CHAPTER 10

Ajax and Java Web Services

In this chapter, I examine how Java Web Services can be used to support *Ajax* clients. Ajax, or Asynchronous JavaScript and XML, is a programming technique that enables you to create user interfaces for a Web browser that behave more like a local, stand-alone application than a collection of HTML pages.

Ajax is a good fit with Java Web Services. Using these two technologies together enables you to publish software components as services (via JAX-WS) and create great browser-based user interfaces on top of them (via Ajax). The entire application can then be packaged as an EAR or WAR and deployed on a Java EE application server.

To demonstrate this capability, I pick up here where I left off at the end of Chapter 9. In that chapter, I showed you how to build an online shopping application, SOAShopper, which can search across multiple Web-service-enabled sites (i.e., eBay, Yahoo! Shopping, and Amazon). In this chapter, I show how you can develop an Ajax front-end to SOAShopper. In particular, the code examined in this chapter demonstrates how to write an Ajax application that consumes RESTful Java Web Services endpoints.

In the second half of this chapter, I review the JavaScript code that implements the SOAShopper Ajax front-end in quite a bit of detail. For those of you who are familiar with Web front-end coding and JavaScript, this detail may seem tedious. I include it because my assumption is that many readers of this book are server-side Java programmers who do not usually do a lot of JavaScript development and, therefore, might be interested in the detailed code explanation.

10.1 Quick Overview of Ajax

Ajax is a well-documented technology, and my purpose here is not to write a detailed tutorial on Ajax programming.[1] However, I do want to go over some of the basics to set the stage for a discussion of the SOAShopper front-end and how it interacts with Java EE.

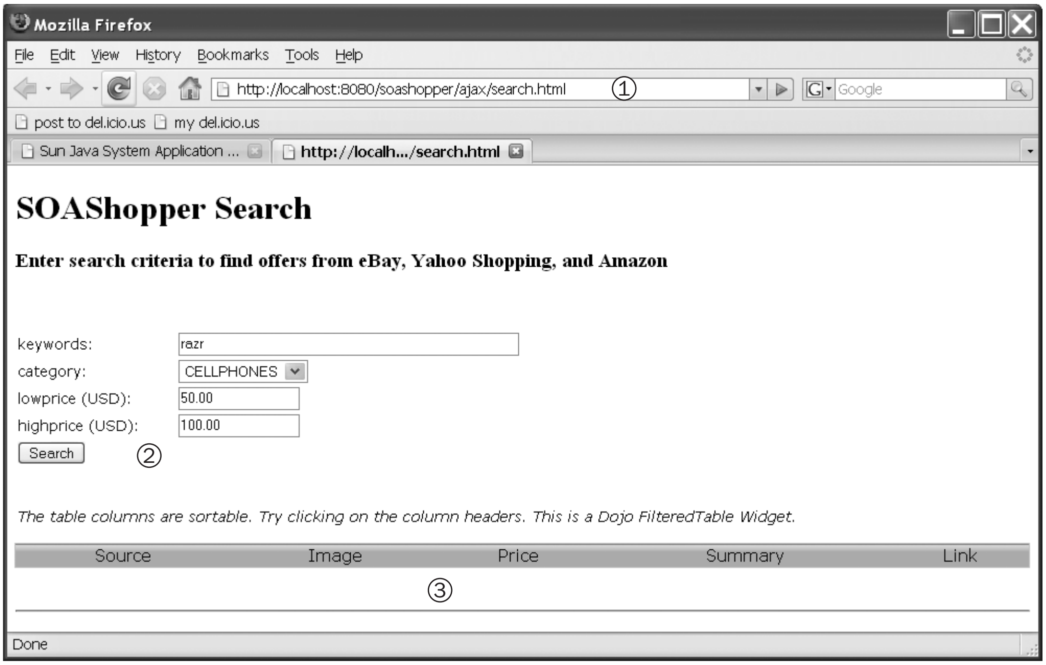
As many of you know, the major benefit of Ajax is that it allows a browser-based application to avoid the need for full-page refreshes each time new data is retrieved from the server. Ajax programmers use the JavaScript type `XMLHttpRequest` to exchange data with the server behind the scenes (i.e., without having to reload the entire HTML page being displayed by the browser). When new data (usually in XML format) is received by an `XMLHttpRequest` instance, JavaScript is used to update the *DOM* structure of the HTML page (e.g., inserting some rows in a table) without rebuilding the entire HTML page in memory.

To see what that means in practice, I walk you through some screen shots from the SOAShopper front-end. Then, in the rest of this chapter, I will show you how to write the code behind these screen shots.

