1

AJAX Technologies

Traditional Web pages use server-side technologies and resources to operate and deliver their features and services to end users. These Web pages require end users to perform full-page postbacks to the server, where these pages can run the required server-side code to deliver the requested service or feature. In other words, these Web pages use the click-and-wait, user-unfriendly interaction pattern, which is characterized by waiting periods that disrupt user workflow and degrade the user experience. This click-and-wait user interaction pattern is what makes the traditional Web applications act and feel very different from their desktop counterparts.

Asynchronous JavaScript And XML (abbreviated *AJAX*) is a popular Web application development approach that uses client-side technologies such as HTML, XHTML, CSS, DOM, XML, XSLT, Javascript, and asynchronous client-callback techniques such as `XMLHttp` requests and hidden-frame techniques to develop more sophisticated and responsive Web applications that break free from the click-and-wait pattern and, consequently, act and feel more like a desktop application. In other words, AJAX is closing the gap between Web applications and their desktop counterparts.

This chapter begins by discussing the main characteristics of AJAX-enabled Web pages in the context of an example.

Google Suggest

The Google Suggest Web page (`www.google.com/webhp?complete=1`) contains an AJAX-enabled search box that completes your search items as you type them in, as shown in Figure 1-1. Under the hood, this AJAX-enabled search box uses AJAX techniques to asynchronously download the required data from the Web server and to display them to the end user without interrupting the user's interaction with the page. All the client-server communications are performed in the background as the end user types into the search box.

An AJAX-enabled component such as the Google Suggest search box exhibits the following four important characteristics:

- It uses HTML, XHTML, CSS, DOM, and JavaScript client-side technologies to implement most of its functionalities where the code runs locally on the client machine to achieve the

Figure 1-1

same response time as its desktop counterpart. This allows an AJAX-enabled component to break free from the click-and-wait user-interaction pattern.

- It uses asynchronous client-callback techniques such as `XMLHttpRequest` to communicate with the server. The main goal of this asynchronous communication model is to ensure that the communication with the server doesn't interrupt what the user is doing. This asynchronous communication model is another step that allows an AJAX-enabled component to break free from the click-and-wait pattern.
- AJAX-enabled components normally send data to and receive data from the server in either XML or JSON format (discussed in detail later in this chapter). This characteristic enables the client-side code to exchange data with any type of server-side code, and vice versa, because almost all platforms have built-in support for reading, writing, and manipulating XML or JSON data.
- The asynchronous communication between the client-side code and the server-side code are normally governed by AJAX communication patterns. These patterns enable AJAX components to take full advantage of the asynchronous nature of the communication between the client-side code and the server-side code to determine the best time for uploading the data to or downloading the data from the server so the data exchange with the server won't interrupt the user workflow and degrade the user experience.

In a traditional Web page, the end users trigger synchronous communications with the Web server, and they then have to wait until the required data is downloaded from the server and the entire page is

rendered all over again to display the new information. AJAX changes all that. As you can see in Figure 1-2, the Ajax engine takes complete control over the client-server communications and the rendering of the new information to ensure that these communications and renderings do not interrupt the user interactions.

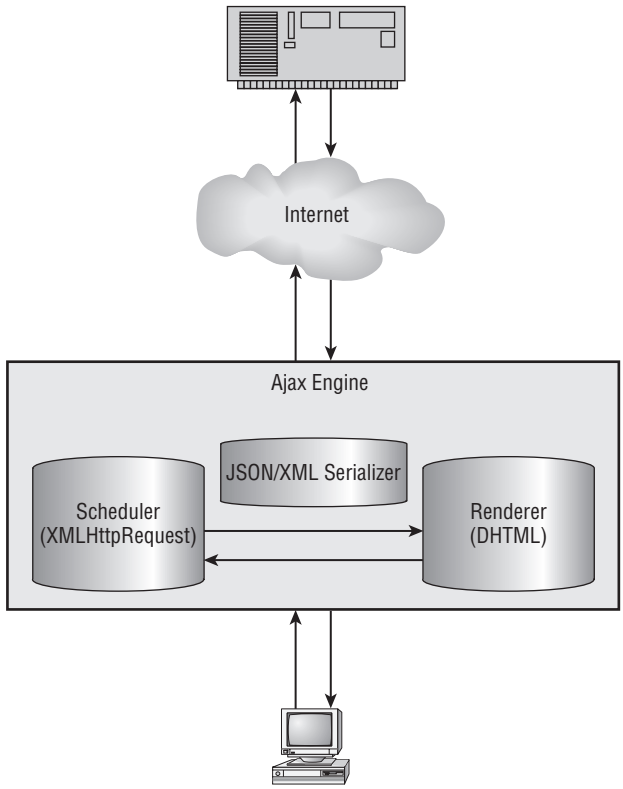

Figure 1-2

As this figure shows, the AJAX engine consists of the following three main components:

- **Scheduler:** The scheduler uses AJAX technologies such as `XMLHttpRequest` to send data to and receive data from the server in an asynchronous fashion. As the name suggests, the scheduler schedules and makes the client requests to the server.
- **Renderer:** The renderer component of the AJAX engine uses DHTML to dynamically update only those portions of the current page that need refreshing without re-rendering or re-loading the entire page.
- **JSON/XML Serializer:** The client and server exchange data in JSON or XML format. The JSON/XML serializer has two main responsibilities:
  - Serialize the client data, which are JavaScript objects, into their JSON or XML representations before these objects are sent to the server
  - Deserialize JavaScript objects from the JSON or XML data received from the server

This chapter provides an overview of the following client-side technologies that form the foundations of the above three main AJAX engine components in the context of an example:

- `XMLHttpRequest`
- DHTML
- XML
- JSON

XMLHttpRequest

`XMLHttpRequest` is one of the main AJAX technologies that the scheduler component of an AJAX engine uses to make asynchronous requests to the server. The instantiation process of the `XMLHttpRequest` object is browser-dependent. Listing 1-1 encapsulates the browser-dependent nature of this instantiation process in a class named `XMLHttpRequest`.

