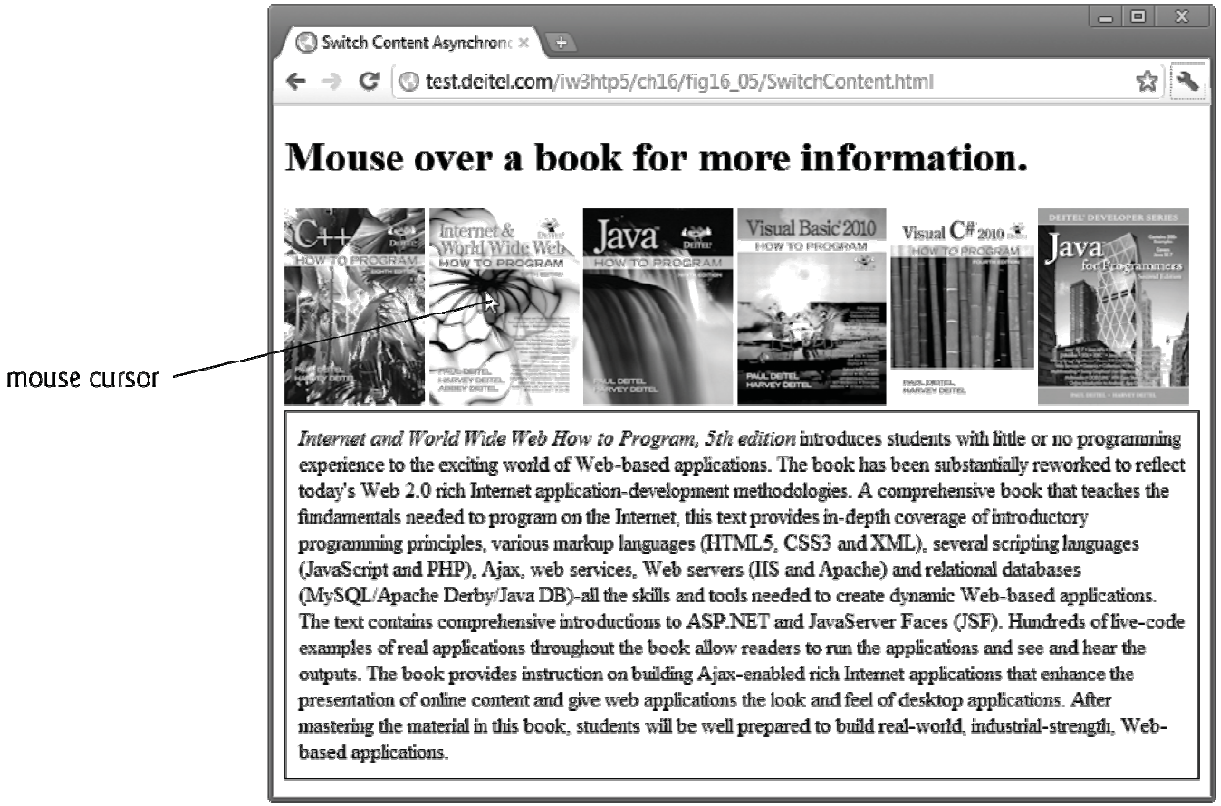


Fig. 16.5 | Asynchronously display content without reloading the page. (Part 7 of 7.)

http://test.deitel.com/iw3htp5/ch16/fig16_05/SwitchContent.html

16.4.2 Exception Handling

- An exception is an indication of a problem that occurs during a program's execution
- Exception handling enables you to create applications that can resolve (or handle) exceptions—in some cases allowing a program to continue executing as if no problem had been encountered
- `try` block
  - Encloses code that might cause an exception and code that should not execute if an exception occurs
  - Consists of the keyword `try` followed by a block of code enclosed in curly braces (`{}`)
- When an exception occurs
  - `try` block terminates immediately
  - `catch` block catches (i.e., receives) and handles an exception
- `catch` block
  - Begins with the keyword `catch`
  - Followed by an exception parameter in parentheses and a block of code enclosed in curly braces
- Exception parameter's name
  - Enables the `catch` block to interact with a caught exception object, which contains `name` and `message` properties

16.4.3 Callback Functions

- A callback function is registered as the event handler for the `XMLHttpRequest` object's `readystatechange` event
  - Whenever the request makes progress, the `XMLHttpRequest` calls the `readystatechange` event handler.
  - Progress is monitored by the `readyState` property, which has a value from 0 to 4
  - The value 0 indicates that the request is not initialized and the value 4 indicates that the request is complete.