If you build and deploy the SOAShopper application on your local machine[2] and point your browser to `http://<your-host>:<your-port>/shoashopper/ajax/search.html`, you should see something similar to what appears in Figure 10–1. This is the initial search screen for SOAShopper. The three labeled items in this figure are worth pointing out for discussion:

1. The URL where the application resides remains constant throughout its use. The search is performed and results are displayed without loading a new page. This is implemented by using JavaScript that updates the DOM residing in the browser's memory.
2. This search page offers you four search parameters: a set of keywords; a category to search; a low price; and a high price. These parameters correspond to the parameters supported by the SOAShopper `offerSearch` REST endpoint discussed in Chapter 9, Section 9.3 (see Figure 9–2). This search page contains JavaScript that converts these parameters into a query string that an `XMLHttpRequest` instance uses to invoke the `offerSearch` endpoint.

1. For a good introduction to Ajax, I recommend "Ajax in Action" [AIA].
2. For instructions, see Appendix B, Section B.9.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/soashopper/ajax/search.html
①
Google
post to del.icio.us
my del.icio.us
Sun Java System Application ...
http://localh.../search.html
Search
②
③
Done

SOAShopper Search

Enter search criteria to find offers from eBay, Yahoo Shopping, and Amazon

keywords:	razr
category:	CELLPHONES
lowprice (USD):	50.00
highprice (USD):	100.00

The table columns are sortable. Try clicking on the column headers. This is a Dojo FilteredTable Widget.

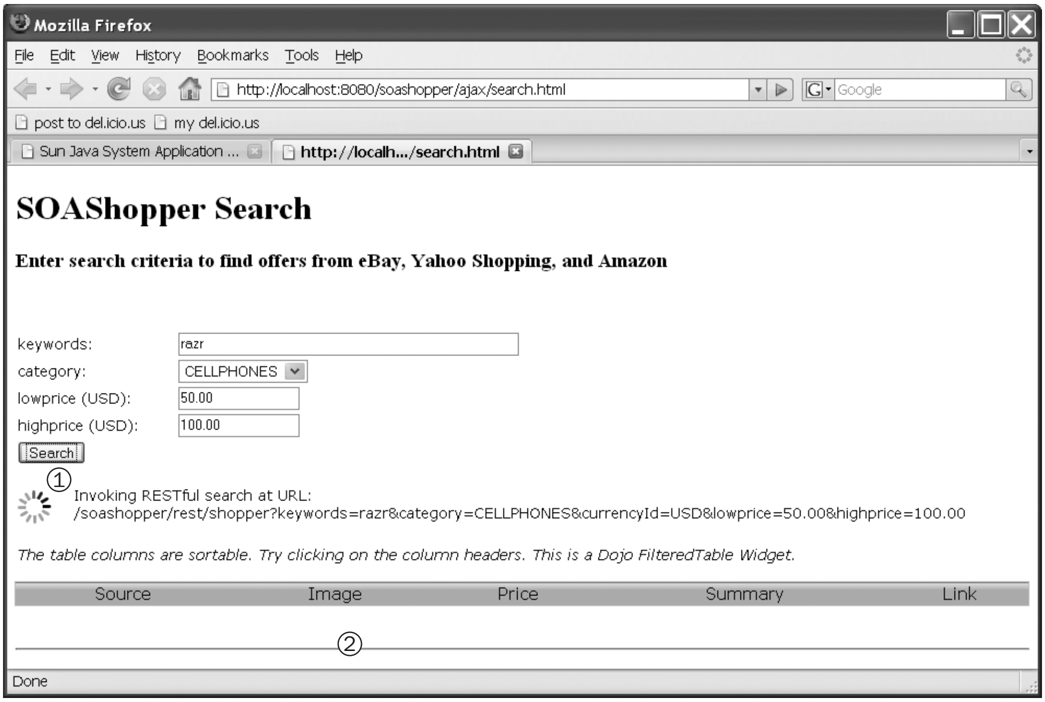
Source	Image	Price	Summary	Link

Figure 10–1 The initial SOAShopper search screen.

3. At the bottom of Figure 10–1 appear some column headings (i.e., Source, Image, Price, Summary) for an empty table. Once a search is performed and the `XMLHttpRequest` has received the results, a JavaScript function contained in this page processes those results and loads them into the table. This table is implemented using the Dojo Foundation's [DOJO] `FilteredTable` widget.

As you can see from Figure 10–1, a user has entered some criteria for a search. The keywords value is "razr." The search category is CELLPHONES and the price range is $50.00–100.00. Figure 10–2 shows what happens to the screen when the user clicks on the Search button. The search takes a while to run (sometimes as long as a minute). This is not because the Java EE 5 application server is slow or because the JavaScript in the Web page is slow. Rather, it is because the shopping sites being searched (particularly eBay) can take quite a while to respond. To handle this, Ajax techniques are used to update the interface and let the user know the application is not broken.