Listing 1-1: Instantiating XMLHttpRequest

```
if (!window.XMLHttpRequest)
{
  window.XMLHttpRequest = function window$XMLHttpRequest()
  {
    var progIDs = [ 'Msxml2.XMLHTTP', 'Microsoft.XMLHTTP' ];
    for (var i = 0; i < progIDs.length; i++)
    {
      try
      {
        var xmlHttp = new ActiveXObject(progIDs[i]);
        return xmlHttp;
      }
      catch (ex) {}
    }
    return null;
  }
}
```

This script first checks whether the `window` object already contains a definition for this class. If not, it defines the constructor of the class. The constructor contains the following array of program ids:

```
var progIDs = [ 'Msxml2.XMLHTTP', 'Microsoft.XMLHTTP' ];
```

This array covers all the possible instantiation scenarios on Internet Explorer. The constructor iterates through the program ids array and takes the following steps for each enumerated program id:

1. It instantiates an `ActiveXObject`, passing in the enumerated program id.
2. If the instantiation succeeds, it returns this `ActiveXObject` instance.
3. If the instantiation fails, the `try` block throws an exception, which the `catch` block catches and forces the loop to move to the next iteration, where the next program id is used.

The XMLHttpRequest object exposes the following methods and properties:

- open: This method takes up to five parameters, but only the first two parameters are required. The first required parameter is a string that contains the HTTP verb (POST or GET) being used to make the request to the server. The second required parameter is a string that contains the target URL, which is the URL of the resource for which the request is made. The third optional parameter is a Boolean value that specifies whether the request is asynchronous. If you don't specify a value for this parameter, it defaults to true. The fourth and fifth optional parameters specify the requester's credentials — the username and password.
- readyState: The XMLHttpRequest exposes an integer property named readyState with possible values of 0, 1, 2, 3, or 4. The XMLHttpRequest goes through different states during its lifecycle, and each state is associated with one of these five possible values.
- onreadystatechange: You must assign a reference to a JavaScript function to this property. The XMLHttpRequest invokes this JavaScript function every time its state changes, which is every time its readyState property changes value. Every time your JavaScript function is invoked, it must check the value of the XMLHttpRequest's readyState property to determine the state of the XMLHttpRequest. The current request is completed only when XMLHttpRequest enters the state associated with the readyState property value of 4. As a result, the typical implementation of the JavaScript function assigned to the onreadystatechange property is as follows:

```
function readyStateChangeCallback()
{
  if (request.readyState == 4 && request.status == 200)
  {
    // Process the server response here
  }
}
```

  The global variable named request in this code fragment references the XMLHttpRequest object. This JavaScript function checks whether the readyState property of the XMLHttpRequest is 4, meaning the request is completed. If so, it processes the server response. If not, it simply returns.

- status: This property contains the HTTP status code of the server response. The JavaScript function that you assign to the onreadystatechange property must also check whether the status property of the XMLHttpRequest is 200, as shown in the boldface portion of the following code fragment. If the status code is a not 200, this is an indication that a server-side error has occurred.

```
function readyStateChangeCallback()
{
  if (request.readyState == 4 && request.status == 200)
  {
    // Process the server response here
  }
}
```

Strictly speaking, any status code within the 200–299 range is considered a success. However, a status code of 200 is good enough in this case.

- `statusText`: This property contains the HTTP status text of the server response. The text describes the HTTP status code. For example, the status text for status code `200` is `OK`.
- `setRequestHeader`: This method sets a specified HTTP request header to a specified value. As such, this method takes two parameters: the first parameter is a string that contains the name of the HTTP request header whose value is being set, and the second parameter is a string that contains the value of this HTTP request header.
- `send`: This is the method that actually sends the request to the server. It takes a string parameter that contains the request body. If you're making a `GET` HTTP request, pass `null` as the value of this parameter. If you're making a `POST` HTTP request, generate a string that contains the body of the request and pass this string into the `send` method.
- `responseText`: This property contains the server response in text format.
- `responseXML`: This property contains the server response in XML format (an XML `Document` to be exact). This property is set only when the `Content-Type` response header is set to the value `text/xml`. If the server-side code does not set the response header to this value, the `responseXML` property will be `null` even when the actual data is in XML format. In such cases, you must load the content of the `responseText` property into an XML document before you can use the client-side XML API to read the XML data.

The `overrideMimeType` *property of* `XMLHttpRequest` *in Mozilla browsers enables you to override the MIME type of the server response. However, this is a browser-specific issue that the current discussion does not need to address.*

- `getResponseHeader`: This method returns the value of a response header with a specified name. As such, it takes the name of the response header as its only argument.
- `getAllResponseHeaders`: This method returns the names and values of all response headers.
- `abort`: Use this method to abort a request.

Listing 1-2 presents an example that uses `XMLHttpRequest` to make an asynchronous request to the server. If you access this page, you see the result shown in Figure 1-3. This page consists of a simple user interface with two text boxes and a button. If you enter the text "**username**" in the top text box and the text "**password**" in the bottom text box and then click the button, you get the result shown in Figure 1-4.

