Pro JSF and Ajax

Building Rich Internet Components

Jonas Jacobi and John R. Fallows

Apress®

Pro JSF and Ajax: Building Rich Internet Components

ISBN-13: 978-1-59059-580-0

ISBN-10: 1-59059-580-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Tony Davis
Technical Reviewers: Peter Lubbers, Kito D. Mann, Matthias Wessendorf
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Managers: Beckie Stones, Elizabeth Seymour
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Molly Sharp, ContentWorks
Proofreader: Elizabeth Berry
Indexer: Carol Burbo
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

CHAPTER 4

Using Rich Internet Technologies

Ajax—in Greek mythology Ajax was a powerful warrior who fought in the Trojan War and supposedly was second only to Achilles, the Greeks' best warrior. Although characterized as slow-witted, Ajax was one of the best fighters among the Greeks and was famed for his steadfast courage in the face of adversity.

—Laboratori Nazionali di Frascati (`http://www.lnf.infn.it`)

It will always be the user who will feel the effect of the technology you choose, and the first priority of any Web or desktop application developer should be the user experience. Users are not interested in what technology is being used or whether the application is a traditional desktop application or a Web application. Users demand a feature-rich and interactive interface.

Traditionally, desktop applications have been able to provide users with the richness required to fulfill their demands, but an increasing number of desktop applications are migrating to the Web. Therefore, Web application developers have to provide richer Web interfaces.

To make you fully appreciate JSF and what it brings to the Internet community, you need to understand the current status of rich Internet applications. Web application developers today are faced with a demand for richer functionality using technologies such as HTML, CSS, JavaScript, and the DOM. However, these technologies were not developed with enterprise applications in mind. The increasing demand from consumers for applications with features not fully supported by these technologies is pushing Web application developers to explore alternative solutions.

New breeds of Web technologies that enhance the traditionally static content provided by Web applications have evolved from these consumer requirements. These technologies are often referred to as *Rich Internet Technologies* (RITs).

In the absence of a standard definition and with the lack of extensibility of the traditional Web technologies, new technologies have emerged, such as Mozilla's XUL, Microsoft's HTC, Java applets, Flex, and OpenLaszlo. These technologies support application-specific extensions to traditional HTML markup while still leveraging the benefits of deploying an application to a central server. Another solution that has returned under a newly branded name is Ajax (recently an acronym for Asynchronous JavaScript and XML and formerly known as XMLHTTP). Applications built with these technologies are often referred to as *Rich Internet Applications* (RIAs).

In this chapter, we will introduce three RITs: Ajax, Mozilla XUL, and Microsoft HTC. This chapter will give a high-level overview of these technologies, and it will show some simple examples to highlight the core feature of each technology. In later chapters, you will get into the details of each technology to improve the user experience of two JSF components—`ProInputDate` and `ProShowOneDeck`.

The following are the four main players in this chapter:

Ajax[1]: Ajax is the new name of an already established technology suite—the DOM, JavaScript, and `XMLHttpRequest`. Ajax is used to create dynamic Web sites and to asynchronously communicate between the client and server.

XUL: XML User Interface Language (XUL) which, pronounced *zuul*, was created by the Mozilla organization (Mozilla.org) as an open source project in 1998. With XUL, developers can build rich user interfaces that may be deployed either as "thin client" Web applications, locally on a desktop or as Internet-enabled "thick client" desktop applications.

XBL: Extensible Binding Language (XBL) is a language used by XUL to define new components. XBL is also used to bridge the gap between XUL and HTML, making it easy to attach behavior to traditional HTML markup.

HTC: Introduced in Microsoft Internet Explorer 5, HTCs provide a mechanism to implement components in script as DHTML behaviors. Saved with an `.htc` extension, an HTC file is an HTML file that contains script and a set of HTC-specific elements that define the component.

After reading this chapter, you should understand what these RITs are, what they provide, and how you can create rich user interface components with them.

Introducing Ajax

Ajax has been minted as a term describing a Web development technique for creating richer and user-friendlier Web applications. In this chapter, we will give you an overview of Ajax.

Ajax was first coined in February 2005 and has since taken the software industry by storm. One of the reasons Ajax has gained momentum and popularity is the `XMLHttpRequest` object and the way this object makes it possible for developers to asynchronously communicate with underlying servers and any business services used by Web applications. Popular sites such as Google GMail and Google Suggest are using Ajax techniques to provide users with rich interfaces that have increased the awareness of Ajax.

Although the name *Ajax* is new, the technologies listed as the foundation of this technique—JavaScript, `XMLHttpRequest`, and the DOM—have been around for some time. In fact, the latest addition to this suite of technologies—the `XMLHttpRequest` object—was introduced by Microsoft in 1999 with the release of Internet Explorer 5.0 and was implemented as an ActiveX component.

The `XMLHttpRequest` object, although widely used, is not a standard; it could at best be called a "de facto" standard, since most modern browsers, including Firefox, Internet Explorer,

[1] This term was first coined in an article by James Garrett of Adaptive Path.

Opera, and Safari, support it. However, a standard has been proposed that covers some of the functionality provided by the `XMLHttpRequest` object—the DOM Level 3 Load and Save specification.

■ **Note** The `XMLHttpRequest` object is not a W3C standard. The W3C DOM Level 3 Load and Save specification contains some similar functionality, but this is not implemented in any browsers yet. So, at the moment, if you need to send an HTTP request from a browser, you will have to use the `XMLHttpRequest` object.

With the `XMLHttpRequest` object, developers can now send requests to the Web server to retrieve specific data and use JavaScript to process the response. This ability to send data between the client and the Web server reduces the bandwidth to a minimum and saves time on the server since most of the processing to update the user interfaces takes place on the client using JavaScript.

The XMLHttpRequest Object

Since the `XMLHttpRequest` object is not a standard, each browser may implement support for it slightly differently; thus, the behavior might vary among browsers. You will notice when creating the sample application in this chapter that Microsoft's Internet Explorer implements the `XMLHttpRequest` object as an ActiveX object, whereas Mozilla Firefox treats it like a native JavaScript object. However, most implementations support the same set of methods and properties. This eases the burden on application developers, since the only difference is in creating an instance of the `XMLHttpRequest` object. Creating an instance of the `XMLHttpRequest` object can look like Code Sample 4-1 or Code Sample 4-2.

Code Sample 4-1. *Creating an Instance of the* `XMLHttpRequest` *Object*

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

Code Sample 4-2. *Creating an Instance of the* `XMLHttpRequest` *Object Using* `ActiveXObject`

```
var xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
```

It is also worth noting that the `XMLHttpRequest` object is not exclusive to standard HTML. The `XMLHttpRequest` object can potentially be used by any HTML/XML-based Web technology such as XUL or HTC.

Methods

An `XMLHttpRequest` object instance provides methods that can be used to asynchronously communicate with the Web server (see Table 4-1).