16.4.4 XMLHttpRequest Object Event, Properties and Methods

- The following figures summarize some of the XMLHttpRequest object's properties and methods, respectively.
- The properties are crucial to interacting with asynchronous requests.
- The methods initialize, configure and send asynchronous requests.

Event or Property	Description
`readystatechange`	Register a listener for this event to specify the *callback* function—the event handler that gets called when the server responds.
`readyState`	Keeps track of the request's progress. It's usually used in the callback function to determine when the code that processes the response should be launched. The `readyState` value `0` signifies that the request is uninitialized; `1` that the request is loading; `2` that the request has been loaded; `3` that data is actively being sent from the server; and `4` that the request has been completed.
`responseText`	Text that's returned to the client by the server.
`responseXML`	If the server's response is in XML format, this property contains the XML document; otherwise, it's empty. It can be used like a `document` object in JavaScript, which makes it useful for receiving complex data (e.g., populating a table).

Fig. 16.6 | `XMLHttpRequest` object event and properties. (Part 1 of 2.)

Event or Property	Description
`status`	HTTP status code of the request. A `status` of `200` means that request was *successful*. A `status` of `404` means that the requested resource was *not found*. A status of `500` denotes that there was an *error* while the server was processing the request. For a complete status reference, visit `www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`.
`statusText`	Additional information on the request's status. It's often used to display the error to the user when the request fails.

Fig. 16.6 | `XMLHttpRequest` object event and properties. (Part 2 of 2.)

Method	Description
`open`	Initializes the request and has two *mandatory* parameters—method and URL. The method parameter specifies the purpose of the request—typically `GET` or `POST`. The URL parameter specifies the address of the file on the server that will generate the response. A third optional Boolean parameter specifies whether the request is *asynchronous*—it's set to `true` by default.
`send`	Sends the request to the server. It has one optional parameter, `data`, which specifies the *data to be `POST`ed to the server*—it's set to `null` by default.
`setRequestHeader`	Alters the request header. The two parameters specify the header and its new value. It's often used to set the `content-type` field.
`getResponseHeader`	Returns the header data that precedes the response body. It takes one parameter, the name of the header to retrieve. This call is often used to *determine the response's type*, to parse the response correctly.
`getAllResponseHeaders`	Returns an array that contains all the headers that precede the response body.
`abort`	Cancels the current request.

Fig. 16.7 | `XMLHttpRequest` object methods.

16.5 Using XML and the DOM

- When passing structured data between the server and the client, Ajax applications often use XML because it consumes little bandwidth and is easy to parse
- `XMLHttpRequest` object `responseXML` property
  - contains the parsed XML returned by the server
- DOM method `createElement`
  - Creates an HTML5 element of the specified type
- DOM method `appendChild`
  - Inserts one HTML5 element into another
- `innerHTML` property of a DOM element
  - Can be used to obtain or change the HTML5 that is displayed in a particular element

```
<!DOCTYPE html>

<!-- Fig. 16.8: PullImagesOntoPage.html -->
<!-- Image catalog that uses 1Ajax to request XML data asynchronously. -->
<html>
<head>
<meta charset="utf-8">
<title> Pulling Images onto the Page </title>
<style type = "text/css">
   li { display: inline-block; padding: 4px; width: 120px; }
   img { border: 1px solid black }
</style>
<script>
   var asyncRequest; // variable to hold XMLHttpRequest object

```

Fig. 16.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 1 of 8.)

```
16      // set up and send the asynchronous request to get the XML file
17      function getImages( url )
18      {
19         // attempt to create XMLHttpRequest object and make the request
20         try
21         {
22            asyncRequest = new XMLHttpRequest(); // create request object
23
24            // register event handler
25            asyncRequest.addEventListener(
26               "readystatechange", processResponse, false);
27            asyncRequest.open( "GET", url, true ); // prepare the request
28            asyncRequest.send( null ); // send the request
29         } // end try
30         catch ( exception )
31         {
32            alert( 'Request Failed' );
33         } // end catch
34      } // end function getImages
35
```