Listing 1-2: A page that uses XMLHttpRequest

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
  void Page_Load(object sender, EventArgs e)
  {
    if (Request.Headers["MyCustomHeader"] != null)
    {
      if (Request.Form["passwordtbx"] == "password" &&
          Request.Form["usernametbx"] == "username")
      {
        Response.Write("Shahram|Khosravi|22223333|Some Department|");
        Response.End();
      }
      else
```

```
            throw new Exception("Wrong credentials");
        }
      }
    </script>
    <html xmlns="http://www.w3.org/1999/xhtml">
    <head id="Head1" runat="server">
      <title>Untitled Page</title>
      <script type="text/javascript" language="javascript">
      var request;

      if (!window.XMLHttpRequest)
      {
        window.XMLHttpRequest = function window$XMLHttpRequest()
        {
          var progIDs = [ 'Msxml2.XMLHTTP', 'Microsoft.XMLHTTP' ];

          for (var i = 0; i < progIDs.length; i++)
          {
            try
            {
              var xmlHttp = new ActiveXObject(progIDs[i]);
              return xmlHttp;
            }
            catch (ex) {}
          }

          return null;
        }
      }

      window.employee = function window$employee(firstname, lastname,
                                                 employeeid, departmentname)
      {
        this.firstname = firstname;
        this.lastname = lastname;
        this.employeeid = employeeid;
        this.departmentname = departmentname
      }

      function deserialize()
      {
        var delimiter="|";
        var responseIndex = 0;
        var delimiterIndex;
        var response = request.responseText;

        delimiterIndex = response.indexOf(delimiter, responseIndex);
        var firstname = response.substring(responseIndex, delimiterIndex);
        responseIndex = delimiterIndex + 1;
        delimiterIndex = response.indexOf(delimiter, responseIndex);
        var lastname = response.substring(responseIndex, delimiterIndex);
        responseIndex = delimiterIndex + 1;

        delimiterIndex = response.indexOf(delimiter, responseIndex);
```

(continued)

Listing 1-2 *(continued)*

```
    var employeeid = response.substring(responseIndex, delimiterIndex);
    responseIndex = delimiterIndex + 1;

    delimiterIndex = response.indexOf(delimiter, responseIndex);
    var departmentname = response.substring(responseIndex, delimiterIndex);

    return new employee(firstname, lastname, employeeid, departmentname);
  }

  function readyStateChangeCallback()
  {
    if (request.readyState == 4 && request.status == 200)
    {
      var credentials = document.getElementById("credentials");
      credentials.style.display="none";
      var employeeinfotable = document.getElementById("employeeinfo");
      employeeinfotable.style.display="block";

      var employee = deserialize();

      var firstnamespan = document.getElementById("firstname");
      firstnamespan.innerText = employee.firstname;
      var lastnamespan = document.getElementById("lastname");
      lastnamespan.innerText = employee.lastname;

      var employeeidspan = document.getElementById("employeeid");
      employeeidspan.innerText = employee.employeeid;

      var departmentnamespan = document.getElementById("departmentname");
      departmentnamespan.innerText = employee.departmentname;
    }
  }

  window.credentials = function window$credentials(username, password)
  {
    this.username = username;
    this.password = password;
  }

  function serialize(credentials)
  {
    var requestBody="";
    requestBody += "usernametbx";
    requestBody += "=";
    requestBody += encodeURIComponent(credentials.username);
    requestBody += "&";
    requestBody += "passwordtbx";
    requestBody += "=";
    requestBody += encodeURIComponent(credentials.password);
    return requestBody;
  }

  function submitCallback()
```

```
    {
      var usernametbx = document.getElementById("usernametbx");
      var passwordtbx = document.getElementById("passwordtbx");
      var credentials1= new credentials(usernametbx.value, passwordtbx.value);
      var body = serialize(credentials1);

      request = new XMLHttpRequest();
      request.open("POST", document.form1.action);
      request.onreadystatechange = readyStateChangeCallback;
      request.setRequestHeader("MyCustomHeader", "true");
      request.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
      request.send(body);
    }
    </script>
  </head>
  <body>
    <form id="form1" runat="server">
      <table id="credentials">
        <tr>
          <td align="right" style="font-weight: bold">
            Username:
          </td>
          <td align="left">
            <asp:TextBox runat="server" ID="usernametbx" /></td>
        </tr>
        <tr>
          <td align="right" style="font-weight: bold">
            Password:
          </td>
          <td align="left">
            <asp:TextBox runat="server" ID="passwordtbx"
            TextMode="Password" />
          </td>
        </tr>
        <tr>
          <td align="center" colspan="2">
            <button id="Button1" type="button"
            onclick="submitCallback()">Submit</button>
          </td>
        </tr>
      </table>
      <table id="employeeinfo"
      style="background-color: LightGoldenrodYellow;
             border-color: Tan; border-width: 1px;
             color: Black; display: none" cellpadding="2">
        <tr style="background-color: Tan; font-weight: bold">
          <th colspan="2">
            Your Information</th>
        </tr>
        <tr>
          <td style="font-weight: bold">
            First Name</td>
          <td>
            <span id="firstname" />
          </td>
```

(continued)

Listing 1-2 *(continued)*