Table 4-1. `XMLHttpRequest` *Object Methods*

Method	Description
`open("method", "URL")`	Assigns destination URL, method, and other optional attributes of a pending request
`send(content)`	Transmits the request, optionally with a string that can be posted or DOM object data
`abort()`	Stops the current request
`getResponseHeader("headerLabel")`	Returns the string value of a single header label
`getAllResponseHeaders()`	Returns a complete set of headers (labels and values) as a string
`setRequestHeader("label", "value")`	Assigns a label/value pair to the header to be sent with a request

In Table 4-1, the `open()` and `send()` methods are the most common ones. The `open("method", "URL"[, "asynch"[, "username"[, "password"]]])` method sets the stage for the request and upcoming operation. Two parameters are required; one is the HTTP method for the request (`GET` or `POST`), and the other is the URL for the connection. The optional `asynch` parameter defines the nature of this request—`true` being the default and indicating that this is an asynchronous request. The other two optional parameters—`username` and `password`—allow application developers to provide a username and password, if needed.

The `send()` method makes the request to the server and is called after you have set the stage with a call to the `open()` method. Any content passed to this method is sent as part of the request body.

Properties

Once an `XMLHttpRequest` has been sent, scripts can look to several properties that all implementations have in common (see Table 4-2).

Table 4-2. `XMLHttpRequest` *Object Properties*

Property	Description
`onreadystatechange`	Event handler for an event that fires at every state change
`readyState`	Object status integer: `0` = uninitialized, `1` = loading, `2` = loaded, `3` = interactive, `4` = complete
`responseText`	String version of data returned from server process
`responseXML`	DOM-compatible document object of data returned from server process
`status`	Numeric code returned by server, such as 404 for “Not Found” or 200 for “OK”
`statusText`	String message accompanying the status code

As with the XMLHttpRequest object methods, two properties will be used more frequently than the others—responseText and responseXML. You can use these two properties to access data returned with the response. The responseText property provides a string representation of the data, which is useful in case the requested data comes in as plain text or HTML. Depending on the context, the responseXML property offers a more extensive representation of the data. The responseXML property will return an XML document object, which can be examined using W3C DOM node tree methods and properties.

Traditional Web Application Development

Before getting into the details of Ajax, you need to first understand how a traditional Web application works and what issues users, and application developers, face when a Web application contains form elements. HTML forms are used to pass data to an underlying Web server. You have probably encountered Web applications with forms, such as when you have filled in a survey, ordered products online from Web sites such as eBay (http://www.ebay.com), or filled in an expense report with a company's HR application.

A form in a traditional Web application is defined by a special HTML tag (<form>) that has a set of parameters—action, method, enctype, and target. The action parameter defines the destination URL to pass the form data, the method parameter defines the HTTP method used for the form postback, the enctype parameter defines the content type to be used for encoding the data, and the target parameter defines the frame that should receive the response.

Regular Postback

You can use two methods when submitting a form—POST and GET. With the HTTP GET method, the form data set is appended to the URL specified by the action attribute (for example, http://forums.oracle.com/forums/forum.jspa?forumID=83), and this new URL is sent to the server. In JSF the value of the action attribute is provided by ViewHandler.getActionURL(viewId) during rendering.

■ **Note** The <h:form> tag defined by the JSF specification does not have the method and action attributes.

With the HTTP POST method, the form data set is included in the body of the request sent to the server. The GET method is convenient for bookmarking, but should be used only when you do not expect form submission side effects as defined in the W3C HTTP specification (http://www.w3.org/Protocols/). If the service associated with the processing of a form causes side effects (for example, if the form modifies a database row or subscribes to a service), you should use the POST method.

Another reason for choosing the POST method over the GET method is that it allows browsers to send an unlimited amount of data to a Web server by adding data as the message body after the request headers on an HTTP request. The GET method is restricted to the URL length, which cannot be more than 2,048 characters. POST removes any limitations from the transmitted data length.

■ **Note** The `GET` method restricts form data set values to ASCII characters. Only the `POST` method (with `enctype="multipart/form-data"`) is specified to cover the entire [ISO10646] character set.

When the user submits a form (for example, by clicking a submit button), as shown in Figure 4-1, the browser processes the controls within the submitted form and builds a form data set. A form data set is a sequence of control-name/current-value pairs constructed from controls within the form. The form data set is then encoded according to the content type specified by the `enctype` attribute of the `<form>` element (for example, `application/x-www-form-urlencoded`).

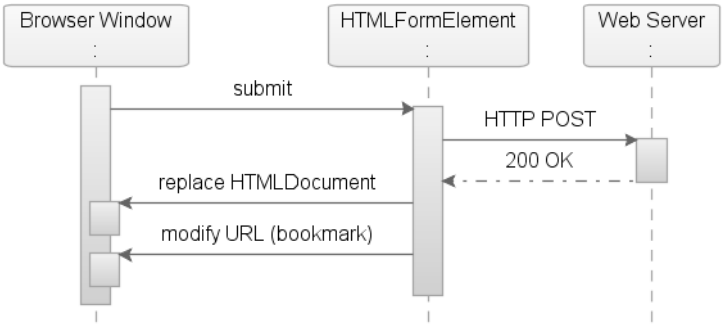

Figure 4-1. *Sequence diagram over a regular postback*

The encoded data is then sent as a `url-formencoded` stream back to the server (HTTP `POST`). The server response contains information about the response status indicating that the request has succeeded (HTTP status code 200 "OK") and sends a full-page response. The browser will then parse the HTML sent on the response to the HTML DOM and render the page in the browser window. Any resources required by the page will be reverified and possibly downloaded again from the server. After the HTML document has been replaced in the browser window, the URL in the browser location bar is also modified to reflect the page from the previous page form action.

Alternatively, the server response can contain information indicating that the request has failed (for example, HTTP status code 404 "Not Found").

Side Effects of Regular Postback

The obvious undesired side effect of regular postback is that it will cause the page to flicker when the page is reloaded in the browser window, and at worst the user will have to wait while

the page downloads all the required resources to the client browser again. Other less prominent, but still annoying, side effects are the loss of scroll position and cursor focus.

■ **Note** Most browsers today have excellent client-side caching functionalities that work well to prevent pages from reloading resources from the Web server, unless caching is turned off or the application is using HTTPS, in which case content may be prevented from being cached on the client.

As part of a page design, it might be required to have multiple forms on a page. When multiple forms are available on a page, only one form will be processed during postback, and the data entered in other forms will be discarded.

One benefit is that bookmarking is possible with regular postbacks. However, the user is often fooled by the URL set in the location bar, since it reflects what was last requested and not what is returned on the response. When the user selects the bookmark, it will return to the previously submitted page. A regular postback also allows the user to click the browser back button to return to the previous page with the only side effect that a form post warning will occur.

Ajax Web Application Development

Developing sophisticated Ajax-enabled applications is not something for the everyday application developer, and just as the Trojans feared Ajax on the battlefield, even the most experienced Web designer dreads to attack Ajax. A major part of the Ajax framework is the client-side scripting language JavaScript. As many Web designers have experienced, JavaScript is not an industrial-strength language and is claimed by many to lack support in professional development tools.

