

AjaXQuery

Using Ajax with XQuery In Web applications

Skill Level: Intermediate

[Brian M. Carey \(careyb@triangleinformationsolutions.com\)](mailto:careyb@triangleinformationsolutions.com)

Information Systems Consultant
Triangle Information Solutions

07 Jul 2009

Discover how you can get the full benefit of using XQuery technology together with Asynchronous JavaScript and XML (Ajax). Your Web application will have the back-end benefit of sophisticated XML querying as well as the client-side benefit of rich presentation without the distraction of repeated requests.

Section 1. Before you start

Discover what you need to get the most from this tutorial.

About this tutorial

Ajax is an acronym for Asynchronous JavaScript + XML. XQuery is a technology used to query Extensible Markup Language (XML) documents. Together, these technologies provide a powerful means of creating rich Web applications that facilitate client-side dynamic content derived from XML document queries—an excellent solution for applications that require data access against an XML document as opposed to a relational database. And, because XML is quickly becoming the generally accepted means of information interchange (especially when platform independence is required), the necessity of querying against XML to present information in a Web application is growing in popularity. A Web application implementation that uses Ajax together with XQuery can enable complex queries

against data stored in an XML format and cleanly present the information to the user.

This tutorial explains how to create a Web application—called *FishinHole.com*—that uses these two powerful technologies together. The Web application uses Java™ enterprise technology with the popular Spring framework. You also use DataDirect's XQuery application programming interface (API), or XQJ, for XQuery purposes. Finally, you deploy the application to an Apache Tomcat server. You can then access the application with a standard Web browser and see the benefits of using XQuery with Ajax.

Objectives

In this tutorial, you first learn a bit about XQuery—what it is and how it works. Next, you learn about Ajax and its place in Web applications. The requirements for the sample Web application are established, and the tutorial guides you through the process of creating a "basic" Web application using Spring. You also create a basic Web application that does *not* use Ajax and XQuery, which will enable you to see how things work without XQuery and Ajax so that you can understand the advantages of using these technologies together. Finally, you're guided through the process of modifying the Web application to use Ajax and XQuery.

Prerequisites

To use this tutorial, you should have a basic understanding of Web application development and deployment as well as a basic understanding of the Java programming language, XML, HTML, and the JavaScript language. Ideally, you should also understand Tomcat server administration.

System requirements

To run the examples in this tutorial, you need a platform that can support the [Apache Tomcat](#) application server. The tutorial assumes that the server will run on a Microsoft® Windows® platform, but UNIX®-savvy users will certainly be able to make the appropriate adjustments to enable implementation on that platform, as well.

You also need the [Spring framework](#), as that is the tool that you use to implement the Model-View-Controller (MVC) pattern in the Web application. This framework is also used for dependency injection.

Finally, you must have the [XQJ library](#). This is the API that the application uses to actually perform the XQuery processing.

Section 2. The introductions

These brief introductions to XQuery and Ajax fill you in on what you need to know to get started.

A brief introduction to XQuery

It's this simple: XQuery is to an XML document what Structured Query Language (SQL) is to a relational database. XQuery enables you to use expressions to extract data from an XML document. That data can include one simple value or an entire subtree of the document, such as an element and all of its children.

To accomplish this, XQuery uses XPath expressions, which involve the famous "FLWOR" expressions. These expressions provide a powerful means of extracting and returning data from XML documents in almost any fashion imaginable.

The syntax of the language is based on the tree-like nature of the XML document itself. XQuery is aware of processing instructions and attributes as well as elements.

Please note that for XQuery to function properly, the XML document being queried must be well formed, but not necessarily valid. Recall that a well-formed XML document means that it conforms to the World Wide Web Consortium's (W3C's) XML standard. (See [Resources](#) for more information.) An XML document is valid when it conforms to its own Document Type Definition (DTD) or schema.

A brief introduction to Ajax

Ajax is one of the latest bleeding-edge technologies Web developers use to enable rich client presentation. It accomplishes this by invoking a new request without disrupting the current view. An XML document is returned that is then displayed to the user, frequently as a sub-page within the current presentation. In short, Ajax gives you the benefit of server-side dynamic content while still looking like client-side dynamic content.

Ajax generally fulfills its billing through the use of the XMLHttpRequest Document Object Model (DOM) API, which—until the advent of Ajax—Web developers rarely used. The request itself can be either one of the `GET` or `POST` varieties. As with any other request, a response is returned, which can be an error. If the response is not erroneous, the actual text of the response is used to update the current view.