There are two items labeled in Figure 10–2 that I want to point out:


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/soashopper/ajax/search.html
post to del.icio.us
my del.icio.us
Sun Java System Application ...
http://localh.../search.html
Search
①
②
Done

SOAShopper Search

Enter search criteria to find offers from eBay, Yahoo Shopping, and Amazon

keywords: razr
category: CELLPHONES
lowprice (USD): 50.00
highprice (USD): 100.00

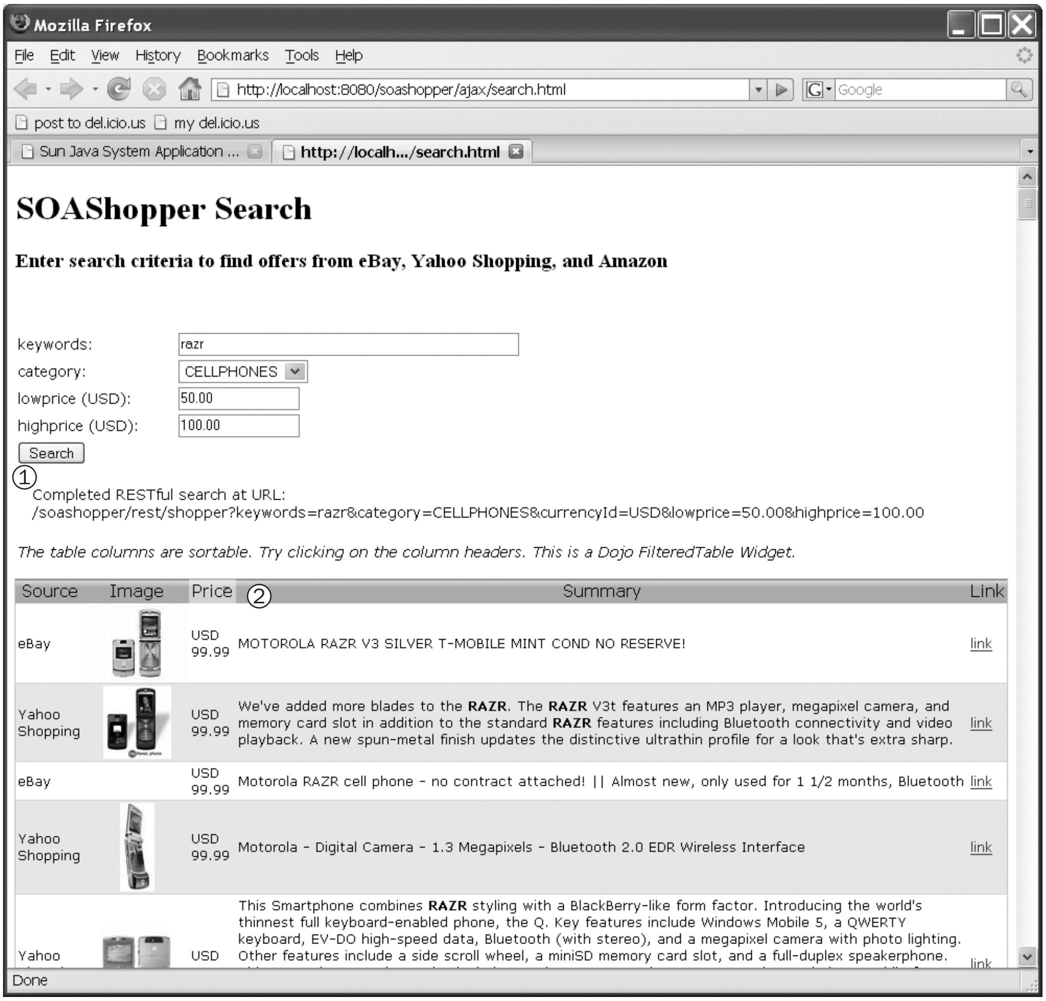
Invoking RESTful search at URL:
/soashopper/rest/shopper?keywords=razr&category=CELLPHONES&currencyId=USD&lowprice=50.00&highprice=100.00

The table columns are sortable. Try clicking on the column headers. This is a Dojo FilteredTable Widget.

Source	Image	Price	Summary	Link

Figure 10–2 Screen shot showing asynchronous processing in progress.

1. First, notice that an icon and some text have appeared below the Search button. The icon is actually an animated GIF that indicates the application is working to retrieve data from the server. The text shows us the URL of the REST endpoint from which the data has been requested: `/soashopper/rest/shopper?keywords=razr&category=CELLPHONES&currencyId=USD&lowprice=50.00&highprice=100.00`. This is the URL and query string structure that are used in Chapter 9, Section 9.3, for the SOAShopper REST endpoint. This icon and message appear while the `XMLHttpRequest` request is happening asynchronously. The `search.html` page has not been reloaded either. Rather, the DOM representation of `search.html` that was loaded by the Web browser (Firefox in this case) has been changed by a JavaScript function that inserted the animated GIF and text into the appropriate place.
2. The search results table is still empty because the asynchronous request for data from the SOAShopper REST endpoint has not yet completed.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/soashopper/ajax/search.html
Google
post to del.icio.us
my del.icio.us
Sun Java System Application ...
http://localh.../search.html
Done

SOAShopper Search

Enter search criteria to find offers from eBay, Yahoo Shopping, and Amazon

keywords:	razr
category:	CELLPHONES
lowprice (USD):	50.00
highprice (USD):	100.00

Search

①

Completed RESTful search at URL:
/soashopper/rest/shopper?keywords=razr&category=CELLPHONES&currencyId=USD&lowprice=50.00&highprice=100.00

The table columns are sortable. Try clicking on the column headers. This is a Dojo FilteredTable Widget.