However, in our opinion, at least two really good JavaScript tools are available—Microsoft's Visual Studio and Mozilla's Venkman. What is true, though, is that maintaining Ajax applications is difficult; the lack of browser consistency in JavaScript implementations makes maintaining browser-specific code a challenge.

MOZILLA'S VENKMAN DEBUGGER

Venkman is the code name for Mozilla's JavaScript debugger (`http://www.mozilla.org/projects/venkman/`). Venkman aims to provide a powerful JavaScript debugging environment for Mozilla-based browsers, including the Netscape 7.*x* series of browsers and Mozilla milestone builds. It does not include Gecko-only browsers such as K-Meleon and Galeon. The debugger is available as an add-on package in XPI format and has been provided as part of the Mozilla install distribution since October 3, 2001.

Ajax Postback

Now that you have familiarized yourself with regular postbacks, it is time to look at Ajax. This section will give you an overview of how to use Ajax postbacks to handle events. You can use Ajax to take control of the form submit action, and instead of using the regular form submit action, you use an `XMLHttpRequest` object to asynchronously submit your request to the Web server. As a side effect, when the user submits a form (for example, by clicking a submit button), no browser helps you process the controls within the submitted form. You now need to handle any form fields that need to be part of the postback and use them to build a form data set—control-name/current-value pairs. You then take the form data set and simulate the encoding (`url-formencoded`) to provide the same syntax as a regular postback (see Figure 4-2).

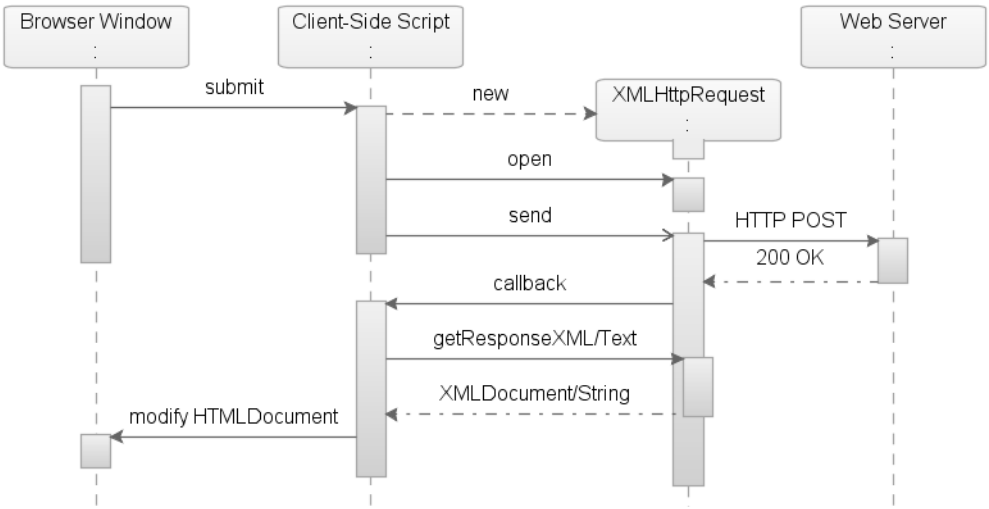

Figure 4-2. *Sequence diagram over an* `XMLHttpRequest` *postback*

After you have created the `XMLHttpRequest` object, you use the `open()` method to set the HTTP method—`GET` or `POST`—intended for the request and the URL for the connection. After you have set the stage for your `XMLHttpRequest` operation, you send the encoded data, using the `XMLHttpRequest` object, as a `url-formencoded` stream back to the server (HTTP `POST`). For the Web server, the request will appear as a traditional HTTP `POST`, meaning that the Web server cannot tell the difference between a regular postback and your Ajax postback. For a JSF solution, this means an Ajax request can be picked up the same way as a regular postback request, allowing server code (for example, JSF request lifecycle) to be unaffected.

If the request is successful, the ready state on your `XMLHttpRequest` object is set to `4`, which indicates that the loading of the response is complete. You can then use two properties to access data returned with the response—`responseText` and `responseXML`.

The `responseText` property provides a string representation of the data, which is useful in case the requested data comes in the shape of plain text or HTML. Depending on the context, the `responseXML` property offers a more extensive representation of the data.

The `responseXML` property will return an XML document object, which is a full-fledged document node object (a DOM `nodeType` of `9`) that can be examined using the W3C DOM node

tree methods and properties. In this traditional Ajax approach, the Ajax handler is in charge of sending the data, managing the response, and modifying the `HTMLDocument` object node tree.

■ **Note** DOM elements can be different types. An element's type is stored in an integer field of `nodeType` (for example, `COMMENT_NODE = 8` and `DOCUMENT_NODE = 9`). For more information about the different `nodeType`s, please visit `http://www.w3.org/`.

Side Effects of Ajax Postback

As with the regular postback, desired and undesired side effects exist when using Ajax for postback. The most prominent and desired side effect is the `XMLHttpRequest` object's strength and ability to set or retrieve parts of a page. This will remove flickering when data is reloaded and increase performance of the application, since there is no need to reload the entire page and all its resources. The undesired side effect of this is that users will typically no longer be able to bookmark a page or use the back button to navigate to the previous page/state.

Another important, but less immediately obvious, implication of using `XMLHttpRequest` in your application is that clients such as mobile phones, PDAs, screen readers, and IM clients lack support for this technology. Also, Ajax requires additional work to make applications accessible; for example, screen readers expect a full-page refresh to work properly.

■ **Note** With `XMLHttpRequest`, you do not need the form element in an application, but one function requires a form regardless of regular postbacks or Ajax postbacks—file upload. If you need file-upload functionality in your application, you have to use `form.submit()`. In the context of Ajax, you can do this by using a hidden `<iframe>` tag and the `form.submit()` function and setting `target`.

Ajax Is Not a Magic Wand

As you know, the `XMLHttpRequest` object is an important player in Ajax, since it transports data asynchronously between the client and the server. It is important to understand that the `XMLHttpRequest` is not a magic wand that automatically solves all your problems. You still need to watch performance and scalability carefully using the `XMLHttpRequest` object. If you are aware of this, it is easy to understand that it is what you send on the request, receive upon the response, and manage on the client that will affect your performance.

Building Ajax Applications

Traditional Web applications are in most cases slower than their desktop application counterparts. With Ajax, you can now send requests to the Web server to retrieve only the data needed using JavaScript to process the response, which creates a more responsive Web application. Figure 4-3 illustrates a page using Ajax to asynchronously communicate with the back-end

and provide a Book Titles drop-down list that includes certain books based on what category the user enters.

Figure 4-3. *An HTML page using Ajax to filter a list of books based on category*

When the user tabs out of the Book Category field, the drop-down list is populated with books based on the entered category without a page refresh.

Figure 4-4 shows the result of entering *Ajax* as the category and tabbing out of the Book Category field.