```
        </tr>
        <tr style="background-color: PaleGoldenrod">
          <td style="font-weight: bold">
            Last Name</td>
          <td>
            <span id="lastname" />
          </td>
        </tr>
        <tr>
          <td style="font-weight: bold">
            Employee ID</td>
          <td>
            <span id="employeeid" />
          </td>
        </tr>
        <tr style="background-color: PaleGoldenrod">
          <td style="font-weight: bold">
            Department
          </td>
          <td>
            <span id="departmentname" />
          </td>
        </tr>
      </table>
    </form>
  </body>
  </html>
```

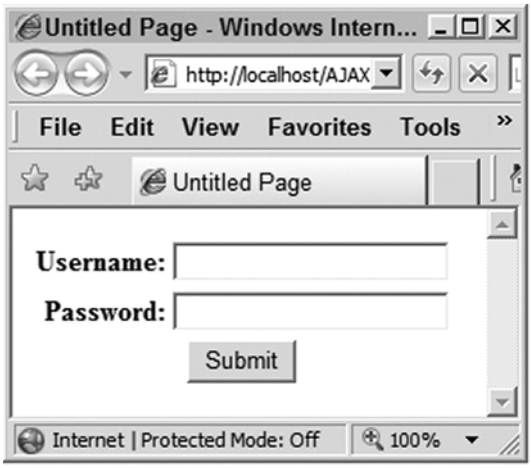

Figure 1-3

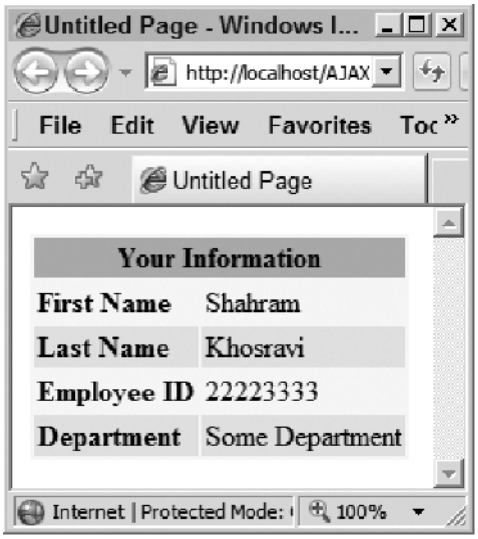

Figure 1-4

Note that Listing 1-2 registers a JavaScript function named `submitCallback` as an event handler for the `click` event of the button. This function encapsulates the logic that schedules and makes the asynchronous request to the server. This logic is what is referred to as the Scheduler in Figure 1-2.

Now let's walk through the `submitCallback` function in the listing. First, `submitCallback` calls the `getElementbyId` method on the `document` object to return a reference to the username text box DOM element:

```
        var usernametbx = document.getElementById("usernametbx");
```

Next, it calls the `getElementById` method again to return a reference to the password text box DOM element:

```
        var passwordtbx = document.getElementById("passwordtbx");
```

Next, it creates an instance of a class named `credentials`:

```
    var credentials1 = new credentials(usernametbx.value, passwordtbx.value);
```

Listing 1-2 defines the `credentials` class as follows:

```
      window.credentials = function window$credentials(username, password)
      {
        this.username = username;
        this.password = password;
      }
```

The next order of business is to serialize this `credentials` object into a format that the server-side code understands. This is exactly what the following JavaScript function named `serialize` does:

```
        var body = serialize(credentials1);
```

This function basically contains the logic referred to as the Serializer in Figure 1-2. The `serialize` function is discussed in more detail shortly, but for now it suffices to say that this function serializes the specified `credentials` object into a string with a specific format.

Next, the `submitCallback` function creates an instance of the `XMLHttpRequest` class previously defined in Listing 1-1:

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

As previously discussed, this class encapsulates the browser-dependent logic that instantiates the appropriate object.

Then, the `submitCallback` function invokes the `open` method on this `XMLHttpRequest` object, passing in two parameters. The first parameter is the string "`POST`" because the function is making a `POST` HTTP request to the server. The second parameter is the value of the `action` property of the `form` element. The `action` property contains the URL of the current page. The page is basically posting back to itself in asynchronous fashion.

```
        request.open("POST", document.form1.action);
```

Next, `submitCallback` assigns a reference, which references a JavaScript function named `readyStateChangeCallback`, to the `onreadystatechange` property of the `XMLHttpRequest` object:

```
        request.onreadystatechange = readyStateChangeCallback;
```

Then, it invokes the `setRequestHeader` method on the `XMLHttpRequest` object to add a custom header named `MyCustomHeader` with the value `true`:

```
        request.setRequestHeader("MyCustomHeader", "true");
```

As you'll see later, when the page finally posts back to itself, the server-side code uses this header to distinguish between ansynchronous and normal synchronous postback requests.

Next, the `submitCallback` function invokes the `setRequestHeader` method again, this time to set the value of the `Content-Type` header request to `application/x-www-form-urlencoded`:

```
        request.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
```

As you'll see later, this will allow you to use the `Request` object to access the posted data.

Finally, `submitCallback` invokes the `send` method on the `XMLHttpRequest` object, passing in the string that contains the post data to make an HTTP `POST` request to the server:

```
        request.send(body);
```

As previously discussed, this string is the return value of the `serialize` method.

Now let's walk through the implementation of the `serialize` function:

```
    function serialize(credentials)
    {
      var requestBody="";
      requestBody += "usernametbx";
      requestBody += "=";
      requestBody += encodeURIComponent(credentials.username);
      requestBody += "&";
      requestBody += "passwordtbx";
      requestBody += "=";
      requestBody += encodeURIComponent(credentials.password);
      return requestBody;
    }
```

The `serialize` function generates a string that consists of two substrings separated by the `&` character. The first substring itself consists of two substrings separated by the equal sign (`=`), where the first substring contains the `name` HTML attribute value of the username text box DOM element and the second substring contains the value that the end user has entered into this text box:

```
        var requestBody = "";
        requestBody += "usernametbx";
        requestBody += "=";
        requestBody += usernametbx.value;
```

The second substring itself consists of two substrings separated by the equal sign (=), where the first substring contains the `name` HTML attribute value of the password text box DOM element and the second substring contains the value that the end user has entered into this text box:

```
        requestBody += "passwordtbx";
        requestBody += "=";
        requestBody += passwordtbx.value;
```

When this HTTP `POST` request arrives at the server, ASP.NET automatically loads the body of the request into the `Request` object's `Form` collection property because the `Content-Type` request header is set to the value `application/x-www-form-urlencoded`. When the `Page_Load` method shown in Listing 1-2 is finally invoked, it first checks whether the current request contains an HTTP header named `MyCustomHeader`:

```
    if (Request.Headers["MyCustomHeader"] != null)
```

If so, this is an indication that the current page postback is an asynchronous page postback and, consequently, the `Page_Load` method first validates the user's credentials. To keep the current discussion focused, this method hardcodes the valid credentials as shown here:

```
          if (Request.Form["passwordtbx"] == "password" &&
              Request.Form["usernametbx"] == "username")
```

If the validation succeeds, `Page_Load` generates a string that contains the server data (which is again hardcoded to keep this discussion focused), invokes the `Write` method on the `Response` object to write this string into the response output stream, and invokes the `End` method on the `Response` object to end the current response and, consequently, to send the server response to the client:

```
            Response.Write("Shahram|Khosravi|22223333|Some Department|");
            Response.End();
```

Ending the current response ensures that the current page will not go through its normal rendering routine where it renders the entire page all over again. That is the reason behind adding the custom HTTP request header "`MyCustomHeader`".

The arrival of the server response changes the state of the `XMLHttpRequest` object to the `completed` state, which in turn changes the value of the `readyState` property of the object to `4`. This change in value automatically invokes the `readyStateChangeCallback` JavaScript function assigned to the `onreadystatechange` property of the object.

The `readyStateChangeCallback` JavaScript function encapsulates the logic that uses DHTML to dynamically update those portions of the page that need refereshing without re-rendering and reloading the entire page all over again. This logic is what is referred to as the Renderer in Figure 1-2.

The `readyStateChangeCallback` JavaScript function first checks whether the `readyState` and `status` properties of the XMLHttpRequest object are set to `4` and `200`, respectively. If so, it invokes the `getElementById` method on the document object to return a reference to the table DOM element that

displays the login dialog box, and sets the `display` property of this DOM element's `style` property to `none` to hide the dialog box:

```
        var credentials = document.getElementById("credentials");
        credentials.style.display="none";
```

Next, `readyStateChangeCallback` invokes the `getElementById` method again, this time to return a reference to the table DOM element that displays the server data, and sets the `display` property of this DOM element's `style` property to `block` to show this DOM element:

```
        var employeeinfotable = document.getElementById("employeeinfo");
        employeeinfotable.style.display="block";
```

Then, it invokes the `responseText` property on the `XMLHttpRequest` object to return a string that contains the server data:

```
        var response = request.responseText;
```

Keep in mind that the server data is in the following format:

```
Shahram|Khosravi|22223333|Some Department|
```

The next order of business is to deserialize an `employee` object from the server data. The following excerpt from Listing 1-2 defines the `employee` class:

```
    window.employee = function window$employee(firstname, lastname,
                                          employeeid, departmentname)
    {
      this.firstname = firstname;
      this.lastname = lastname;
      this.employeeid = employeeid;
      this.departmentname = departmentname
    }
```

As you can see in the following excerpt from Listing 1-2, the `readyStateChangeCallback` function invokes a JavaScript function named `deserialize`:

```
var employee = deserialize();
```

This `deserialize` JavaScript function encapsulates the logic that deserializes an `employee` object from the server data (described in more detail later). This logic is what is referred to as the Serializer in Figure 1-2.

Next, the `readyStateChangeCallback` function uses DHTML to update the relevant parts of the page with employee information in the `employee` object. First, it calls the `getElementyById` method on the `document` object to return a reference to the `<span>` DOM element with the id HTML attribute of

`firstname`, and assigns the `firstname` property of the employee object to the `innerText` property of this DOM element to display the first name of the employee:

```
        var firstnamespan = document.getElementById("firstname");
        firstnamespan.innerText = employee.firstname;
```

Next, it calls the `getElementyById` method again, this time to return a reference to the `<span>` DOM element with the id HTML attribute of `lastname`, and assigns the `lastname` property of the employee object to the `innerText` property of this DOM element to display the last name of the employee:

```
        var lastnamespan = document.getElementById("lastname");
        lastnamespan.innerText = employee.lastname;
```

It then repeats the same process to display the employee's id and department name:

```
        var employeeidspan = document.getElementById("employeeid");
        employeeidspan.innerText = employee.employeeid;

        var departmentnamespan = document.getElementById("departmentname");
        departmentnamespan.innerText = employee.departmentname;
```

As mentioned, the `deserialize` JavaScript function deserializes an employee object from the server data:

```
    function deserialize(response)
    {
      var delimiter="|";
      var responseIndex = 0;
      var delimiterIndex;

      delimiterIndex = response.indexOf(delimiter, responseIndex);
      var firstname = response.substring(responseIndex, delimiterIndex);
      responseIndex = delimiterIndex + 1;
      delimiterIndex = response.indexOf(delimiter, responseIndex);
      var lastname = response.substring(responseIndex, delimiterIndex);
      responseIndex = delimiterIndex + 1;

      delimiterIndex = response.indexOf(delimiter, responseIndex);
      var employeeid = response.substring(responseIndex, delimiterIndex);
      responseIndex = delimiterIndex + 1;

      delimiterIndex = response.indexOf(delimiter, responseIndex);
      var departmentname = response.substring(responseIndex, delimiterIndex);

      return new employee(firstname, lastname, employeeid, departmentname);
    }
```

The `deserialize` function basically contains the logic that knows how to parse a string with the following format into an `employee` object:

```
   Shahram|Khosravi|22223333|Some Department|
```

XML

As you saw earlier, Listing 1-2 contains a JavaScript function named `serialize` that serializes a given credentials object into a string with the following format before this object is sent over the wire to the server:

```
usernametbx=username&passwordtbx=password
```

Listing 1-2 also contains a JavaScript function named `deserialize` that deserializes an `employee` object from a string with the following format:

```
Shahram|Khosravi|22223333|Some Department|
```

The `serialize` and `deserialize` methods encapsulate the logic that was referred to as the Serializer in Figure 1-2.

The great thing about the XML format is that the server- and client-side technologies provide built-in support for serializing objects into XML and deserializing objects from XML. Listing 1-3 presents a new version of Listing 1-2 where the `Page_Load` server-side method serializes the server data into XML, which is then sent over the wire to the client, where the `deserialize` JavaScript function deserializes an `employee` object from the XML.

Listing 1-3: A version of Listing 1-2 that uses XML format

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.IO" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
  void Page_Load(object sender, EventArgs e)
  {
    if (Request.Headers["MyCustomHeader"] != null)
    {
      if (Request.Form["passwordtbx"] == "password" &&
          Request.Form["usernametbx"] == "username")
      {
        string xml="";
        using (StringWriter sw = new StringWriter())
        {
          XmlWriterSettings settings = new XmlWriterSettings();
          settings.Indent = true;
          settings.OmitXmlDeclaration = true;
          using (XmlWriter xw = XmlWriter.Create(sw, settings))
          {
            xw.WriteStartDocument();
            xw.WriteStartElement("employeeInfo");
            xw.WriteElementString("firstName", "Shahram");
            xw.WriteElementString("lastName", "Khosravi");
            xw.WriteElementString("employeeId", "22223333");
            xw.WriteElementString("departmentName", "Some Department");
            xw.WriteEndElement();
```

```
            xw.WriteEndDocument();
          }
          xml = sw.ToString();
        }
        Response.ContentType = "text/xml";
        Response.Write(xml);
        Response.End();
      }
      else
      throw new Exception("Wrong credentials");
    }
  }
</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
  <title>Untitled Page</title>
  <script type="text/javascript" language="javascript">
  var request;

  if (!window.XMLHttpRequest)
  {
    // Same as Listing 2
  }

  window.employee = function window$employee(firstname, lastname,
                                          employeeid, departmentname)
  {
    // Same as Listing 2
  }

  function deserialize()
  {
    var response = request.responseXML;
    var employeeInfo = response.documentElement;
    var firstNameElement = employeeInfo.childNodes[0];
    var firstname = firstNameElement.firstChild.nodeValue;

    var lastNameElement = employeeInfo.childNodes[1];
    var lastname = lastNameElement.firstChild.nodeValue;

    var employeeIdElement = employeeInfo.childNodes[2];
    var employeeid = employeeIdElement.firstChild.nodeValue;

    var departmentNameElement = employeeInfo.childNodes[3];
    var departmentname = departmentNameElement.firstChild.nodeValue;

    return new employee(firstname, lastname, employeeid, departmentname);
  }

  function readyStateChangeCallback()
  {
    // Same as Listing 2
  }

  window.credentials = function window$credentials(username, password)
```

(continued)

Listing 1-3 *(continued)*

```
    {
      // Same as Listing 2
    }

    function serialize(credentials)
    {
      // Same as Listing 2
    }

    function submitCallback()
    {
      // Same as Listing 2
    }
    </script>
  </head>
  <body>
    <form id="form1" runat="server">
      <!- Same as Listing 2 ->
    </form>
  </body>
  </html>
```

Now let's walk through the implementations of the `Page_Load` server-side method and the `deserialize` JavaScript function in this listing, starting with the `Page_Load` method.

The `Page_Load` method begins by instantiating a `StringWriter` into which the XML data will be written:

```
    string xml = "";
    using (StringWriter sw = new StringWriter())
```

Then it instantiates an `XmlWriterSettings` object that specifies the settings for the XML document. In this case, the XML document will be indented and it will not contain the XML declaration:

```
            XmlWriterSettings settings = new XmlWriterSettings();
            settings.Indent = true;
            settings.OmitXmlDeclaration = true;
```

Next, it instantiates an `XmlWriter` object with the specified settings and wraps the `StringWriter`. In other words, this `XmlWriter` will write the XML into the `StringWriter`:

```
            using (XmlWriter xw = XmlWriter.Create(sw, settings))
```

Then, it invokes the `WriteStartDocument` method on the `XmlWriter` to mark the beginning of the XML document:

```
              xw.WriteStartDocument();
```

Next, it invokes the `WriteStartElement` method on the `XmlWriter` to write a new element named `employeeInfo` into the `XmlWriter`, which in turn writes this element into the `StringWriter`:

```
            xw.WriteStartElement("employeeInfo");
```

This element will act as the document element of the XML document. Every XML document must have a single element known as the *document element* that encapsulates the rest of the XML document.