②

Source	Image	Price	Summary	Link
eBay		USD 99.99	MOTOROLA RAZR V3 SILVER T-MOBILE MINT COND NO RESERVE!	link
Yahoo Shopping		USD 99.99	We've added more blades to the **RAZR**. The **RAZR** V3t features an MP3 player, megapixel camera, and memory card slot in addition to the standard **RAZR** features including Bluetooth connectivity and video playback. A new spun-metal finish updates the distinctive ultrathin profile for a look that's extra sharp.	link
eBay		USD 99.99	Motorola RAZR cell phone - no contract attached! || Almost new, only used for 1 1/2 months, Bluetooth	link
Yahoo Shopping		USD 99.99	Motorola - Digital Camera - 1.3 Megapixels - Bluetooth 2.0 EDR Wireless Interface	link
Yahoo		USD	This Smartphone combines **RAZR** styling with a BlackBerry-like form factor. Introducing the world's thinnest full keyboard-enabled phone, the Q. Key features include Windows Mobile 5, a QWERTY keyboard, EV-DO high-speed data, Bluetooth (with stereo), and a megapixel camera with photo lighting. Other features include a side scroll wheel, a miniSD memory card slot, and a full-duplex speakerphone.	link

Figure 10–3 Screen shot showing search results displayed in the Dojo table widget.

Figure 10–3 shows the appearance of the SOAShopper search page after the search results have been returned from the server. At this point, the `XmlHttpRequest` object has received the search data from the REST endpoint and invoked a JavaScript function to load that data into the results table. Two other items, labeled in the figure, are worth pointing out:

1. The animated GIF has disappeared and the text below the Search button has changed to indicate that the results have been received.

2. The search results table has been populated. As you can see, these results included a list of cell phones. The leftmost column, "Source," indicates which site the offer came from (Figure 10–3 shows results from eBay and Yahoo! Shopping). A thumbnail image, if available, is displayed, along with the price and summary. The rightmost column contains a link to the page containing the offer. Clicking this link will take you to a page where you can purchase the cell phone that is listed.

One cool feature of the Dojo table widget used here is that the results can be sorted by column. Figure 10–3 shows the results sorted by price from high to low. Hence, the $99.99 phone appears at the top of the list.

That wraps up a quick overview of the SOAShopper search interface. In the next section, I look at the working relationship between Ajax and Java EE that has been demonstrated in these screen shots.

10.2 Ajax Together with Java EE Web Services

Figure 10–4 shows the interrelationship between the Ajax front-end illustrated by screen shots in Section 10.1, and the SOAShopper application described in Chapter 9. The numbered items in this figure trace the flow of events that implement the search:

1. First, there is a JavaScript function, `retrieveURL (url)`, contained in the HTML page (`search.html`), that has been loaded by the browser. When the Search button is pressed, this function is invoked with the parameter `url` set to the value of the REST endpoint with the query string determined by the search parameters.
2. Next, the `showSearchingMsg()` function is invoked to display the animated GIF and message illustrated in Figure 10–2.
3. Then, the `retrieveURL()` function instantiates an `XMLHttpRequest` object, which invokes the SOAShopper's REST endpoint asynchronously. It also configures a handler (the `processStateChange()` function used in step 7) on the `XMLHttpRequest` object.
4. The `XMLHttpRequest` object makes an HTTP GET request to the SOAShopper REST endpoint. This is an asynchronous request, and the main thread of execution returns to handle any other interactions that may occur while the search is going on.
5. Meanwhile, inside the Java EE container that has deployed the SOAShopper REST endpoint, processing of the `XMLHttpRequest`'s