Remembering a famous statement by Voltaire (namely, "The Holy Roman Empire was neither holy, nor Roman, nor an empire"), thorough research into various Ajax implementations leads one to realize that Ajax does not require JavaScript code, does not require XML, and does not need to be asynchronous. After stripping all of that out, the only part left in the acronym is the conjunction *and*. But the acronym sounds cool, so the industry has decided to keep it.

Section 3. Laying the foundation of a Web application

Now it's time to start putting together a Web application. Actually, two Web applications. The first one does not use Ajax with XQuery. The second application does. This way, you'll be able to see the advantages of the latter.

What follows is a brief overview of the technologies used in these Web applications so that you can compile, deploy, and execute them in your local environment.

Java technology

You'll use the latest version of the Java programming language (Version 6) in these applications. This is done for two reasons: First, the Java software development kit (JDK) is available at no charge, and second, if a certain technology is free, why wouldn't you want the latest of it?

XQJ

XQJ is DataDirect's implementation of XQuery. This fabulous library is a must for anyone requiring XQuery functions within a Java development environment. Although this is a commercial product, the company offers a free trial download. As of this writing, the trial period ends 15 days after installation.

Apache Tomcat

Tomcat is a Web application server compliant with Java enterprise standards, brought to the development community by The Apache Software Foundation. The good folks at that foundation have created some of the finest technical products and made them available at the best price possible: US\$0. For purposes of this tutorial, you use Version 6 of the Tomcat server.

The Spring Framework

One of the latest (and probably one of the greatest, in my opinion) Java enterprise frameworks is the Spring Framework. For purposes of this tutorial, you'll use Version 2.5 of this framework for dependency injection (automatic instantiation and insertion of dependent objects into their respective hosts) and implementing the MVC pattern.

Other: please specify

The other technologies used in this tutorial are implicit. For example, JavaScript code is automatically processed when the Web pages are served through the Tomcat server. So no other downloads should be required.

Section 4. A tale of two applications: part 1

Build your first application—an application that doesn't use Ajax with XQuery.

The business requirements

Congratulations! You have just been promoted to chief information officer (CIO) of FishinHole.com, an e-commerce corporation that markets fishing tackle over the Internet. Your marketing department has just informed you that there is a desperate need for a search facility on the Web site. This facility must let users search for lures by usage (casting or trolling) and configuration (minnow shaped or spoon shaped). This is a great chance for you to make a good first impression.

Before you got promoted, your predecessor kept the entire FishinHole.com catalog in XML format. So, you won't need to worry about querying against a relational database to fulfill these requirements. However, you will need to query the XML document as shown in Listing 1.

Listing 1. fishinhole.xml

```
<lures>
  <casting>
    <minnows brand="Yohuri" style="deep" size="3">
      <minnow color="midnight blue">
        <international-prices>
          <price denomination="dollar">3.25</price>
          <price denomination="euro">4.25</price>
        </international-prices>
      </minnow>
    </minnows>
  </casting>
</lures>
```

```

    <price denomination="canadian-dollar">4</price>
    <price denomination="peso">100</price>
  </international-prices>
  <availability>
    <shipping>
      <delay>0</delay>
    </shipping>
    <regions>
      <shipping-region>United States</shipping-region>
      <shipping-region>European Union</shipping-region>
      <shipping-region>Canada</shipping-region>
      <shipping-region>Mexico</shipping-region>
    </regions>
  </availability>
</minnow>
...

```

The existing FishinHole.com Web site is already implemented as a Java enterprise application running with the Spring Framework on a Tomcat server. Those technologies will be left in place.

Coding it out

The first thing you need to do is to put the framework in place, then you can build the actual implementation of this business requirement on top of that. By way of analogy, you first create the skeleton, then you put meat on the bones. So, you configure your web.xml file as shown in Listing 2.

Listing 2. web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">

  <display-name>FishinHole</display-name>

  <servlet>
    <servlet-name>FishinHole</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>FishinHole</servlet-name>
    <url-pattern>*.htm</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

</web-app>

```

Of particular importance in this listing is that the Web application is configured to use the `DispatcherServlet` that the Spring Framework provides. URLs that use the `FishinHole` context and end with the `.htm` extension will be intercepted by that servlet.

You are following conventional patterns for implementing the solution. One pattern is the familiar MVC pattern, which separates information representation (the model) from presentation (the view) from response to user input (the controller). This will be accomplished using the Spring Framework.

Another pattern you will implement is a strong separation of concerns between your controller and your data-access object. The old-schoolers may refer to this as the *Business Delegate* pattern. Sometimes, it is just called a *service* or a *manager*.

The first thing you implement is the data-access object, shown in Listing 3. This is the code that accepts criteria and queries the XML document for elements that match the criteria. You decide on a simple StAX parser for this purpose.

Listing 3. LuresDao