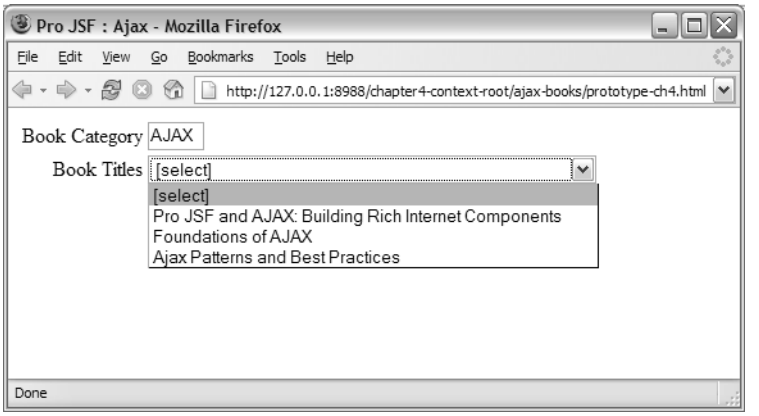

Figure 4-4. *An HTML page using Ajax to filter a list of books based on category*

As you can see, the Book Titles drop-down list has been populated with books about the related topic.

A traditional Ajax application leverages standard HTML/XHTML as the presentation layer and JavaScript to dynamically change the DOM, which creates an effect of "richness" in the

user interface with no dependency on a particular runtime environment. Code Sample 4-3 shows the actual HTML source behind this simple application.

Code Sample 4-3. *An HTML Page Leveraging Ajax to Update a* `<select>` *Element*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <script type="text/javascript"
            src="projsf-ch4/dynamicBookList.js" >
    </script>
    <title>Select a book</title>
  </head>
  <body>
    <form name="form" method="get">
      <table>
        <tr>
          <td align="right">Book Category</td>
          <td>
            <input type="text" size="3" maxlength="8"
                   onchange="populateBookList('/chapter4-context-root/projsf-ch4',
                                              'bookListId', this.value);" />
          </td>
        </tr>
        <tr>
          <td align="right">Book Title</td>
          <td >
            <select id="bookListId" >
              <option value="[none]">
                [enter a book category]
              </option>
            </select>
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

At the top of this page, you have a reference to your Ajax implementation—`dynamicBookList.js`. This code adds an `onchange` event handler to the `<input>` element that will call a JavaScript function, `populateBookList()`, which is invoked when the cursor leaves the input field. The `populateBookList()` function takes three arguments—the service URL for retrieving the book list data, the book category entered in the input field `this.value`, and the ID of the `select` element to populate with books (`'bookListId'`).

The Ajax Book Filter Implementation

The Ajax book filter implementation consists of three JavaScript functions—`populateBookList()`, `createHttpRequest()`, and `transferListItems()`—and a data source containing information about the books. As soon as the cursor leaves the Book Category field, the `getBookList()` function is invoked (see Figure 4-5).

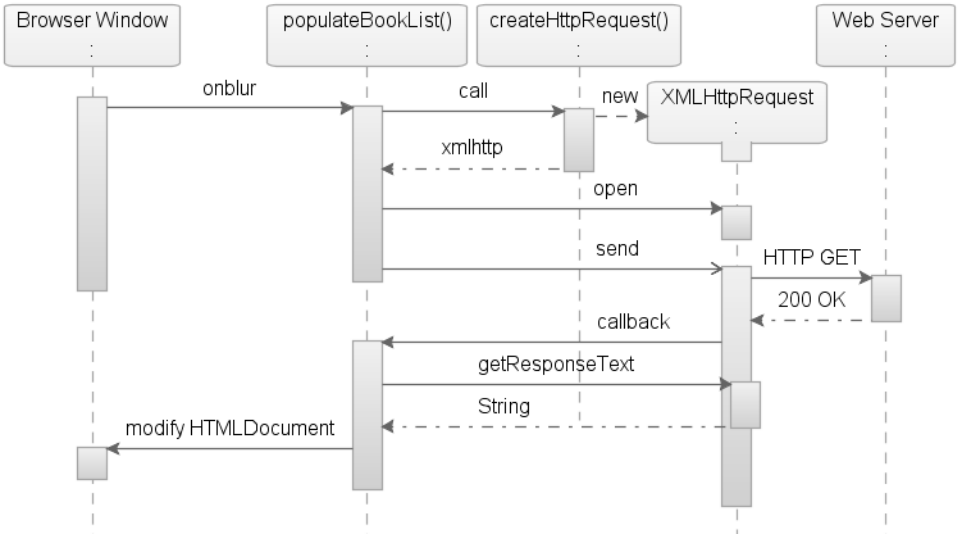

Figure 4-5. *Sequence diagram over the book filter* `XMLHttpRequest`

The `populateBookList()` function will call the `createHttpRequest()` function, which will create a new instance of the `XMLHttpRequest` object. You then use this `XMLHttpRequest` object to set the stage for your request and send the encoded data as a `url-formencoded` stream back to the server (HTTP `GET`). If the request is successful, the `XMLHttpRequest` object calls your callback function. This function will get the response text from the `XMLHttpRequest` object and use the content passed (for example, a list of books) to modify the HTML document and populate the `<select>` element with data. Code Sample 4-4 shows the actual code behind this book filter.

Code Sample 4-4. *The* `populateBookList()` *Function*

```
/**
 * Populates the select element with a list of books in a specific book category.
 *
 * @param serviceURL  the service URL for retrieving JSON data files
 * @param selectId    the id of the target select element to populate
 * @param category    the book category for the populated books
 */
function populateBookList(
  serviceURL,
  selectId ,
```

```
  category)
{
  var xmlhttp = createHttpRequest();

  // You can use any type of data source, but for the sample
  // you are going to use a simple JSON file that contains your data.
  var requestURL = serviceURL + '/booklist-' + category.toLowerCase() + '.json';
  xmlhttp.open("GET", requestURL);
  xmlhttp.onreadystatechange=function()
  {
    if (xmlhttp.readyState == 4)
    {
      if (xmlhttp.state == 200)
      {
        transferListItems(selectId, eval(xmlhttp.responseText));
      };
    };
  };
  xmlhttp.send(null);
};
```

With this code, you first create a new instance of the `XMLHttpRequest` object by calling a function called `createHttpRequest()`. You initiate your request by calling the `open("GET", requestURL)` method on the `XMLHttpRequest` object instance and passing two arguments. The `GET` string indicates the HTTP method for this request, and the `requestURL` variable represents the URL to your data source, which in this case is a simple text file. If a request is successful, the `readyState` on your `XMLHttpRequest` object is set to `4`, and the `state` is set to `200`. You use the `onreadystatechange` event handler to invoke the `transferListItems()` function when `readyState` is set to `4`, passing the `responseText` property from the `XMLHttpRequest` object. The `transferListItems()` function will take the returned string and populate the `<select>` element with data.

Creating an instance of the `XMLHttpRequest` object is simple, although as shown in Code Sample 4-5, you have a few things to consider.