Fig. 16.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 2 of 8.)

```
36      // parses the XML response; dynamically creates an undordered list and
37      // populates it with the response data; displays the list on the page
38      function processResponse()
39      {
40         // if request completed successfully and responseXML is non-null
41         if ( asyncRequest.readyState == 4 && asyncRequest.status == 200 &&
42            asyncRequest.responseXML )
43         {
44            clearImages(); // prepare to display a new set of images
45
46            // get the covers from the responseXML
47            var covers = asyncRequest.responseXML.getElementsByTagName(
48               "cover" )
49
50            // get base URL for the images
51            var baseUrl = asyncRequest.responseXML.getElementsByTagName(
52               "baseurl" ).item( 0 ).firstChild.nodeValue;
53
54            // get the placeholder div element named covers
55            var output = document.getElementById( "covers" );
56
57            // create an unordered list to display the images
58            var imagesUL = document.createElement( "ul" );
```

Fig. 16.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 3 of 8.)

```
59
60          // place images in unordered list
61          for ( var i = 0; i < covers.length; ++i )
62          {
63             var cover = covers.item( i ); // get a cover from covers array
64
65             // get the image filename
66             var image = cover.getElementsByTagName( "image" ).
67                item( 0 ).firstChild.nodeValue;
68
69             // create li and img element to display the image
70             var imageLI = document.createElement( "li" );
71             var imageTag = document.createElement( "img" );
72
73             // set img element's src attribute
74             imageTag.setAttribute( "src", baseUrl + escape( image ) );
75             imageLI.appendChild( imageTag ); // place img in li
76             imagesUL.appendChild( imageLI ); // place li in ul
77          } // end for statement
78
79          output.appendChild( imagesUL ); // append ul to covers div
80       } // end if
81    } // end function processResponse
```

Fig. 16.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 4 of 8.)

```
82
83      // clears the covers div
84      function clearImages()
85      {
86         document.getElementById( "covers" ).innerHTML = "";
87      } // end function clearImages
88
89      // register event listeners
90      function registerListeners()
91      {
92         document.getElementById( "all" ).addEventListener(
93            "click", function() { getImages( "all.xml" ); }, false );
94         document.getElementById( "simply" ).addEventListener(
95            "click", function() { getImages( "simply.xml" ); }, false );
96         document.getElementById( "howto" ).addEventListener(
97            "click", function() { getImages( "howto.xml" ); }, false );
98         document.getElementById( "dotnet" ).addEventListener(
99            "click", function() { getImages( "dotnet.xml" ); }, false );
100        document.getElementById( "javaccpp" ).addEventListener(
101           "click", function() { getImages( "javaccpp.xml" ); }, false );
102        document.getElementById( "none" ).addEventListener(
103           "click", clearImages, false );
104     } // end function registerListeners
```

Fig. 16.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 5 of 8.)

```
105
106     window.addEventListener( "load", registerListeners, false );
107     </script>
108  </head>
109  <body>
110     <input type = "radio" name ="Books" value = "all"
111        id = "all"> All Books
112     <input type = "radio" name = "Books" value = "simply"
113        id = "simply"> Simply Books
114     <input type = "radio" name = "Books" value = "howto"
115        id = "howto"> How to Program Books
116     <input type = "radio" name = "Books" value = "dotnet"
117        id = "dotnet"> .NET Books
118     <input type = "radio" name = "Books" value = "javaccpp"
119        id = "javaccpp"> Java/C/C++ Books
120     <input type = "radio" checked name = "Books" value = "none"
121        id = "none"> None
122     <div id = "covers"></div>
123  </body>
124  </html>
```

Fig. 16.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 6 of 8.)

a) User clicks the **All Books** radio button to display all the book covers. The application sends an *asynchronous* request to the server to obtain an XML document containing the list of book-cover filenames. When the response is received, the application performs a *partial page update* to display the set of book covers.