`Page_Load` then invokes the `WriteElementString` method four times to write three elements named `firstName`, `lastName`, `employeeId`, and `departmentName` with the specified values into the `XmlWriter`, which in turn writes these elements into the `StringWriter`:

```
            xw.WriteElementString("firstName", "Shahram");
            xw.WriteElementString("lastName", "Khosravi");
            xw.WriteElementString("employeeId", "22223333");
            xw.WriteElementString("departmentName", "Some Department");
```

Next, `Page_Load` invokes the `ToString` method on the `StringWriter` to return a string that contains the entire XML document:

```
            xml = sw.ToString();
```

Then, it sets the `Content-Type` HTTP response header to the value `text/xml` to signal the client code that the server response contains XML data:

```
        Response.ContentType="text/xml";
```

Next, it writes the string that contains the XML data into the server response output stream:

```
        Response.Write(xml);
```

Finally, it invokes the `End` method on the `Response` object to end the response right away and, consequently, to send the XML document to the client, bypassing the normal rendering routine of the current page:

```
        Response.End();
```

Now let's walk through the implementation of the `deserialize` JavaScript function in Listing 1-3. This function invokes the `responseXML` property on the `XMLHttpRequest` object to return the XML document:

```
    var response = request.responseXML;
    var employeeInfo = response.documentElement;
```

Then, it uses the XML API to extract the employee's `firstname`, `lastname`, `employeeid`, and `departmentname` from the XML document:

```
        var firstNameElement = employeeInfo.childNodes[0];
        var firstname = firstNameElement.firstChild.nodeValue;

        var lastNameElement = employeeInfo.childNodes[1];
        var lastname = lastNameElement.firstChild.nodeValue;

        var employeeIdElement = employeeInfo.childNodes[2];
        var employeeid = employeeIdElement.firstChild.nodeValue;

        var departmentNameElement = employeeInfo.childNodes[3];
        var departmentname = departmentNameElement.firstChild.nodeValue;
```

Finally, it instantiates and returns an `employee` object with the returned `firstname`, `lastname`, `employeeid`, and `departmentname`:

```
        return new employee(firstname, lastname, employeeid, departmentname);
```

JSON

One of the main tasks in an AJAX-enabled application is to serialize client/server-side objects into the *appropriate* format before data is sent over the wire and to deserialize client/server-side objects from an *appropriate* format after data is received over the wire. In general there are two common data-interchange formats: XML and JSON. XML format was discussed in the previous section. Now let's move on to the second common data-interchange format: JSON.

JavaScript Object Notation (*JSON*) is a data-interchange format based on a subset of the JavaScript language. The following sections present the fundamental JSON concepts and terms.

object

A JSON object is an unordered, comma-separated list of name/value pairs enclosed within a pair of braces. The name and value parts of each name/value pair are separated by a colon (`:`). The name part of each name/value pair is a string; and the value part is an array, another object, a string, a number, `true`, `false`, or `null`.

array

A JSON array is an ordered, comma-separated list of values enclosed within a pair of square brackets (`[ ]`). Each value is an array, another object, a string, a number, `true`, `false`, or `null`.

string

A JSON string is a collection of zero or more Unicode characters enclosed within double quotes (`" "`). You must use a JSON string to represent a single character, and the character must be in double quotes. You must use the backslash character (`\`) to escape the following characters:

- Quotation mark (`\"`)
- Solidus (`\/`)

- Reverse solidus (`\\`)
- Backspace (`\b`)
- Formfeed (`\f`)
- Newline (`\n`)
- Carriage return (`\r`)
- Horizontal tab (`\t`)

number

A JSON number is very similar to a C# number with one major exception: JSON does not support octal and hexadecimal formats.

null, true, and false

JSON supports `null`, `true`, and `false` as valid values.

JSON is a simple-yet-powerful, data-interchange format. It has the same hierarchical nature as XML, without the extra angle brackets, as shown in the following example:

```
{
 "departments":[
                {"departmentName":"department1",
                 "departementManager":{"name":"someName1",
                                       "employeeID":1,
                                       "managesMultipleDepts":true
                                      },
                "sections":[
                            {"sectionName":"section1",
                             "sectionManager":{"name":"someName2",
                                               "employeeID":2
                                              },
                            "employees":[
                                         {"name":"someName3",
                                          "employeeID":3
                                         },
                                         {"name":"someName4",
                                          "employeeID":4
                                         }
                                        ]
                            },
                            {"sectionName":"section2",
                             "sectionManager":{"name":"someName5",
                                               "employeeID":5
                                              },
                             "employees":[
                                         {"name":"someName6",
                                          "employeeID":6
                                         },
```

(continued)

```
                                          {"name":"someName7",
                                           "employeeID":7
                                          }
                                         ]
                                }
                               ]
                 }
                ]
}
```

One of the great things about JSON is that JavaScript provides easy, built-in support for parsing a JSON representation, as shown in Listing 1-4. This example is a version of Listing 1-2 that uses JSON.

Listing 1-4: A version of Listing 1-2 that uses JSON

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.IO" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
  void Page_Load(object sender, EventArgs e)
  {
    if (Request.Headers["MyCustomHeader"] != null)
    {
      if (Request.Form["passwordtbx"] == "password" &&
          Request.Form["usernametbx"] == "username")
      {
       string json="{\"firstname\": \"Shahram\",";
       json += "\"lastname\": \"Khosravi\",";
       json += "\"employeeid\": 22223333,";
       json += "\"departmentname\": \"Some Department\"}";
       Response.Write(json);
       Response.End();
      }
      else
        throw new Exception("Wrong credentials");
    }
  }
</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
  <title>Untitled Page</title>
  <script type="text/javascript" language="javascript">
  var request;

  if (!window.XMLHttpRequest)
  {
    // Same as Listing 2
  }

  function readyStateChangeCallback()
  {
    if (request.readyState == 4 && request.status == 200)
    {
```