```
public class LuresDao {
    private static final String XML_FILE = "c:/fishinhole/fishinhole.xml";

    public List<Lure> fetchLuresByUsageAndConfiguration(String usage, String configuration) {
        List<Lure> lures = new ArrayList<Lure>();

        try {
            XMLInputFactory inputFactory = XMLInputFactory.newInstance();

            InputStream in = new FileInputStream(XML_FILE);
            XMLStreamReader eventReader = inputFactory.createXMLStreamReader(in);
            boolean usageMatch = false;
            String currentBrand = null;
            Integer currentSize = null;

            while (eventReader.hasNext()) {
                XMLEvent event = eventReader.nextEvent();

                if (event.isStartElement()) {
                    if (event
                        .asStartElement()
                        .getName()
                        .getLocalPart()
                        .equals(usage)) {
                        usageMatch = true;
                    }
                }

                if (event
                    .asStartElement()
                    .getName()
                    .getLocalPart()
                    .equals("minnows")
                    && configuration.equals("minnow") && usageMatch) {
                    QName qName = QName.valueOf("brand");
                    currentBrand = event
                        .asStartElement()
                        .getAttributeByName(qName)
                }
            }
        } catch (Exception e) {
            // ...
        }
    }
}
```

```

                .getValue();

                QName qName = QName.valueOf("size");
                currentSize = new Integer(event

                .asStartElement()
                .getAttributeByName(qName)
                .getValue());
            }

            if (event
                .asStartElement()
                .getName()
                .getLocalPart()
                .equals("spoons")
                && configuration.equals("spoon") && usageMatch) {
                QName qName = QName.valueOf("brand");
                currentBrand = event
                    .asStartElement()
                    .getAttributeByName(qName)
                    .getValue();

                QName qName = QName.valueOf("size");
                currentSize = new Integer(event

                .asStartElement()
                .getAttributeByName(qName)
                .getValue());
            }

            if (event
                .asStartElement()
                .getName()
                .getLocalPart()
                .equals(configuration) && usageMatch) {
                QName qName = QName.valueOf("color");
                String color = event
                    .asStartElement()
                    .getAttributeByName(qName)
                    .getValue();

                Lure lure = new Lure();
                lure.setConfiguration(configuration);
                lure.setColor(color);
                lure.setBrand(currentBrand);
                lure.setUsage(usage);
                lure.setSize(currentSize);
                lures.add(lure);

                continue;
            }
        } else if (event.isEndElement()) {
            if (event.asEndElement().getName().getLocalPart().equals(usage)) {
                usageMatch = false;
            }
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (XMLStreamException e) {
        e.printStackTrace();
    }
}

return lures;
}
}

```

It is beyond the scope of this tutorial to explain the use of StAX. The basic idea

behind this code is that the method `fetchLuresByUsageAndConfiguration` accepts two parameters: `usage` and `configuration`. *Usage* refers to how the lure is actually used (casting or trolling). *Configuration* refers to the physical shape of the lure (spoon or minnow). The method searches the XML document found at `C:\fishinhole\fishinhole.xml` for elements that match that criterion. It then translates those elements into a list of `Lure` objects, simple beans with properties that identify important information about the lures.

The `service` component simply implements one method (also called `fetchLuresByUsageAndConfiguration`) and delegates that method to `LuresDao`.

The `controller` defines what happens when a user accesses the `FishinHole` Web context and the `catalog.htm` page, shown in Listing 4.

Listing 4. CatalogController

```
protected ModelAndView handleRequestInternal(HttpServletRequest request,
    HttpServletResponse response) throws Exception {
    ModelAndView mv = new ModelAndView();
    mv.setViewName("catalog");
    String usage = request.getParameter("usage");
    String configuration = request.getParameter("configuration");

    System.out.println("usage is " + usage);
    System.out.println("configuration is " + configuration);

    if (usage != null
        && !usage.equals("")
        && !usage.equals("0")) {

        if (configuration != null
            && !configuration.equals("")
            && !configuration.equals("0")) {
            List<Lure> lures = luresService
                .fetchLuresByUsageAndConfiguration(usage, configuration);
            mv.addObject("lures", lures);
            System.out.println("size is " + lures.size());
        }
    }