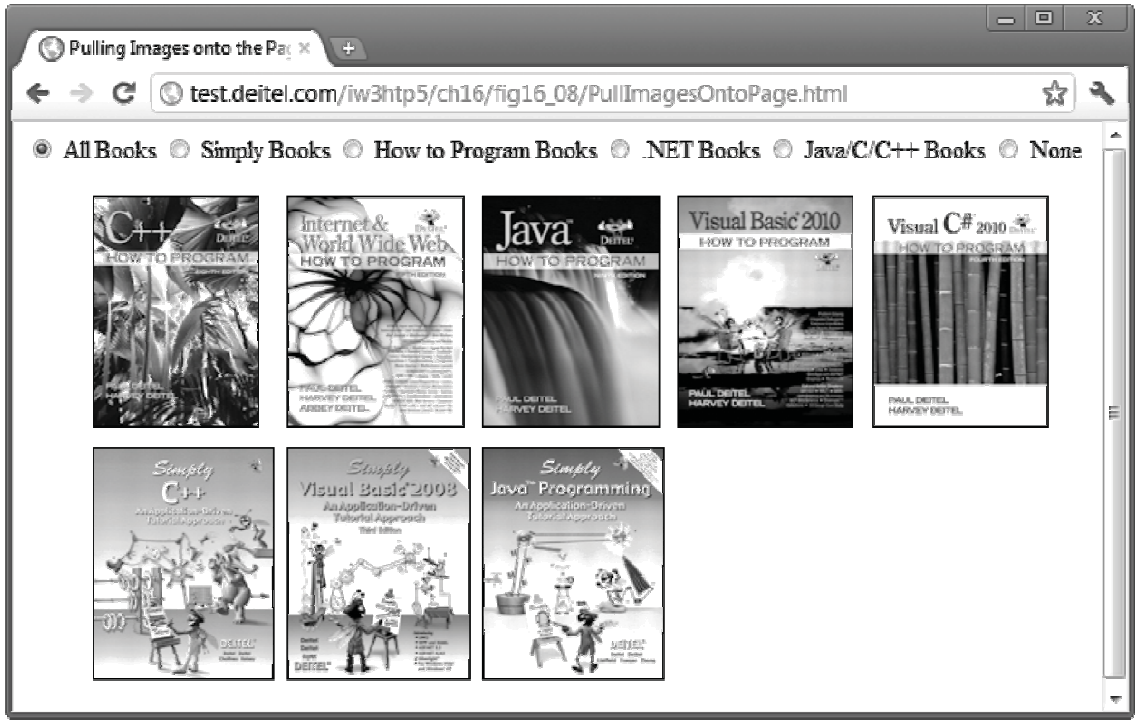


Fig. 16.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 7 of 8.)

b) User clicks the **How to Program Books** radio button to select a subset of book covers to display. Application sends an *asynchronous* request to the server to obtain an XML document containing the appropriate subset of book-cover filenames. When the response is received, the application performs a *partial page update* to display the subset of book covers.

Fig. 16.8 | Image catalog that uses Ajax to request XML data asynchronously. (Part 8 of 8.)

http://test.deitel.com/iw3htp5/ch16/fig16_08/PullImagesOntoPage.html

16.6 Creating a Full-Scale Ajax-Enabled Application

- JSON (JavaScript Object Notation)
  - Simple way to represent JavaScript objects as strings
  - A simpler alternative to XML for passing data between the client and the server
- JSON object
  - Represented as a list of property names and values contained in curly braces
- Array
  - Represented in JSON with square brackets containing a comma-separated list of values
  - Each value in a JSON array can be a string, a number, a JSON representation of an object, `true`, `false` or `null`
- JSON strings
  - Easier to create and parse than XML
  - Require fewer bytes
  - For these reasons, JSON is commonly used to communicate in client/server interaction

16.6.2 Rich Functionality

- The previous examples in this chapter requested data from files on the server.
- The next example is an address-book application that communicates with a server-side web service.
- The application uses server-side processing to give the page the functionality and usability of a desktop application.
- We use JSON to encode server-side responses and to create objects on the fly.
- Each time the user clicks a name, the address book uses Ajax functionality to load the person's address from the server and expand the entry *without reloading the page*—and it does this *in parallel* with allowing the user to click other names.

```
<!DOCTYPE html>

<!-- Fig. 16.9 addressbook.html -->
<!-- Ajax enabled address book application. -->
<html>
<head>
   <meta charset="utf-8">
   <title>Address Book</title>
   <link rel = "stylesheet" type = "text/css" href = "style.css">
   <script src = "AddressBook.js"></script>
</head>
<body>
   <div>
      <input id = "addressBookButton" type = "button"
         value = "Address Book">
      <input id = "addEntryButton" type = "button"
         value = "Add an Entry">
   </div>
   <div id = "addressBook"">
      <p>Search By Last Name: <input id = "searchInput"></p>
      <div id = "Names"></div>
   </div>
```