Code Sample 4-5. *The* `createHttpRequest()` *Function That Creates the* `XMLHttpRequest` *Object*

```
/**
 * Creates a new XMLHttpRequest object.
 */
function createHttpRequest()
{
  var xmlhttp = null;
  if (window.ActiveXObject)
  {
    xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
  }
  else if (window.XMLHttpRequest)
```

```
  {
    xmlhttp = new XMLHttpRequest();
  }
  return xmlhttp;
};
```

Code Sample 4-5 creates the `XMLHttpRequest` object, and as in many browsers with JavaScript support, different browsers support the `XMLHttpRequest` object slightly differently. This means you need to implement support for different browsers in your `createHttpRequest()` function. For Microsoft Internet Explorer, you have to create the `XMLHttpRequest` object using `new ActiveXObject("Microsoft.XMLHTTP")`. With any browser supporting the Mozilla GRE, you can use a native call—`new XMLHttpRequest()`—to create an instance of the `XMLHttpRequest` object.

The `transferListItems()` function, shown in Code Sample 4-6, returns the data requested by the user and populates the `<select>` element with data.

Code Sample 4-6. *The* `transferListItems()` *Function That Populates the* `<select>` *Element*

```
/**
 * Transfers the list items from the JSON array
 * to options in the select element.
 *
 * @param selectId    the id of the target select element to populate
 * @param listArray   the retrieved list of books
 */
function transferListItems (
  selectId,
  listArray)
{
  var select = document.getElementById(selectId);

  // reset the select options
  select.length = 0;
  select.options[0] = new Option('[select]');

  // transfer the book list items
  for(var i=0; i < listArray.length; i++)
  {
    // create the new Option
    var option = new Option(listArray[i]);
    // add the Option onto the end of the select options list
    select.options[select.length] = option;
  };
};
```

The `transferListItems()` function takes two arguments—`selectId` and `listArray`. The `listArray` represents the data returned by your request, and `selectId` represents the `<select>`

element that is being populated with this data. Code Sample 4-7 is just showing your simple data source, in JavaScript Object Notation (JSON) syntax, so that you can replicate the sample application.

Code Sample 4-7. *Source for Your Ajax Titles*—`ajax.json`

```
['Pro JSF and Ajax: Building Rich Internet Components',
 'Foundations of Ajax',
 'Ajax Patterns and Best Practices']
```

This file contains a JavaScript expression that defines a new array of Ajax related books.

■ **Note** JSON is a lightweight data interchange format. It is based on a subset of the JavaScript programming language (standard ECMA-262, third edition). JSON is a text format that is completely language independent but uses conventions familiar to programmers of the C family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others.

Ajax Summary

You should now understand what Ajax is and be familiar with the `XMLHttpRequest` object, which is a vital part of the Ajax technique, and the lifecycle of a regular `XMLHttpRequest`. You should also have enough knowledge to be able to create simple Ajax solutions. In the coming chapters, you will dive deeper into Ajax.

Introducing Mozilla XUL

What is Mozilla XUL? Is it a crossbreed between a dinosaur and an evil Ghostbuster spirit? No, Mozilla XUL is an open source project that is known as the development platform for the Mozilla Firefox browser and Mozilla Thunderbird email client. In the following sections of this chapter, you will get a high-level overview of Mozilla XUL and its subcomponents. In 1998 the Mozilla organization (`Mozilla.org`) created an open source project called XUL, which is an extensible UI language based on XML and, as such, can leverage existing standards including XSLT, XPath, the DOM, and even Web Services (SOAP).

Using XUL, developers can build rich user interfaces that can be deployed as Web applications, as desktop applications locally, or as desktop applications on other Internet-enabled devices. XUL leverages the support of the Mozilla Gecko Runtime Environment (GRE) in order to fully provide the consumer with a rich user interface. The Firefox browser and the Thunderbird email client, as well as numerous plug-ins, are available for these clients and are two good examples of applications based on XUL and the Mozilla GRE.

One of the great features of XUL is its extensibility. Using XBL, XUL provides a declarative way to create new and extend existing XUL components. XBL can also bridge the gap between XUL and HTML, since it is not possible to embed XUL components directly into an HTML

page. The following section introduces how to build XUL applications and some of the components used when building XUL applications.

■ **Tip** An excellent sample to look at to get a feel for what is possible with XUL is the Mozilla Amazon Browser (MAB) at `http://www.faser.net/mab/`.

Building XUL Applications

The idea behind XUL is to provide a markup for building user interfaces, much like HTML, while leveraging technologies such as CSS for the look and feel and JavaScript for the event and behavior. Also, APIs are available to give developers access to read from and write to file systems over the network and give them access to Web Services. As an XML-based language, developers can also use XUL in combination with other XML languages such as XHTML and SVG. You can load an application built with XUL in three ways:

- You can load the XUL page the traditional way from the local file system.
- You can load it remotely using an HTTP URL to access content on a Web server.
- You can load it using the chrome URL provided by the Mozilla GRE.

XUL Components

XUL comes with a base set of components (see Table 4-3) that are available through the Mozilla GRE, and as such, XUL does not need to download components to draw an application in the browser. You can also design your own components with XUL; these *will* need to be downloaded upon request and cached in the browser.

MOZILLA XUL'S CHROME SYSTEM

In addition to loading files from the local file system or from a Web server, the Mozilla engine has a special way of installing and registering applications as a part of its *chrome system*. The chrome system allows developers to package applications and install them as plug-ins to clients supporting the Mozilla GRE. XUL applications deployed in this way gain read and write access to the local file system, and so on. This type of access can be hard to achieve in a traditional Web application unless the application has been signed with a digital certificate, and the end-user grants access permission.

An important distinction exists between accessing an application via an HTTP URL (`http://`) and accessing it via a chrome URL (`chrome://`). The chrome URL always refers to packages or extensions that are installed in the chrome system of the Mozilla engine. An example of an application that can be reached by a chrome URL is `chrome://browser/content/bookmarks/bookmarksManager.xul`. This chrome URL will open the Bookmarks Manager available in the Firefox browser.

Table 4-3. *Subset of Available XUL Components**

Component Name	Description
`<button>`	A button that can be clicked by the user. Event handlers can be used to trap mouse, keyboard, and other events. A button is typically rendered as a gray outset rectangle. You can specify the label of the button by using the `label` attribute or by placing content inside the button.
`<window>`	Describes the structure of a top-level window. It is the root node of a XUL document, and it is by default a horizontally oriented box. Because it is a box, it takes all the box attributes. By default, the window will have a platform-specific frame around it.
`<menubar>`	A container that usually contains menu elements. On a Mac, the menu bar is displayed along the top of the screen, and all non-menu-related elements inside the menu bar will be ignored.
`<menu>`	An element, much like a button, that is placed on a menu bar. When the user clicks the `<menu>` element, the child `<menupopup>` of the menu will be displayed. This element is also used to create submenus.
`<menupopup>`	A container used to display menus. It should be placed inside a menu, menu list, or menu-type button element. It can contain any element but usually will contain `<menuitem>` elements. It is a type of box that defaults to vertical orientation.
`<menuitem>`	A single choice in a `<menupopup>` element. It acts much like a button, but it is rendered on a menu.
`<radio>`	An element that can be turned on and off. Radio buttons are almost always grouped together in clusters. Only one radio button within the same `<radiogroup>` can be selected at a time. The user can switch which radio button is turned on by selecting it with the mouse or keyboard. Other radio buttons in the same group are turned off. A label, specified with the `label` attribute, can be added beside the radio button to indicate its function to the user.
`<radiogroup>`	A group of radio buttons. Only one radio button inside the group can be selected at a time. The radio buttons can direct either children of the `<radiogroup>` or descendants. Place the `<radiogroup>` inside a `<groupbox>` if you would like a border or caption around the group. The `<radiogroup>` defaults to vertical orientation.
`<checkbox>`	An element that can be turned on and off. The user can switch the state of the check box by selecting it with the mouse. A label, specified with the `label` attribute, may be added beside the check box to indicate to the user its function.
`<box>`	A container element that can contain any number of child elements. If the box has an `orient` attribute that is set to `horizontal`, the child elements are laid out from left to right in the order they appear in the box. If `orient` is set to `vertical`, the child elements are laid out from top to bottom. Child elements do not overlap. The default orientation is `horizontal`.
`<splitter>`	An element that should appear before or after an element inside a container. When the splitter is dragged, the sibling elements of the splitter are resized.
`<image>`	An element that displays an image, much like the HTML `<img>` element. The `src` attribute can be used to specify the URL of the image.