    return mv;
}
```

The idea here is that if `catalog.htm` looks at the URL string for two parameters—`usage` and `configuration`—and both of those parameters exist and have valid values, the controller invokes `fetchLuresByUsageAndConfiguration` on the service. The service, as you will recall, returns a list of `Lure` objects that match those criteria. This list is then added to the model, where the JavaServer Pages (JSP) page can display it.

Deployment

If you haven't done so already, this would be an excellent time to go the [Downloads](#) section and grab the first Web archive (WAR), FishinHole.war. For licensing reasons, the Java archives (JARs) needed to actually run the WAR—shown below—on the Tomcat server were not included:

- ddxq.jar
- ddxqsaxon8.jar
- ddxqsaxon8a.jar
- commons-logging-1.1.1.jar
- jstl.jar
- log4j-1.2.9.jar
- spring-web.jar
- spring-webmvc.jar
- spring.jar
- standard.jar

The good news is that all the JAR files except for the first three can be found within the Spring distribution. The others are part of XQJ. See [Resources](#) for information on downloading these technologies.

When you have obtained the necessary JAR files, simply include them in the WEB-INF/lib directory of the WAR file. This is as simple as treating the WAR file as a standard compressed file (for example, a file in ZIP format) and adding those files to that location.

When the WAR file is complete, you need to put the data and configuration files in place. For purposes of this tutorial, it's assumed that these files will reside in the C:\fishinhole directory. In that directory, you should have the following files:

- fishinhole.xml
- searchResults.xq

The second file will not be used in this part of the tutorial, but go ahead and put it there anyway as a forward-looking measure.

If you need to change the location of the files for any reason, please make sure that you update the appropriate line in [Listing 3](#).

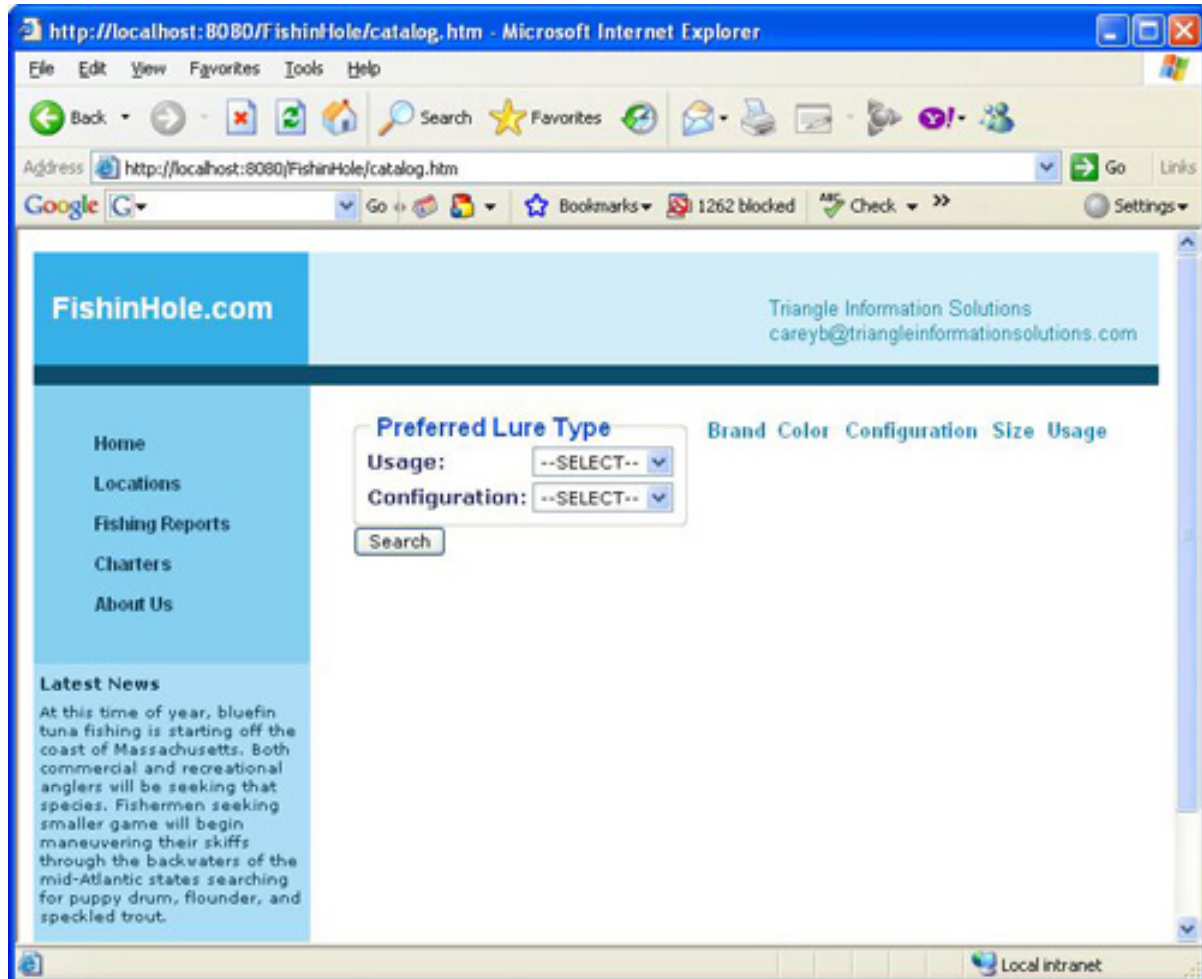
Now, launch Tomcat and deploy the WAR file. Doing so requires a basic understanding of Tomcat server administration, which is beyond the scope of this

tutorial. It is assumed that you are running the server on localhost (127.0.0.1) and using port 8080 to listen for requests. The latter is the Tomcat default.

The fruits of your labor

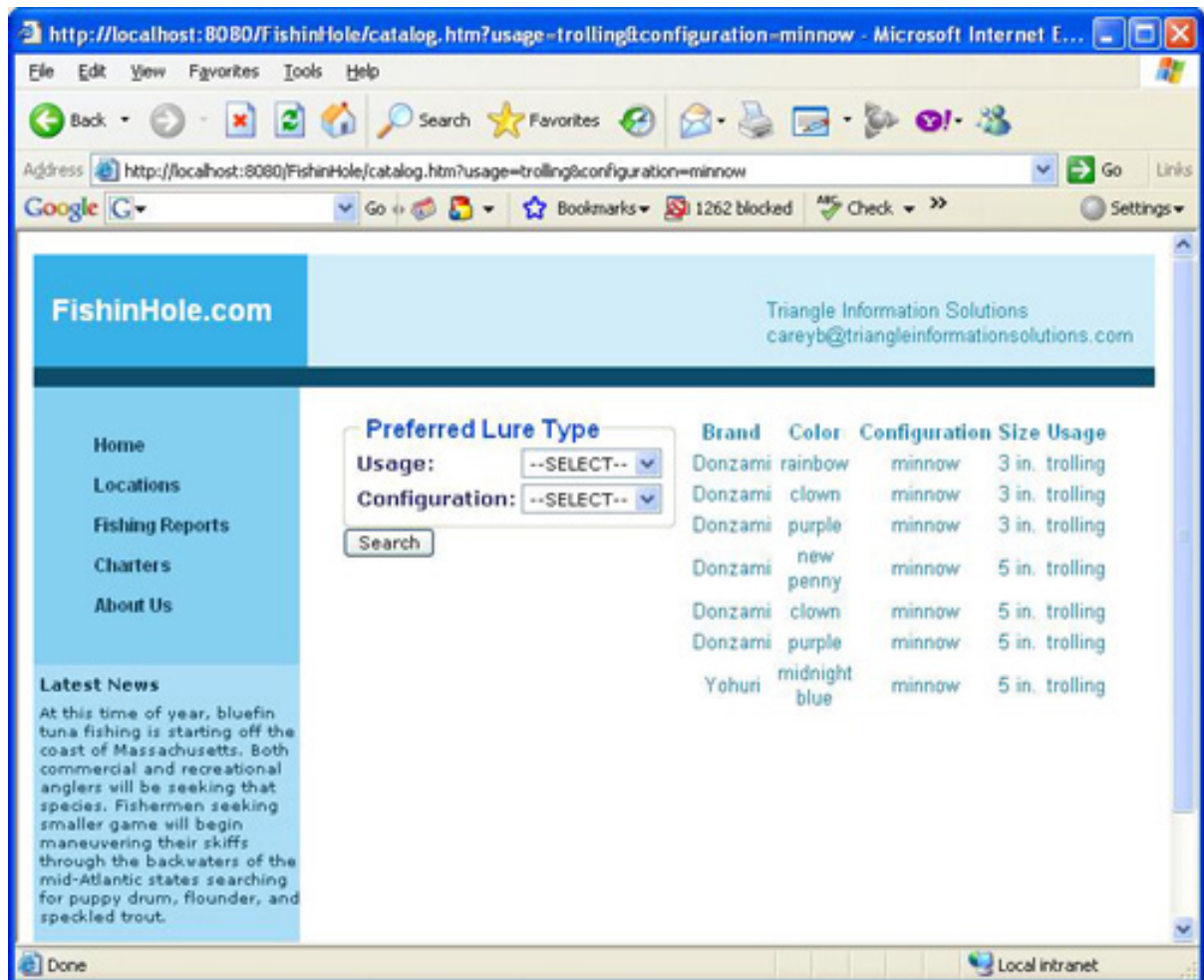
Open a browser, and point your URL to `http://localhost:8080/FishinHole/catalog.htm`. You should see something like Figure 1.

Figure 1. The Catalog page



Now, from the **Usage** drop-down list, select **Trolling**; from the **Configuration** drop-down list, select **Minnow**. Click **Search**. You should now see something like Figure 2.

Figure 2. The search results