Fig. 16.9 | Ajax-enabled address-book application. (Part 1 of 7.)

```
23      <div id = "addEntry" style = "display : none">
24         <p><label>First Name:</label> <input id = "first"></p>
25         <p><label>Last Name:</label> <input id = "last"></p>
26         <p class = "head">Address:</p>
27         <p><label>Street:</label> <input id = "street"></p>
28         <p><label>City:</label> <span id = "city" class = "validator">
29            </span></p>
30         <p><label>State:</label> <span id = "state" class = "validator">
31            </span></p>
32         <p><label>Zip:</label> <input id = "zip">
33            <span id = "validateZip" class = "validator"></span></p>
34         <p><label>Telephone:</label> <input id = "phone">
35            <span id = "validatePhone" class = "validator"></span></p>
36         <p><input id = "submitButton" type = "button" value = "Submit"></p>
37         <div id = "success" class = "validator"></div>
38      </div>
39   </body>
40   </html>
```

Fig. 16.9 | Ajax-enabled address-book application. (Part 2 of 7.)

a) Page is loaded. All the entries are displayed.

b) User clicks on an entry. The entry expands,

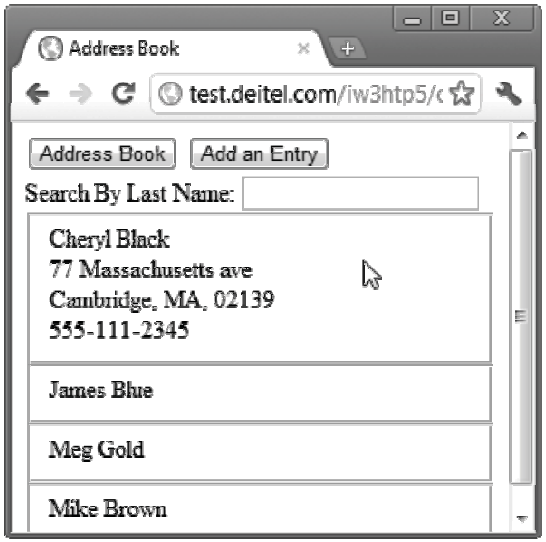


Fig. 16.9 | Ajax-enabled address-book application. (Part 3 of 7.)

c) User types "B" in the search field. Application loads the entries whose last names start with "B".

d) User types "Bl" in the search field. Application loads the entries whose last names start with "Bl".

Fig. 16.9 | Ajax-enabled address-book application. (Part 4 of 7.)

e) User types "Bla" in the search field. Application loads the entries whose last names start with "Bla".

f) User clicks **Add an Entry** button. The form allowing user to add an entry is displayed.

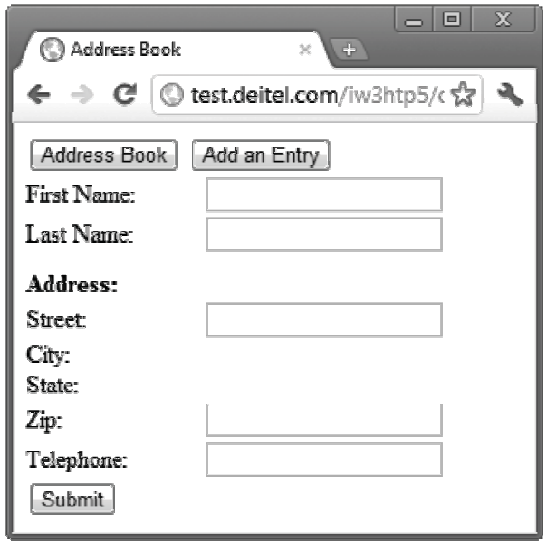


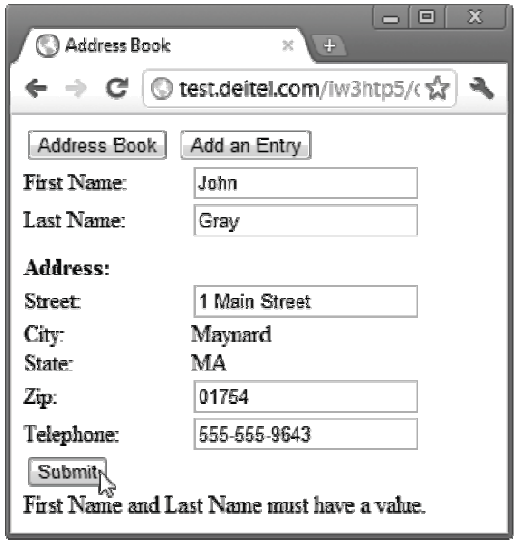
Fig. 16.9 | Ajax-enabled address-book application. (Part 5 of 7.)

g) User enters a valid ZIP code, then tabs to the next field. The server finds the city and state associated with the ZIP code entered and displays them on the page.

h) The user enters a telephone number and tries to submit the data. The application does not allow this, because the First Name and Last Name are empty.