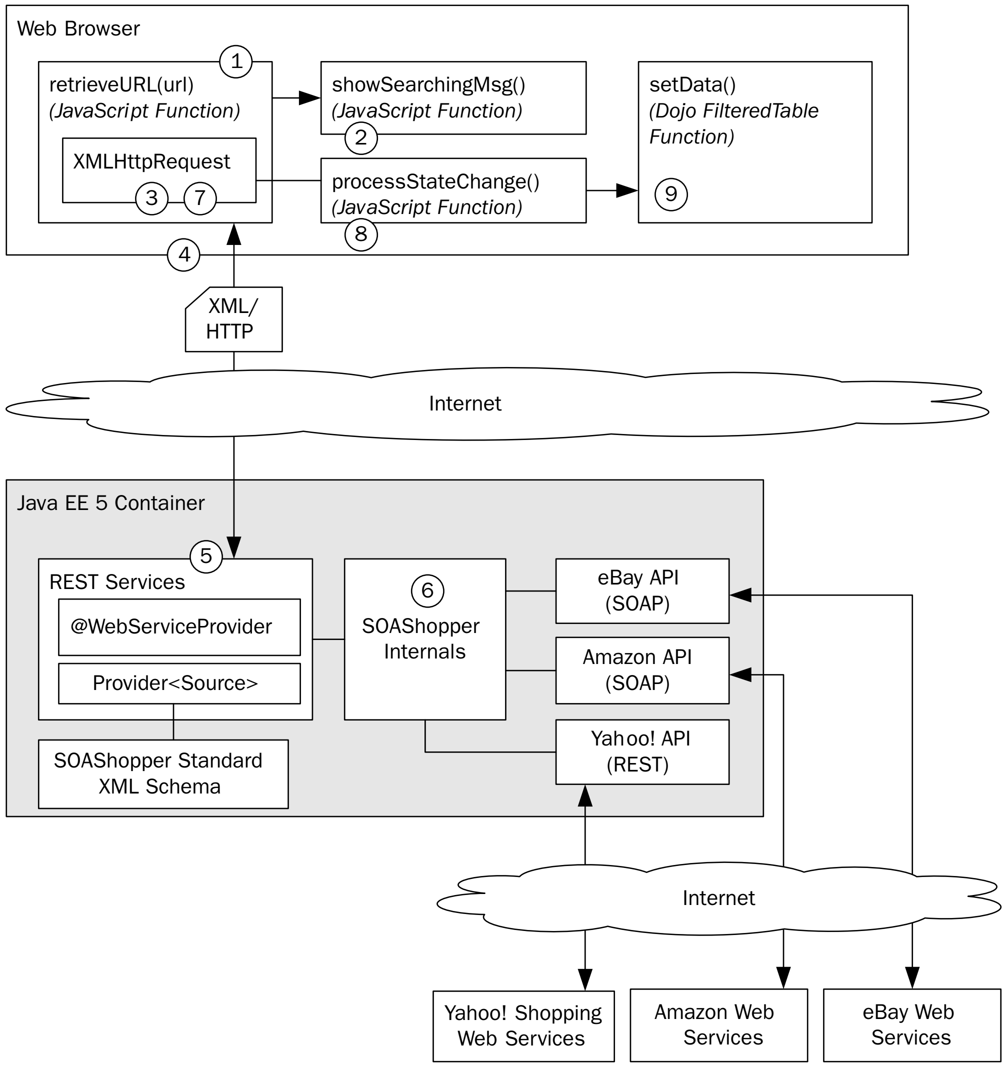


Figure 10–4 A typical Ajax client invokes REST endpoints asynchronously.

HTTP GET request is taking place. As described in Chapter 9, Section 9.3, query parameters are parsed from the query string and passed to the

6. SOAShopper internals. SOAShopper then translates the search request into the appropriate form for each online shopping service (eBay, Amazon, and Yahoo! Shopping), gets the results, and packages them into an XML document compliant with the `retail.xsd` schema

(see Chapter 9, Example 9–4, from Section 9.2). The XML document is then sent back to the `XMLHttpRequest` object over the HTTP response to its original GET request.

7. When the `XMLHttpRequest`'s state changes, indicating that the search response has been received, the `processStateChange()` handler (set in step 2) gets invoked.
8. The `processStateChange()` handler calls other functions that (i) change the message to indicate the search has finished, and (ii) process and format the XML data received from SOAShopper so that it can be displayed.
9. Lastly, the Dojo table widget's `setData()` function is invoked to display the search results.

One other relationship between the Ajax application running in the Web browser and the Java EE container is not shown in Figure 10–4. The Web container on the Java EE side also acts as a Web server hosting the Ajax application. So, the `search.html` page that contains the Ajax code is served by the Java EE container as well.

In the next section, I walk through the JavaScript code that implements steps 1–9. My goal is to give you a detailed understanding of how to implement an Ajax application that can interact with your Java EE REST endpoints.

10.3 Sample Code: An Ajax Front-End for SOAShopper

The code example discussion starts with the JavaScript function `retrieveURL()`, shown as step 1 in Figure 10–4. As you can see in Example 10–1, the first thing this code does is invoke the `showSearchingMsg()` function to display the message on the browser indicating that the search is underway.