A couple of things should be noted at this point. First, did you notice how the entire page reloaded when you clicked Search? It was quick, but it happened. If you didn't notice, feel free to perform another search. Select a different combination, then click **Search** again. The entire page reloads.

Another thing to note is that the default options are selected from the drop-down lists. In this case, the **--SELECT--** options are preselected as opposed to the actual options that you selected. Yes, this is fixable with some JavaServer Pages Standard Tag Library (JSTL) work and server-side settings, but the point is that when the entire page is reloaded, you have to account for setting those values. If, however, only a small subset of the page is reloaded, independent of the drop-down lists, then you don't have to worry about tracking and resetting the appropriate values.

An unexpected response

You show these results to your marketing department personnel and quickly learn that this solution is not acceptable. They inform you that entire page reloads and the

default values being displayed in the drop-down lists after the search results are unacceptable to your customer base. They begin to question your ability as a CIO. You go home depressed.

That night you have a dream about Ajax.

Section 5. A tale of two applications: part 2

Rebuild your application and regain your company's faith in you.

Rethinking things

It's a new day, and last night's dream has turned into an epiphany.

You realize that you can solve both of the problems that the marketing department raised by using Ajax. The idea is that instead of refreshing the entire page, you would only refresh the portion of the page that displays the search results.

But what about the presentation? You could create a separate JSP page that the controller returns and that displays the Lure beans, as mentioned before. In that case, you would still use StAX to parse the XML document, and the current patterns would be left in place. The view that would be returned would be a small JSP page that contained only the search results, not the rest of the page. Then, using Ajax, that view would be displayed in the area of the page used for presenting those results.

However, you recall from reading an article I wrote on developerWorks that XQuery can be used not only to retrieve XML data based on criteria but also to translate the output into HTML. The translated output can then be displayed only in the portion of the page reserved for the search results. The advantage here is that no additional JSP page design is required, and the XQuery implementation could take care of the data-access portion of the application.

Changing the controller layer

So, now you change the code in the controller to get the ball rolling in this direction. Listing 5 shows those changes.

Listing 5. CatalogController revisited