Fig. 16.9 | Ajax-enabled address-book application. (Part 6 of 7.)

i) The user enters the last name and the first name and clicks the **Submit** button.



j) The address book is redisplayed with the new name added in.

Fig. 16.9 | Ajax-enabled address-book application. (Part 7 of 7.)

http://test.deitel.com/iw3htp5/ch16/fig16_09-10/AddressBook.html

```
1  // Fig. 16.10 addressbook.js
2  // Ajax-enabled address-book JavaScript code
3  // URL of the web service
4  var webServiceUrl = "/AddressBookWebService/Service.svc";
5
6  var phoneValid = false; // indicates if the telephone is valid
7  var zipValid = false; //indicates if the ZIP code is valid
8
9  // get a list of names from the server and display them
10 function showAddressBook()
11 {
12    // hide the "addEntry" form and show the address book
13    document.getElementById( "addEntry" ).style.display = "none";
14    document.getElementById( "addressBook" ).style.display = "block";
15
16    callWebService( "/getAllNames", parseData );
17 } // end function showAddressBook
18
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 1 of 15.)

```
19  // send the asynchronous request to the web service
20  function callWebService( methodAndArguments, callBack )
21  {
22     // build request URL string
23     var requestUrl = webServiceUrl + methodAndArguments;
24
25     // attempt to send the asynchronous request
26     try
27     {
28        var asyncRequest = new XMLHttpRequest(); // create request
29
30        // set up callback function and store it
31        asyncRequest.addEventListener("readystatechange",
32           function() { callBack( asyncRequest ); }, false);
33
34        // send the asynchronous request
35        asyncRequest.open( "GET", requestUrl, true );
36        asyncRequest.setRequestHeader("Accept",
37           "application/json; charset=utf-8" );
38        asyncRequest.send(); // send request
39     } // end try
40     catch ( exception )
41     {
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 2 of 15.)

```
42          alert ( "Request Failed" );
43       } // end catch
44    } // end function callWebService
45
46    // parse JSON data and display it on the page
47    function parseData( asyncRequest )
48    {
49       // if request has completed successfully, process the response
50       if ( asyncRequest.readyState == 4 && asyncRequest.status == 200 )
51       {
52          // convert the JSON string to an Object
53          var data = JSON.parse(asyncRequest.responseText);
54          displayNames( data ); // display data on the page
55       } // end if
56    } // end function parseData
57
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 3 of 15.)

```
58  // use the DOM to display the retrieved address-book entries
59  function displayNames( data )
60  {
61     // get the placeholder element from the page
62     var listBox = document.getElementById( "Names" );
63     listBox.innerHTML = ""; // clear the names on the page
64
65     // iterate over retrieved entries and display them on the page
66     for ( var i = 0; i < data.length; ++i )
67     {
68        // dynamically create a div element for each entry
69        // and a fieldset element to place it in
70        var entry = document.createElement( "div" );
71        var field = document.createElement( "fieldset" );
72        entry.onclick = function() { getAddress( this, this.innerHTML ); };
73        entry.id = i; // set the id
74        entry.innerHTML = data[ i ].First + " " + data[ i ].Last;
75        field.appendChild( entry ); // insert entry into the field
76        listBox.appendChild( field ); // display the field
77     } // end for
78  } // end function displayAll
79
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 4 of 15.)

```
80  // search the address book for input
81  // and display the results on the page
82  function search( input )
83  {
84     // get the placeholder element and delete its content
85     var listBox = document.getElementById( "Names" );
86     listBox.innerHTML = ""; // clear the display box
87
88     // if no search string is specified, all the names are displayed
89     if ( input == "" ) // if no search value specified
90     {
91        showAddressBook(); // Load the entire address book
92     } // end if
93     else
94     {
95        callWebService( "/search/" + input, parseData );
96     } // end else
97  } // end function search
98
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 5 of 15.)