* *Source:* `http://xulplanet.com/references/elemref/`

We will cover the details of XBL shortly, but the sample XUL file in Code Sample 4-8 demonstrates how to embed standard, namespaced HTML elements into base XUL controls.

Code Sample 4-8. *A Simple XUL File with Embedded HTML Elements*

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css" ?>
<xul:window title="Pro JSF and AJAX: Mozilla XUL" align="start"
      xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
      xmlns:html="http://www.w3.org/1999/xhtml" >
  <xul:groupbox>
    <xul:caption label="Search" />
    <xul:hbox>
      <html:input id="find-text" />
      <xul:button label="Search" />
    </xul:hbox>
  </xul:groupbox>
</xul:window>
```

Code Sample 4-8 shows how to use a namespaced HTML input element—`<html:input id="find-text"/>`—embedded in a XUL page and mixed with regular XUL components.

To be able to deploy and run a XUL application on a remote server, the Web server needs to be configured to send files with the content type of `application/vnd.mozilla.xul+xml`. A browser that uses the Mozilla GRE (Netscape and Firefox, in other words) will use this content type to determine the markup used by the requesting application. A browser with the GRE does not use the file extension unless the file is read from the file system.

Events, State, and Data

Depending on what type of client is being developed—thick or thin—the event handling will be slightly different. This section, however, is showing XUL for Web deployment, and you use JavaScript to handle events and application logic.

Using XUL event handling is not that different from using HTML event handling. The GRE implementation supports DOM Level 2 (and partially DOM Level 3), which is virtually the same for HTML and XUL. Changes to the state and events are propagated through a range of DOM calls. XUL elements come with predefined event handlers, much like the event handlers provided with the standard HTML elements.

Code Sample 4-9 shows a simple use case where a button will launch an alert that will display the value entered by the user in an input field.

Code Sample 4-9. *A Simple Use Case of an Event and Predefined Event Handler*

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
```

```
<xul:window title="Pro JSF and AJAX : Mozilla XUL" align="start"
      xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
      xmlns:html="http://www.w3.org/1999/xhtml" >
  <xul:groupbox>
    <xul:caption label="Search" />
    <xul:hbox>
      <html:input id="find-text" />
      <xul:button label="Search"
                  oncommand="alert('Book choice: ' +
                            document.getElementById('find-text').value)" />
    </xul:hbox>
  </xul:groupbox>
</xul:window>
```

Figure 4-6 shows the aforementioned code running in Mozilla Firefox.

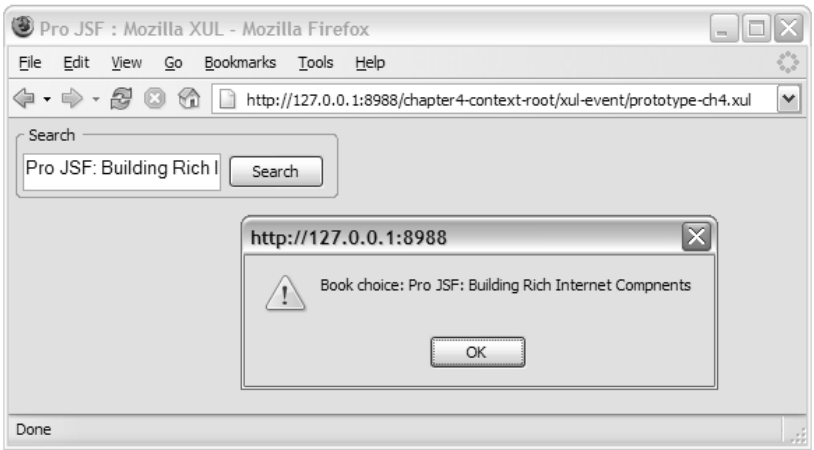

Figure 4-6. *A simple XUL file rendered in the Firefox browser*

As in HTML, developers can use JavaScript functions located in external files of the form `myScript.js`. You can access these methods and functions by using the `src` attribute on the `<script>` element or by embedding them in the page. Developers can refer to a remote server using the `http://` URL, as shown in Code Sample 4-10.

Code Sample 4-10. *Script Reference Using* `http://`

```
<script type="text/javascript" src="http://www.apress.com/projsf/js/myScript.js">
```

A large set of event handler attributes is available, and some of them work only on specific XUL/HTML elements. An example is the XUL `<window>` element that listens for DOM events (for example, `load`). Table 4-4 lists a subset of the available predefined event handlers.

Table 4-4. *Listing of Predefined Event Handlers Provided by the GRE DOM Implementation**

Event Handler	Description
`onload`	An event handler property for window loading. This event is being sent when the `window` element is finished loading and when all objects in the document are available in the DOM tree. This event handler can also be used on `image` elements.
`oncommand`	This event replaces the `onclick` event handler and is called when an element is activated. The activation can vary from element to element, but essentially it can be called from different user interactions such as clicking and hitting the Enter key or shortcut keys, which is not the case for the `onclick` event handler.
`onblur`	The blur event is raised when an element loses focus.
`onfocus`	The opposite of the `onblur` event. This event is raised when an element gets focus.

* *Source:* `http://www.xulplanet.com`

Creating Custom XUL Components Using XBL

To fully understand how Mozilla XUL can provide a mechanism for JSF to use XUL as a rendering technology, you have to understand XBL. XBL is an XML-based language that allows developers to extend XUL and add "custom" components to the already extensive set of XUL elements. In XUL, developers can change the look and feel using CSS and can attach skins, but they have no way to change the behavior of XUL elements in XUL itself.

To do this, developers have to use another language—XBL. Developers can look at XUL as the "implementation" that comes with a set of base components or as tag libraries that can be used to build a user interface, much like the JSF Reference Implementation. XBL is the language developers use to extend XUL components and enable integration with HTML, similar to how Java is used to extend JSF components.