```
protected ModelAndView
handleRequestInternal(HttpServletRequest request,
    HttpServletResponse response) throws Exception
{
    ModelAndView mv = new ModelAndView();
    mv.setViewName("catalog");

    return mv;
}
```

This code is significantly shorter than the same method in [Listing 4](#). This is because this controller is only going to be used to display the initial page, not the search results.

So, what will change the search results? That would be the `UpdateSearchResultsController`. The salient portion of that controller is displayed in [Listing 6](#).

Listing 6. UpdateSearchResultsController

```
protected ModelAndView handleRequestInternal(HttpServletRequest request,
    HttpServletResponse response) throws Exception {

    ModelAndView mv = new ModelAndView();
    mv.setViewName("searchResults");

    String usage = request.getParameter("usage");
    String configuration = request.getParameter("configuration");

    System.out.println("usage is " + usage);
    System.out.println("configuration is " + configuration);

    if (usage != null && !usage.equals("")
        && !usage.equals("0")) {

        if (configuration != null
            && !configuration.equals("")
            && !configuration.equals("0")) {

            String xqFile = "c:/fishinhole/searchResults.xq";
            String lures = luresService
                .fetchLuresByUsageAndConfiguration
                (usage, configuration, xqFile);
            System.out.println(lures);
            mv.addObject("lures", lures);

        }

    }

    return mv;
}
```

There are some important things to note here. First, you're referencing an XQ file. This is the file that actually contains the XQuery code. As stated previously, it is located in the same directory as the `fishinhole.xml` file for purposes of this tutorial. Again, if you need to change that location, make sure you change the appropriate code above and recompile.

The next thing to note is that you're returning a different view. Recall that your epiphany led you to believe that you wouldn't need to create a new view outright, because XQuery can generate custom HTML based on the criteria provided. So, what's all this about a new view, then? Actually, it's not much when you consider the actual JSP page, shown in Listing 7.

Listing 7. searchResults.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<link rel="stylesheet"
      href='<%= request.getContextPath() %>/style/stylesheet.css'
      type="text/css" />
${lures}
```

Not too complicated, huh? This is because the `lures` variable shown above actually contains the HTML elements that XQuery produced in response to the provided criteria!

Revisiting the service layer

Before getting into the details of the service, it's important to revisit a concept that was mentioned briefly earlier: You are no longer coding the data-access layer. Instead, XQJ acts as your data-access layer, which makes perfect sense, because that is the code that is going to actually access the data and return results based on the given criteria. So, the service layer is no longer dependent on a data-access layer that you write, but rather on XQJ.