```
        var credentials = document.getElementById("credentials");
        credentials.style.display="none";
        var employeeinfotable = document.getElementById("employeeinfo");
        employeeinfotable.style.display="block";

        var response = request.responseText;
        eval("var employee = " + response + ";");

        var firstnamespan = document.getElementById("firstname");
        firstnamespan.innerText = employee.firstname;
        var lastnamespan = document.getElementById("lastname");
        lastnamespan.innerText = employee.lastname;

        var employeeidspan = document.getElementById("employeeid");
        employeeidspan.innerText = employee.employeeid;

        var departmentnamespan = document.getElementById("departmentname");
        departmentnamespan.innerText = employee.departmentname;
      }
    }

    window.credentials = function window$credentials(username, password)
    {
      // Same as Listing 2
    }

    function serialize(credentials)
    {
      // Same as Listing 2
    }

    function submitCallback()
    {
      // Same as Listing 2
    }
    </script>
  </head>
  <body>
    <form id="form1" runat="server">
      // Same as Listing 2
    </form>
  </body>
  </html>
```

In this listing, the `Page_Load` method generates a string that contains the JSON representation of the employee object. This method writes the JSON representation into the response output stream and ends the response as usual:

```
          string json="{\"firstname\": \"Shahram\",";
          json += "\"lastname\": \"Khosravi\",";
          json += "\"employeeid\": 22223333,";
          json += "\"departmentname\": \"Some Department\"}";

          Response.Write(json);
          Response.End();
```

Things are pretty simple on the client side, as you can see in the following code fragment from Listing 1-4:

```
var response = request.responseText;
eval("var employee=" + response + ";");
```

This simply calls the `eval` JavaScript function to deserialize an `employee` object in the JSON string received from the server. As you can see, the messy XML deserialization code presented in Listing 1-3 is all gone and replaced with a simple call into the `eval` JavaScript function. However, this simplicity comes with a price. Because the `eval` JavaScript function basically trusts the scripts that it runs, it introduces serious security issues. This is not a problem in this example because the JSON representation is coming from a trusted server. However, in general, you must be very careful about what gets passed into `eval`.

ASP.NET AJAX

The ASP.NET AJAX framework brings to the world of AJAX-enabled Web application development what ASP.NET and the .NET Framework brought to the world of server-side Web application development over the past few years. The biggest advantage of ASP.NET over the earlier server-side Web development technologies such as the classic ASP is that you get to program in the .NET Framework, which provides the following benefits among many others:

- The .NET Framework is a full-fledged, object-oriented framework that enables you to take full advantage of all the well-known benefits of object-oriented programming such as classes, interfaces, namespaces, polymorphism, inheritance, and the like.
- The .NET Framework comes with a large set of managed classes with convenient methods, properties, and events that save you from having to write lots of infrastructure and generic code that have nothing to do with the specifics of your application.
- The .NET Framework includes a full-fledged typing and type-reflection system that enables you to perform runtime type inspections, discoveries, instantiations, invocations, and the like.
- The .NET Framework provides you with groundbreaking facilities and capabilities such as the following:
  - *Application lifecycle and its events:* The `HttpApplication` object that represents an ASP.NET application goes through a set of steps or phases collectively known as the application lifecycle. This object raises events before and/or after each lifecycle phase to allow you to customize the application lifecycle.
  - *Page lifecycle and its events:* Every ASP.NET page goes through a set of steps or phases collectively known as the page lifecycle. The `Page` object that represents the ASP.NET page raises events before and/or after each lifecycle phase to allow you to customize the page lifecycle.
  - *Server controls:* Server controls enable you to program against the underlying markup using the .NET Framework and its rich, object-oriented class library. This gives you the same programming experience as these server controls desktop counterparts provide.

- *Control architecture:* Every server control goes through a set of steps or phases collectively known as the control lifecycle, and raises events before and/or after each lifecycle phase to allow you to customize the control lifecycle.
- *Declarative programming:* The ASP.NET declarative programming enables you to program declaratively without writing a single line of imperative code. The ASP.NET runtime automatically parses the declarative code, dynamically generates the associated imperative code, dynamically compiles the imperative code, caches the compiled imperative code for future use, and instantiates and initializes the associated compiled .NET types.

Thanks to ASP.NET and the .NET Framework, the server-side Web application development world can take full advantage of these important programming benefits to enormously boost productivity and to write more reliable and architecturally sound programs.

As you'll see throughout this book, the ASP.NET AJAX framework provides similar programming benefits to developers of AJAX-enabled Web applications. The ASP.NET AJAX Framework consists of two frameworks: the ASP.NET AJAX client-side framework and the ASP.NET AJAX server-side framework. The ASP.NET AJAX server-side framework is an extension of the ASP.NET Framework, which provides all the server-side support that an AJAX-enabled Web application needs.

Installing the ASP.NET AJAX Extensions and ASP.NET Futures

Make sure both the ASP.NET AJAX Extensions and ASP.NET Futures are installed on your computer. You can download free copies of the ASP.NET AJAX Extensions and ASP.NET Futures from the official Microsoft ASP.NET AJAX site at

Summary

This chapter first discussed the main AJAX technologies. Then it provided a brief description of the ASP.NET AJAX framework. As mentioned, the ASP.NET AJAX framework consists of two main frameworks: the ASP.NET AJAX client-side framework and ASP.NET AJAX server-side framework.

The next chapter begins your journey of the ASP.NET AJAX client-side framework, where you'll learn a great deal about the ASP.NET AJAX JavaScript base type extensions.