Creating XBL Bindings

XBL is an XML language, and a file created with XBL contains a set of bindings. These bindings each describe the behavior of a XUL component. Besides describing the behavior, these bindings also describe the XUL elements that make up the component along with properties and methods of the component. In Code Sample 4-11, the root shows that the `<bindings>` element contains one `<binding>` element.

Code Sample 4-11. *An XBL File Containing One Binding*—projsf-bindings.xml

```
<?xml version="1.0"?>
<xbl:bindings xmlns:xbl="http://www.mozilla.org/xbl"
      xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
      xmlns:html="http://www.w3.org/1999/xhtml" >

  <xbl:binding id="welcome" >
    <xbl:content>
      <xul:text value="Welcome, " />
      <xul:text value="Guest" xbl:inherits="value=name" />
      <xul:text value="!" />
    </xbl:content>
  </xbl:binding>
</xbl:bindings>
```

A <bindings> element can contain an infinite number of <binding> elements. The namespace in the <bindings> element defines what syntax will be used, and in Code Sample 4-11 it is XBL—xmlns=http://www.mozilla.org/xbl. The file also contains some XUL elements: <xul:text/>. This is extremely useful to simplify development by encapsulating several components that later can be referred to as one component.

The xbl:inherits attribute on one of the <xul:text> elements allows the <xul:text> element to inherit values from the bound element by defining a variable name and, in this case, assigning it to the value attribute. If no value is defined in the bound element in the page using this component, the text field will default to Guest.

The id attribute on the <xbl:binding> element (in Code Sample 4-11, welcome) will identify the binding.

Using the XBL Bindings

To attach an XBL component or behavior to a XUL application, XUL uses CSS. Using CSS, a developer can assign a binding to an element by setting the -moz-binding property to a URI pointing to the XBL document.

■ **Note** Netscape has submitted a proposal to the W3C to define how to attach custom behavior to an HTML element in "A Modular Way of Defining Behavior for XML and HTML" (http://www.w3.org/TR/NOTE-AS).

Code Sample 4-12 illustrates a CSS file that attaches a binding to the <pro:welcome> element.

Code Sample 4-12. *A Sample CSS File That Has the* `-moz-binding` *Property Set*—`projsf.css`

```
@namespace pro url('http://projsf.apress.com/tags');

pro|welcome
{
  -moz-binding: url('projsf-bindings.xml#welcome');
}
```

In Code Sample 4-12, the selector has the `-moz-binding` set to point to an XBL file named `projsf-bindings.xml` and uses `#welcome` to refer to a specific binding in the XBL file. This is similar to how anchors are referenced in HTML files.

■ **Note** To provide a consistent sample tag throughout the chapter's samples, Code Sample 4-12 uses CSS3 standard syntax to simulate the sample element—`<pro:welcome>`.

If the binding `id` is omitted when assigned to an element, XUL will default to the first binding listed. In Code Sample 4-12, the `welcome` binding has been declared as the `id`, and the element that has been assigned this binding is `<pro:welcome>`.

In Code Sample 4-13, the `projsf-bindings.css` style sheet has been attached to the XUL document, and two elements (`<pro:welcome id="guest" />` and `<pro:welcome id="duke" name="Duke" />`) are inserted in the page. The first element displays a welcoming greeting for the specified user, "Duke". The second element displays the "Welcome, " string defined in the XBL file plus a default value user, "Guest". One of the cool features of using encapsulation of behavior, as provided by XBL, is that it creates a document tree within the scope of the custom component that is separate from the XUL page. What this means is that the content of the XBL component is not "exploded" into the main document, losing encapsulation. Figure 4-7 shows the DOM using a DOM inspector.

Code Sample 4-13. *A Sample HTML File with XUL Components*—`prototype-ch4.xul`

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css" ?>
<?xml-stylesheet href="projsf-bindings.css" type="text/css" ?>
<xul:window title="Pro JSF : Mozilla XBL" align="start"
      xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
      xmlns:pro="http://projsf.apress.com/tags" >
  <xul:groupbox>
    <xul:caption label="Greeting" />
    <pro:welcome id="duke" name="Duke" />
    <pro:welcome id="guest" />
  </xul:groupbox>
</xul:window>
```

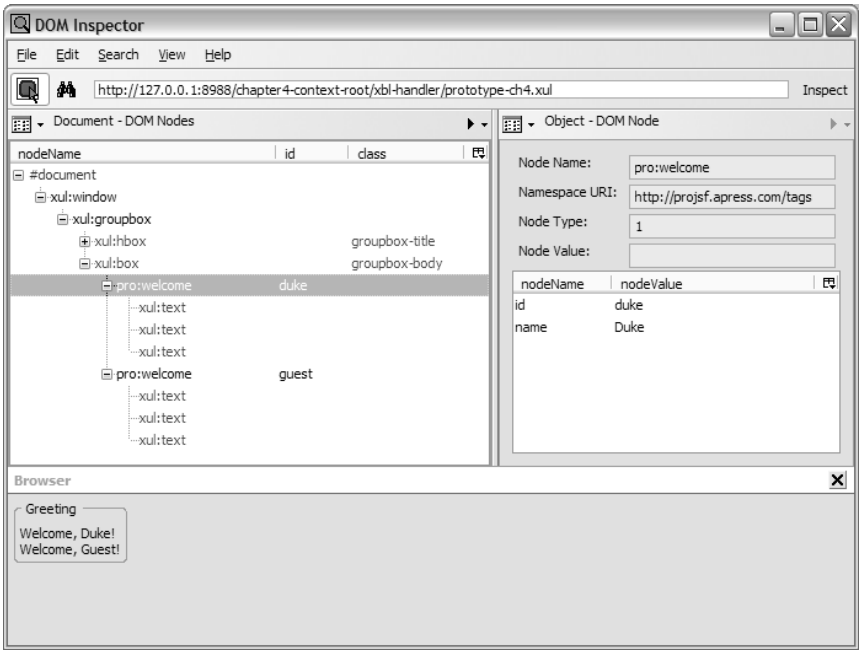

Figure 4-7. *A page's DOM tree with an XBL component*

The direct benefits of encapsulation are that the component author has full control over the behavior and look and feel and that the component is not exposing internal implementation details. In Figure 4-7, the nested `<xul:text>` elements are shown in the DOM inspector but never exposed in the actual main document.

Extending the XBL Bindings

Apart from creating a widget that is a collection of one or more XUL elements (as shown in the previous sections), you can also use XBL to add new properties and methods. XBL has three types of items that can be added to the binding—`fields`, `properties`, and `methods`:

- The `field` item is a simple container that can store a value, which can be retrieved and set.
- The `property` item is slightly more complex and is used to validate values stored in fields or values retrieved from XBL-defined element attributes. Since the `property` item cannot hold a value, you have no way to set a value directly on a `property` item without using the `onset` handler or the `onget` handler. Using these handlers, you can perform precalculation or validation of the value retrieved or modified.
- `Methods` are object functions, such as `window.open()`, that allow developers to add custom functions to custom elements.

In Code Sample 4-14, these three items are defined in an `<implementation>` element that is a child element of the `<binding>` element.

Code Sample 4-14. *Adding Properties and Methods*—pro-bindings.xml