Example 10–1 The `retrieveURL()` JavaScript Function Uses an `XMLHttpRequest` Object to Asynchronously Invoke the SOAShopper REST Endpoint

```
function retrieveURL(url) {
  restURL = url;
  showSearchingMsg(restURL);
  if (window.XMLHttpRequest) { // Non-IE browsers
    req = new XMLHttpRequest();
```

```
      req.onreadystatechange = processStateChange;
      try {
        req.open("GET", url, true);
        req.setRequestHeader('Content-type','text/xml');
      } catch (e) {
        alert(e);
      }
      req.send(null);
    } else if (window.ActiveXObject) { // IE
      req = new ActiveXObject("Microsoft.XMLHTTP");
      if (req) {
        req.onreadystatechange = processStateChange;
        try {
          req.open("GET", url, true);
          req.setRequestHeader('Content-type','text/xml');
        } catch (e) {
          alert(e);
        }
        req.send();
      }
    }
  }
```

book-code/chap09/soashopper/soashopper-ajax/src/main/webapp
  /search.html

Next, the code instantiates the `XMLHttpRequest` object and stores it in `the req` variable. Actually, the code needs to handle two cases for Microsoft and non-Microsoft browsers. In a non-Microsoft browser, it is created using:

```
new XMLHttpRequest()
```

However, in Internet Explorer, it is creating using:

```
new ActiveXObject("Microsoft.XMLHTTP")
```

Once the `XMLHttpRequest` object is instantiated, an HTTP GET request is made to the specified `url` parameter using `req.open()` and `req.send()` functions. The `setRequestHeader()` call is made to add the:

```
Content-type: text/xml
```

HTTP request header to the GET request. Strictly speaking, this should not be necessary. However, some REST endpoints require that the `Content-type` header be configured this way. For example, early versions of JAX-WS (including the first production release of GlassFish), required it.

Example 10–2 shows the code that implements step 2 from Figure 10–4. This code manipulates the Web browser's DOM representation of the `search.html` document.

Example 10–2 The JavaScript Method `showSearchingMsg()` Updates the Web Browser's DOM to Display an Animated GIF and Text Message

```
function showSearchingMsg(url) {
  var messageTDElt = document.getElementById('searchingMessageId');
  var loadingTDElt = document.getElementById('loadingId');
  loadingTDElt.setAttribute('width','50');
  var loadingNode = document.createElement('img');
  loadingNode.setAttribute('src','images/bigrotation2.gif');
  loadingNode.setAttribute('style', 'margin-right: 6px; margin-top: 5px;');
  var existingLoadingNode = loadingTDElt.firstChild;
  if (existingLoadingNode) {
    loadingTDElt.removeChild(existingLoadingNode);
  }
  loadingTDElt.appendChild(loadingNode);
  var msg = "Invoking RESTful search at URL: " + url;
  var msgNode = document.createTextNode(msg);
  var existingMsg = messageTDElt.firstChild
  if (existingMsg) {
    messageTDElt.removeChild(existingMsg);
  }
  messageTDElt.appendChild(msgNode);
}
```

`book-code/chap09/soashopper/soashopper-ajax/src/main/webapp/search.html`

First, this code gets a reference to a DOM element (stored in `messageTDElt`) where the text message should be displayed. The ID, `'searchingMessageId'`, refers to a cell in a table, halfway down the page. That cell is empty when the `search.html` page is loaded. However, the code here—in particular, the last line:

```
messageTDElt.appendChild(msgNode)
```

places the text "Invoking RESTful search at URL: ..." in that cell. Similarly, other parts of this code place an animated GIF reference (i.e., `images/bigrotation2.gif`) into another cell with the ID `'loadingID'`.

If you have done server-side Java DOM programming,[3] this type of HTML DOM programming in JavaScript should make sense. If you haven't seen any kind of DOM programming before, you might want to look at "Ajax in Action" [AIA] Chapter 2 for an introduction to manipulating HTML DOM.

Getting back to the HTTP GET request issued by the `XMLHttpRequest` object, this request is received by the Java EE container where SOAShopper is deployed—in particular, the request handled by the JAX-WS runtime where it ends up calling the `ShopperServiceRESTImp.invoke()` method,[4] which has been deployed at the endpoint invoked by the Ajax application. In Figure 10–4, this part of the process is labeled step 5. This Web service method, `ShopperServiceRESTImp.invoke()`, in turn invokes the SOAShopper API shown in Example 10–3.

Example 10–3 The Java Method `offerSearch` Is Bound to the REST Endpoint by JAX-WS (The Query String Parameters from the Browser's `XMLHttpRequest` Request End Up Getting Mapped to the Parameters of This Method)[5]

```
public interface ShopperServiceREST {

  public OfferList offerSearch(String keywords, String category,
      String currencyId, Double lowprice, Double highprice);

}
```

```
book-code/chap09/soashopper/soashopper-services-rest/src/main/java/com/javector
  /soashopper/endpoint/rest/ShopperServiceREST.java
```

At this point, the server-side SOAShopper application does the search of eBay, Yahoo! Shopping, and Amazon. This is step 6 in Figure 10–4. The internals of SOAShopper are described in detail in Chapter 9.

3. See, for example, the programming for WSDL processing and XML validation discussed in Chapter 7, Section 7.5—particularly Example 7–10.

4. See Chapter 9, Section 9.3, Example 9–11.

5. See Chapter 9, Section 9.3.

Of interest here, from the Ajax perspective, is what happens when the server-side SOAShopper application returns. As indicated by step 7 in Figure 10–4, a handler function—`processStateChange()`—is invoked.