```
99  // Get address data for a specific entry
100 function getAddress( entry, name )
101 {
102    // find the address in the JSON data using the element's id
103    // and display it on the page
104    var firstLast = name.split(" "); // convert string to array
105    var requestUrl = webServiceUrl + "/getAddress/"
106       + firstLast[ 0 ] + "/" + firstLast[ 1 ];
107
108    // attempt to send an asynchronous request
109    try
110    {
111       // create request object
112       var asyncRequest = new XMLHttpRequest();
113
114       // create a callback function with 2 parameters
115       asyncRequest.addEventListener("readystatechange",
116          function() { displayAddress( entry, asyncRequest ); }, false);
117
118       asyncRequest.open( "GET", requestUrl, true );
119       asyncRequest.setRequestHeader("Accept",
120          "application/json; charset=utf-8"); // set response datatype
121       asyncRequest.send(); // send request
122    } // end try
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 6 of 15.)

```
123     catch ( exception )
124     {
125        alert ( "Request Failed." );
126     } // end catch
127  } // end function getAddress
128
129  // clear the entry's data
130  function displayAddress( entry, asyncRequest )
131  {
132     // if request has completed successfully, process the response
133     if ( asyncRequest.readyState == 4 && asyncRequest.status == 200 )
134     {
135        // convert the JSON string to an object
136        var data = JSON.parse(asyncRequest.responseText);
137        var name = entry.innerHTML // save the name string
138        entry.innerHTML = name + "<br>" + data.Street +
139           "<br>" + data.City + ", " + data.State
140           + ", " + data.Zip + "<br>" + data.Telephone;
141
142        // change event listener
143        entry.onclick = function() { clearField( entry, name ); };
144     } // end if
145  } // end function displayAddress
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 7 of 15.)

```
146
147 // clear the entry's data
148 function clearField( entry, name )
149 {
150    entry.innerHTML = name; // set the entry to display only the name
151    entry.onclick = function() { getAddress( entry, name ); };
152 } // end function clearField
153
154 // display the form that allows the user to enter more data
155 function addEntry()
156 {
157    document.getElementById( "addressBook" ).style.display = "none";
158    document.getElementById( "addEntry" ).style.display = "block";
159 } // end function addEntry
160
161 // send the ZIP code to be validated and to generate city and state
162 function validateZip( zip )
163 {
164    callWebService ( "/validateZip/" + zip, showCityState );
165 } // end function validateZip
166
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 8 of 15.)

```
167 // get city and state that were generated using the zip code
168 // and display them on the page
169 function showCityState( asyncRequest )
170 {
171    // display message while request is being processed
172    document.getElementById( "validateZip" ).
173       innerHTML = "Checking zip...";
174
175    // if request has completed successfully, process the response
176    if ( asyncRequest.readyState == 4 )
177    {
178       if ( asyncRequest.status == 200 )
179       {
180          // convert the JSON string to an object
181          var data = JSON.parse(asyncRequest.responseText);
182
183          // update ZIP-code validity tracker and show city and state
184          if ( data.Validity == "Valid" )
185          {
186             zipValid = true; // update validity tracker
187
188             // display city and state
189             document.getElementById( "validateZip" ).innerHTML = "";
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 9 of 15.)

```
190             document.getElementById( "city" ).innerHTML = data.City;
191             document.getElementById( "state" ).
192                innerHTML = data.State;
193          } // end if
194          else
195          {
196             zipValid = false; // update validity tracker
197             document.getElementById( "validateZip" ).
198                innerHTML = data.ErrorText; // display the error
199
200             // clear city and state values if they exist
201             document.getElementById( "city" ).innerHTML = "";
202             document.getElementById( "state" ).innerHTML = "";
203          } // end else
204       } // end if
205       else if ( asyncRequest.status == 500 )
206       {
207          document.getElementById( "validateZip" ).
208             innerHTML = "Zip validation service not avaliable";
209       } // end else if
210    } // end if
211 } // end function showCityState
212
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 10 of 15.)

```
213 // send the telephone number to the server to validate format
214 function validatePhone( phone )
215 {
216    callWebService( "/validateTel/" + phone, showPhoneError );
217 } // end function validatePhone
218
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 11 of 15.)