However, now the service layer is a bit more complicated than it was in the first iteration. Instead of simply delegating a method to a data-access object that you write, more work is involved this time. See Listing 8.

Listing 8. LuresService

```
private XQExpression getGenericExpression(XQConnection conn) throws XQException {
    XQExpression expression = conn.createExpression();

    expression.bindString(new QName("docName"),
        "c:/fishinhole/fishinhole.xml",
        conn.createAtomicType(XQItemType.XQBASETTYPE_STRING));
    return expression;
}

public String fetchLuresByUsageAndConfiguration(String usage,
    String configuration, String xqFile) throws Exception {

    DDXQDataSource dataSource = new DDXQDataSource();
    XQConnection conn = dataSource.getConnection();
    XQExpression expression = getGenericExpression(conn);
```

```
expression.bindString(new QName("configuration"), configuration,
    conn.createAtomicType(XQItemType.XQBASETYPE_STRING));

expression.bindString(new QName("usage"), usage,
    conn.createAtomicType(XQItemType.XQBASETYPE_STRING));

FileReader fileReader = new FileReader(xqFile);
XQSequence results = expression.executeQuery(fileReader);

return results.getSequenceAsString(new Properties());
}
```

The first method (`getGenericExpression`) is private to the service and is used to basically tell XQuery which XML document you're going to be querying against. Once again, this is the code you change if you are not using the default location that this tutorial recommends.

The next method is the public method invoked by the controller layer. Note that you are now using three parameters instead of two. The first two parameters are the same as in the first iteration, namely, `usage` and `configuration`. The third parameter is the location of the XQ file. Again, this is the file that actually contains the rules for producing HTML based on the criteria. The criteria, of course, are the first two parameters.

It's important to discuss the philosophy behind this design. Because the controller layer is responsible for returning a particular view in response to user input, it's considered a best practice to let the controller decide which XQ file to use and pass that on to the service. This gives you a clean separation of concerns. In this case, the service doesn't concern itself with the view being returned: It just accepts the XQ file from the controller (which does concern itself with the view being returned) and returns the appropriate results.

Note also how the public method relies on the private method described earlier. The private method creates a generic expression that will become more specific based on the selections the user made.

Then, you use XQJ's API to actually bind the values (`usage` and `configuration`) that the user selected to the variables that will be used in the XQ file. This is how XQuery actually selects the elements that match user inputs.

A `FileReader` object is constructed to read the XQ file. And once again, the XQJ API is invoked using that `FileReader` object as the XQuery expression. Finally, the results are returned as an `XQSequence` object.

Because you don't want to return the results as an `XQSequence` object, you convert the object to a `String` object, which is nothing more than HTML. Don't worry about the `Properties` object at this time. The XQJ API requires it, and it is not used here for any reason other than that.

The XQ file

Much has been discussed about the XQ file thus far. But what does it actually look like? Take a look at Listing 9.

Listing 9. searchResults.xq (abbreviated)

```

declare variable $docName as xs:string external;
declare variable $configuration as xs:string external;
declare variable $usage as xs:string external;

<table width="100%" class="searchResultsTable">
  <tr>
    <th>Brand</th>
    <th>Color</th>
    <th>Configuration</th>
    <th>Size</th>
    <th>Usage</th>
  </tr>

  {
  if ($configuration = 'minnow' and $usage = 'casting') then
  for $minnows in doc($docName)//casting/minnows
  return
  <div>
    {
    for $minnow in $minnows/minnow
    return
    <tr>
      <td>{data($minnows/@brand)}</td>
      <td>{data($minnow/@color)}</td>
      <td>{$configuration}</td>
      <td>{data($minnows/@size)}</td>
      <td>{$usage}</td>
    </tr>
    }
  </div>
  ...

```

Take note of the first three lines (the `declare` statements). You'll probably recall that all of those variables were bound to values back in the service layer. Look at [Listing 8](#) for verification.

Next in the XQ files comes some fairly rudimentary HTML. Again, recall that XQuery not only performs queries, but can return those queries in a format that you determine. In this case, that format is HTML and will be used to replace the existing HTML of the view that contains the search results.

Next comes a simple `if . . . then` statement using XQuery syntax. It basically states that if the configuration is a minnow lure and the usage is casting, then it needs to look specifically for `minnows` elements underneath the `casting` element. When it finds that element, it returns an HTML row in a table based on the values found in each `minnow` child element.

Note that the cells in each row are populated from either the input data or an attribute from the `minnow` element. That is the significance of the at symbol (@) in front of words like *brand* or *color*. The query will retrieve those attributes from that element. The `data` function forces XQuery to simply retrieve the value of the attribute instead of the actual attribute itself.