Example 10–4 The JavaScript Function `processStateChange()` Is Invoked When the Asynchronous `XMLHttpRequest.send()` Function Returns (If the REST Query Returns "200 OK", the `processXML()` Function Is Invoked to Display the Search Results)

```
function processStateChange() {
  if (req.readyState == 4) { // Complete
    showFinishedMsg(restURL);
    if (req.status == 200) { // OK response
      processXML(req.responseXML);
    } else {
      alert("Problem invoking REST endpoint: " + restURL + " : "
        + req.status + " " + req.statusText);
    }
  }
}
```

`book-code/chap09/soashopper/soashopper-ajax/src/main/webapp`
`/search.html`

Example 10–4 shows the code for that handler function. It simply checks that an HTTP response code 200 was received (indicating success) and then invokes the `processXML()` function. For code clarity, it makes sense to keep such a handler function as simple as possible and organize the real work in another function. If the HTTP response is not 200 (indicating a problem), the code here simply sends an alert message. In a real production application, some diagnostics would take place together with an attempt to recover from the failure and maybe reissue the HTTP request.

Supposing that the HTTP response code is 200, the next step in this process is to parse the XML document returned by the SOAShopper service. As indicated by Example 10–3, the return type of the SOAShopper API is `OfferList`. `OfferList` is a JAXB schema-generated Java class compiled from the `retail:offerList` schema element in the `retail.xsd` schema shown in Example 10–5. This is the schema referenced in the REST endpoint documentation from Chapter 9.[6]

6. See Chapter 9, Section 9.3, Figure 9-2.

Example 10–5 The XML Schema Definition for the XML Document Received by the Ajax Application from the SOAShopper REST Endpoint

```
<xs:element name="offerList">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="tns:offer" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="offer" type="tns:OfferType"/>

<xs:complexType name="OfferType">
  <xs:sequence>
    <xs:element name="offerId" type="xs:string" nillable="true"/>
    <xs:element name="productId" type="xs:string" minOccurs="0"/>
    <xs:element name="source" type="tns:SourceType"/>
    <xs:element name="thumbnail" type="tns:PictureType" minOccurs="0"/>
    <xs:element name="price" type="tns:PriceType"/>
    <xs:element name="merchantName" type="xs:string" minOccurs="0"/>
    <xs:element name="summary" type="xs:string"/>
    <xs:element name="offerUrl" type="xs:anyURI"/>
  </xs:sequence>
</xs:complexType>
```

`book-code/chap09/soashopper/soashopper-services-soap/src/main/webapp/WEB-INF /wsdl/retail.xsd`

This schema was used as a guide for writing the `processXML()` function appearing in Example 10–6. In this function, the response XML document from SOAShopper is passed in as the parameter `searchDoc`. As indicated by the schema, each individual offer[7] returned is contained in an `<offer>` element. Hence, the line:

```
var listOffers = searchDoc.getElementsByTagName('offer');
```

7. An "offer" is a product offered for sale on one of eBay, Yahoo! Shopping, or Amazon.

returns an array[8] of `<offer>` elements. The `processXML()` function then proceeds to iterate through that array, using the DOM API to extract the following information:

- `source`—the source of the offer (i.e., eBay, Yahoo!, or Amazon)
- `thumbnailHtml`—a fragment of HTML referencing a thumbnail image of the product offered (e.g., `<img src="http:// ..." width=".." height=".."/>`)
- `priceStr`—the price of the offer (e.g., USD 19.95)
- `summary`—a string containing a description of the offer
- `urlHtml`—a fragment of HTML referencing the page where the offer can be purchased (e.g., `<a href="http:// ....">link</a>`)

Example 10–6 does not show the code used to extract each variable, but it contains enough to give you an idea of how the DOM API is used to process the returned XML.

Example 10–6 The Function `processXML()` Walks the DOM of the XML Returned by the REST Endpoint to Extract the Data That Gets Displayed

```
function processXML(searchDoc) {
try {
var listOffers = searchDoc.getElementsByTagName('offer');
for (var i=0; i<listOffers.length; i++) {
  var item = listOffers.item(i);
  var sourceStr =
    item.getElementsByTagName('source').item(0).firstChild.data;
  var thumbnailElts = item.getElementsByTagName('thumbnail');
  var thumbnailElt;
  var thumbnailUrl = "";
  var thumbnailHtml = "";
  if (thumbnailElts && thumbnailElts.item(0)) {
    thumbnailElt = thumbnailElts.item(0);
    thumbnailUrl =
     thumbnailElt.getElementsByTagName('url').item(0).firstChild.data;
    thumbnailHtml = "<img src='"+thumbnailUrl+"'";
    var h = thumbnailElt.getElementsByTagName('pixelHeight');
    if (h && h.item(0)) {
      thumbnailHtml += " height='"+h.item(0).firstChild.data+"'";
    }
```

8. Technically a `NodeList`.

```
          var w = thumbnailElt.getElementsByTagName('pixelWidth');
          if (w && w.item(0)) {
            thumbnailHtml += " width='"+w.item(0).firstChild.data+"'";
          }
          thumbnailHtml += "/>";
        }
```

book-code/chap09/soashopper/soashopper-ajax/src/main/webapp/search.html

Next, the values that have been extracted need to be put into a format that can be loaded into the Dojo table widget. The widget accepts data in *JSON* format,[9] so the `processXML()` function creates a new variable, `jsonData`, to hold the data in that form. Each offer, in JSON format, is loaded into a global array named `theSOAShopperLiveData` (see Example 10–7).