```
219 // show whether the telephone number has correct format
220 function showPhoneError( asyncRequest )
221 {
222    // if request has completed successfully, process the response
223    if ( asyncRequest.readyState == 4 && asyncRequest.status == 200 )
224    {
225       // convert the JSON string to an object
226       var data = JSON.parse(asyncRequest.responseText);
227
228       if ( data.ErrorText != "Valid Telephone Format" )
229       {
230          phoneValid = false; // update validity tracker
231          document.getElementById( "validatePhone" ).innerHTML =
232             data.ErrorText; // display the error
233       } // end if
234       else
235       {
236          phoneValid = true; // update validity tracker
237       } // end else
238    } // end if
239 } // end function showPhoneError
240
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 12 of 15.)

```
241 // enter the user's data into the database
242 function saveForm()
243 {
244    // retrieve the data from the form
245    var first = document.getElementById( "first" ).value;
246    var last = document.getElementById( "last" ).value;
247    var street = document.getElementById( "street" ).value;
248    var city = document.getElementById( "city" ).innerHTML;
249    var state = document.getElementById( "state" ).innerHTML;
250    var zip = document.getElementById( "zip" ).value;
251    var phone = document.getElementById( "phone" ).value;
252
253    // check if data is valid
254    if ( !zipValid || !phoneValid  )
255    {
256       // display error message
257       document.getElementById( "success" ).innerHTML =
258          "Invalid data entered. Check form for more information";
259    } // end if
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 13 of 15.)

```
260     else if ( ( first == "" ) || ( last == "" ) )
261     {
262        // display error message
263        document.getElementById("success").innerHTML =
264           "First Name and Last Name must have a value.";
265     } // end if
266     else
267     {
268        // hide the form and show the address book
269        document.getElementById( "addEntry" ).style.display = "none";
270        document.getElementById( "addressBook" ).style.display = "block";
271
272        // call the web service to insert data into the database
273        callWebService( "/addEntry/" + first + "/" + last + "/" + street +
274           "/" + city + "/" + state + "/" + zip + "/" + phone, parseData );
275     } // end else
276  } // end function saveForm
277
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 14 of 15.)

```
278 // register event listeners
279 function start()
280 {
281    document.getElementById( "addressBookButton" ).addEventListener(
282       "click", showAddressBook, false );
283    document.getElementById( "addEntryButton" ).addEventListener(
284       "click", addEntry, false );
285    document.getElementById( "searchInput" ).addEventListener(
286       "keyup", function() { search( this.value ); } , false );
287    document.getElementById( "zip" ).addEventListener(
288       "blur", function() { validateZip( this.value ); } , false );
289    document.getElementById( "phone" ).addEventListener(
290       "blur", function() { validatePhone( this.value ); } , false );
291    document.getElementById( "submitButton" ).addEventListener(
292       "click", saveForm , false );
293
294    showAddressBook();
295 } // end function start
296
297 window.addEventListener( "load", start, false );
```

Fig. 16.10 | JavaScript code for the address-book application. (Part 15 of 15.)

16.6.3 Interacting with a Web Service on the Server

- An ASP.NET REST web service created for this example does the server-side processing.
- The web service contains a collection of methods, including `getAllNames`, that can be called from a web application.

16.6.4 Parsing Json Data

- Each of our web service's methods in this example returns a JSON representation of an object or array of objects.
- For example, when the web application requests the list of names in the address book, the list is returned as a JSON array.
- Each object in has the attributes `first` and `last`.

```
[ { "first": "Cheryl", "last": "Black" },
  { "first": "James", "last": "Blue" },
  { "first": "Mike", "last": "Brown" },
  { "first": "Meg", "last": "Gold" } ]
```

Fig. 16.11 | Address-book data formatted in JSON.

16.6.5 Creating HTML5 Elements and Setting Event Handlers on the Fly

- To determine which address the user clicked, we introduce the `this` keyword.
- In an event-handling function, `this` refers to the DOM object on which the event occurred.
- Our function uses `this` to refer to the clicked entry.
- The `this` keyword allows us to use one event handler to apply a change to one of many DOM elements, depending on which one received the event.

16.6.6 Implementing Type-Ahead

- To implement type-ahead
  - Can use an element's keyup event handler to make asynchronous requests

16.6.6 Implementing a Form with Asynchronous Validation

- The `addEntry div` in the HTML5 document contains a set of entry fields, some of which have event handlers (registered in the JavaScript start function) that enable validation that occurs *asynchronously* as the user continues to interact with the page.
- When a user enters a ZIP code, then moves the cursor to another field, the `validateZip` function is called.
  - calls an external web service to validate the ZIP code.
  - If it's valid, that external web service returns the corresponding city and state.