```
<?xml version="1.0"?>
<xbl:bindings xmlns:xbl="http://www.mozilla.org/xbl"
      xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
      xmlns:html="http://www.w3.org/1999/xhtml" >

  <xbl:binding id="welcome" >
    <xbl:content>
      <xul:text id="greeting" value="Welcome, " />
      <xul:text value="Guest" xbl:inherits="value=name" />
      <xul:text value="!" />
    </xbl:content>
    <xbl:implementation>
      <xbl:constructor>
      <![CDATA[
        this._greetingNode = document.getElementById('greeting');
      ]]>
      </xbl:constructor>
      <xbl:property name="greeting"
                    onget="return this._greetingNode.getAttribute('value');"
                    onset="this._greetingNode.setAttribute('value', val);" />
    </xbl:implementation>
  </xbl:binding>
</xbl:bindings>
```

In Code Sample 4-14, you have added one method and one property. The method used in Code Sample 4-14 is a special method supported by XBL called `constructor`. A `constructor` is called whenever the `binding` is attached to an element. It is used to initialize the content such as loading preferences or setting the default values of fields. The `property` has been defined with an `onget` handler and an `onset` handler, which get and set the `value` attribute on your `<pro:welcome>` tag. To access these properties and call methods on the custom element, developers can use the `getElementById()` function. In Figure 4-8, a XUL button is added that triggers the `oncommand` event handler.

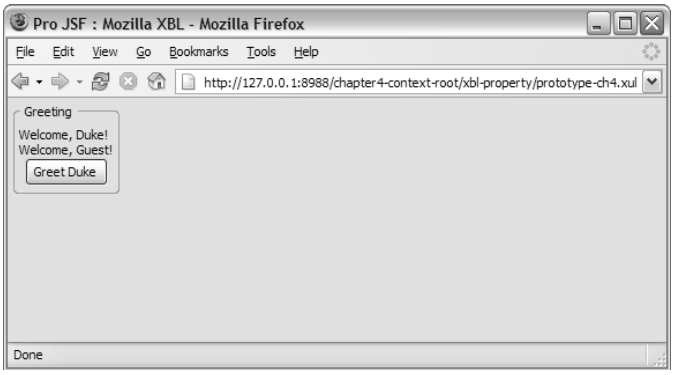

Figure 4-8. *A page using the welcome XBL component*

When the button Greet Duke is clicked, the text of the first <pro:welcome> tag changes and displays a new welcome message instead of the default message defined earlier in the projsf-bindings.xml file. Code Sample 4-15 shows the code behind this page.

Code Sample 4-15. *A Sample XUL File with XBL Components*—prototype-ch4.xul

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css" ?>
<?xml-stylesheet href="projsf-bindings.css" type="text/css" ?>
<xul:window title="Pro JSF : Mozilla XBL" align="start"
      xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
      xmlns:pro="http://projsf.apress.com/tags" >
  <xul:groupbox>
    <xul:caption label="Greeting" />
    <pro:welcome id="duke" name="Duke" />
    <pro:welcome id="guest" />
    <xul:button label="Greet Duke"
                oncommand="var duke = document.getElementById('duke');
                           duke.greeting = 'Howdy, ';" />
  </xul:groupbox>
</xul:window>
```

In Code Sample 4-15, a XUL button has been added that triggers the oncommand event handler. The oncommand event handler will execute the script encapsulated—var duke = document.getElementById('duke'); duke.greeting = 'Howdy, ';. This will set the value of the XUL element with the identifier greeting defined in your binding to "Howdy, " instead of the default greeting "Welcome, " causing Duke's greeting to change to "Howdy, Duke!" whereas the Guest greeting remains unchanged.

Event Handling and XBL Bindings

In XBL, developers can add event handlers directly to the XUL elements listed as children to the content element (for example, <xul:button label="Press me!" oncommand= "alert('welcome')" />). Sometimes developers need to add an event handler for all the child elements in the content element.

In XBL, you can do this by adding a <handler> element. The <handler> element is a child of the <handlers> element, and it can contain one or more event handlers. Each handler defines the action that will be taken for a particular event in the scope of the binding in which it is defined. If an event is not captured, it will just pass to the inner elements.

In Code Sample 4-15, you had a button and an event handler in the actual page source. Code Sample 4-16 shows how you can move this functionality into an XBL component.

Code Sample 4-16. *Adding Event Handlers*—projsf-bindings.xml

```
<?xml version="1.0"?>
<xbl:bindings xmlns:xbl="http://www.mozilla.org/xbl"
      xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
      xmlns:html="http://www.w3.org/1999/xhtml" >

  <xbl:binding id="welcome" >
    <xbl:content>
```

```
      <xul:text value="Welcome, " />
      <xul:text value="Guest" xbl:inherits="value=name" />
      <xul:text value="!" />
    </xbl:content>
    <xbl:handlers>
      <xbl:handler event="click" >
        if (this.hasAttribute('name'))
          alert('Nice to see you again, ' + this.getAttribute('name') + '.');
      </xbl:handler>
    </xbl:handlers>
  </xbl:binding>
</xbl:bindings>
```

In Code Sample 4-16, one handler has been added to capture all click events in the context of the `welcome` binding. The handler will display an alert only if the attribute name has been set on the `<pro:welcome>` tag. You now have a simple but well-defined and encapsulated XUL component. Code Sample 4-17 shows a simple XUL page that is using this new `<pro:welcome>` tag.

Code Sample 4-17. *A Simple XUL Page Using an XBL Binding with Attached Event Handler*

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css" ?>
<?xml-stylesheet href="projsf-bindings.css" type="text/css" ?>
<xul:window align="start"
      xmlns:xul="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
      xmlns:pro="http://projsf.apress.com/tags" >
  <xul:groupbox>
    <xul:caption label="Greeting" />
    <pro:welcome id="duke" name="Duke" />
    <pro:welcome id="guest" />
  </xul:groupbox>
</xul:window>
```

In this page only one `<pro:welcome>` tag has the `name` attribute defined. So, when the page is launched in a browser (a Mozilla GRE-compliant browser), the click event will launch an alert only when the "Welcome, Duke!" text is clicked, as shown in Figure 4-9.

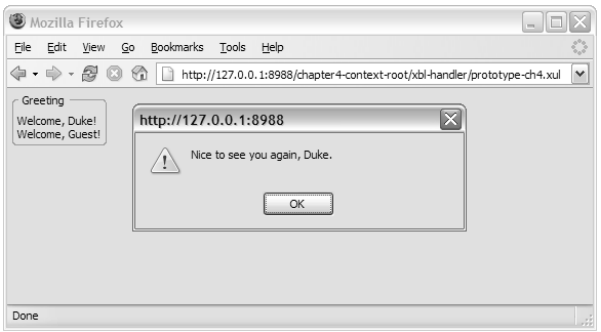

Figure 4-9. *Simple XUL page using a custom XBL binding with attached event handler*