Example 10–7 Search Results Data Is Converted to JSON Format for Display by the Dojo Table Widget

```
        var jsonData = {
          Id:i,
          source:sourceStr,
          thumbnail:thumbnailHtml,
          price:priceStr,
          summary:summaryStr,
          url:urlHtml
        };
        theSOAShopperLiveData.push(jsonData);
      } // end for
      populateTableFromLiveSOAShopperData();
```

book-code/chap09/soashopper/soashopper-ajax/src/main/webapp/search.html

Finally, when all the offer data has been converted to JSON and loaded into the array, the function `populateTableFromLiveSOAShopperData()` is called to load the Dojo table widget.

9. JSON is a text-based data interchange format used as a serialization alternative to XML. It is commonly used in Ajax programming because it works well with JavaScript. See the Glossary. See also www.json.org.

Example 10–8 shows the code that loads the table.

Example 10–8 The JSON Data Is Loaded into the Dojo Table Widget

```
function populateTableFromLiveSOAShopperData() {
try {
  var w = dojo.widget.byId("fromSOAShopperData");
  if(w.store.get().length > 0){
    alert("you already loaded SOAShopper data :)");
    return;
  }
  w.store.setData(theSOAShopperLiveData);
} catch(e) {
  alert(e);
}
}
```

book-code/chap09/soashopper/soashopper-ajax/src/main/webapp/search.html

Note that the first step is to invoke a Dojo function (`dojo.widget.byId`) to get a reference to the Dojo table widget. The Dojo functions are loaded using script elements such as:

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

in the HTML `<head>` element. Once we have a reference to the table widget, it is loaded with the JSON data by calling the `store.setData()` method. Note that `w.store` is the data store associated with the table widget referenced by `w`.

Example 10–9 The Dojo `FilteringTable` Widget Is Used to Display the Search Results

```
<table dojoType="filteringTable" id="fromSOAShopperData" multiple="true"
  alternateRows="true" cellpadding="0" cellspacing="0" border="0"
  style="margin-bottom:24px;">
  <thead>
    <tr>
      <th field="source" dataType="String">Source</th>
      <th field="thumbnail" dataType="html" align="center">Image</th>
      <th field="price" dataType="String">Price</th>
```

```
        <th field="summary" dataType="String">Summary</th>
        <th field="url" dataType="html">Link</th>
      </tr>
    </thead>
  </table>
```

book-code/chap09/soashopper/soashopper-ajax/src/main/webapp/search.html

Wrapping up this tour of the SOAShopper JavaScript, Example 10–9 shows the HTML for the Dojo table widget. Notice that it contains the attribute `dojoType` that identifies it as a `FilteringTable`. The `<th>` header cells in this table contain field attributes that map each column to the corresponding JSON field name (see Example 10–7).

10.4 Conclusions about Ajax and Java EE

In this chapter, I presented a brief overview of Ajax programming by focusing on how to create a front-end for the SOAShopper application constructed in Chapter 9. I hope you have enjoyed this little detour from Java programming and found it helpful for understanding one type of consumer of Java EE Web services. Some of the more important takeaways from this chapter are as follows:

- Ajax and Java EE support a nice separation of concerns, where server-side Java EE handles the hard-code SOA integration and deployment of Web service endpoints, and Ajax provides an attractive and user-friendly front-end.
- The entire application, Ajax front-end, and Java EE back-end can be bundled as a single EAR for painless deployment to any Java EE application server.
- Creating Ajax applications requires a mastery of JavaScript and HTML DOM that may not be familiar to most server-side Java EE programmers. However, as I illustrated in the SOAShopper search example presented here, it is not too difficult to pick up those skills.
- When creating and deploying Java EE service endpoints, it is probably good practice, at least for the more complex services, to create a simple Ajax front-end to go along with the service. An Ajax front-end makes it easy for the consumers of a service you have written

to visually experience the data your service returns. The ability to "play" with a Web service in such a manner can give a developer a much better intuitive sense for the service interface than a WSDL or XML schema.

In the next and final chapter, I look at an alternative to the Java Web Services framework that is WSDL-centric, rather than Java-centric. This SOA-J framework, first mentioned at the end of Chapter 1, leverages JWS, but provides an alternative paradigm for Web services development and deployment.