The Ajax implementation

Recall that the *j* in *Ajax* refers to JavaScript. So, that is how you'll be coding your Ajax implementation. You'll create a JavaScript file that is referenced by your various JSP pages that require the use of Ajax. In that JavaScript file, you will write code as it appears in Listing 10.

Listing 10. `searchResults.xq` (abbreviated)

```
function ajax(url, parameters, callback, isGet) {
    var request;

    var parameterString = "";
    for (var param in parameters) {
        if (parameterString) parameterString += "&";
        parameterString += param + "=" + escape(parameters[param]);
    }

    if (isGet) url += "?" + parameterString;

    if (window.XMLHttpRequest) {
        request = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        request = new ActiveXObject("Microsoft.XMLHTTP");
    }

    if (!request) {
        return;
    }

    if (callback) {
        request.onreadystatechange = function() {
            if (request.readyState == 4) {
                callback(request.status == 200, request.responseText);
            }
        }
    }

    request.open(isGet ? "GET" : "POST", url, true);
    if (!isGet) request.setRequestHeader("Content-Type",
        "application/x-www-form-urlencoded");
    request.setRequestHeader("X-Requested-With", "XMLHttpRequest");
    request.send(isGet ? null : parameterString);
}
```

This Ajax code is similar to the many flavors of Ajax code that you can download from innumerable sources all across the vast expanse of the Internet. It's cross-browser compliant and supports both GET and POST requests.

The first parameter is the actual URL. This is the URL that will be invoked under the covers. The next parameter is called `parameters` and is an array similar to a `HashMap` in the Java programming language. In this case, the key is the name of the parameter, and the value is the actual value of the parameter. In the case of this tutorial, a valid parameter would be `usage`, and a valid value for that parameter would be `casting`.

The next parameter is called `callback` and simply defines the JavaScript function that should be invoked when the request is completed. For example, if you wanted to invoke a function named `searchCompleted` after the request was completed, you would specify that here.

The final parameter is a simple Boolean called `isGet`. If that Boolean is `True`, then the request is a `GET`. If not, then it is assumed that the request is a `POST`.

The first part of the body of the code constructs a request string in response to the name-value pairs that have been provided in the `parameters` parameter. The following line of code appends the name-value pairs properly to the URL if the request is a `GET`.

The next segment of the code ensures cross-browser compliance. The actual type of request is here instantiated based on the browser that the user is currently using.

Next, a test is performed to determine whether there is a callback. If so, then the `callback` function is invoked after the request is complete.

The last four lines of code actually initiate the request. The request is executed as either a `GET` or a `POST`, with the appropriate header set based on the result of the `isGet` parameter.

The JSP page

The JSP page (`catalog.jsp`) is similar to the same page in the first iteration. The outstanding difference is that when the user clicks **Search**, instead of a simple form submission, the script shown in Listing 11 is executed.

Listing 11. `searchResults.xq` (abbreviated)

```
<script type="text/javascript">
function search() {
    var usageElement = document.getElementById("usageSelect");
    var configurationElement = document.getElementById("configurationSelect");

    var usage = usageElement.options[usageElement.selectedIndex].value;
    var configuration
        = configurationElement.options[configurationElement.selectedIndex].value;

    if (usage == "0") {
```

```
        alert("Please select a usage!");
        return;
    }

    if (configuration == "0") {
        alert("Please select a configuration!");
        return;
    }

    var parameters = {};
    parameters["usage"] = usage;
    parameters["configuration"] = configuration;

    var resultsDiv = document.getElementById("searchResults");

    ajax("./updateSearchResults.htm",parameters,function(success,response) {
        resultsDiv.innerHTML = response;
    });
}
</script>
```

You'll note that this script starts off by doing nothing more than grabbing the values that the user selected in the drop-down list, validating them, then adding them to the aforementioned `parameters` array.

Then, the fun begins. First, the script obtains a reference to the `<div>` element with the ID of `searchResults`. This starts off as an empty element and is populated with the search results based on the user's selections. This reference is named `resultsDiv` in the script.

Next, the script invokes the `ajax` function, which was described in [Listing 10](#). Instead of referring to a separate callback function, the script just builds one on the fly. In this case, the `innerHTML` property of `resultsDiv` is changed to the value of `response`. The `response` value is the actual HTML that you receive from the invocation of the `ajax` function. This HTML then replaces the existing HTML code (if any—and there won't be if this is the first search since the page was loaded) with the new results.

Also note that the `ajax` function invokes a local URL—in this case, `./updateSearchResults.htm`. This URL will be mapped in your Spring configuration file to the controller that was described in [Listing 6](#), `UpdateSearchResultsController`.

Blast off!

You now have a complete application! It's time to deploy it and test it as was done previously.

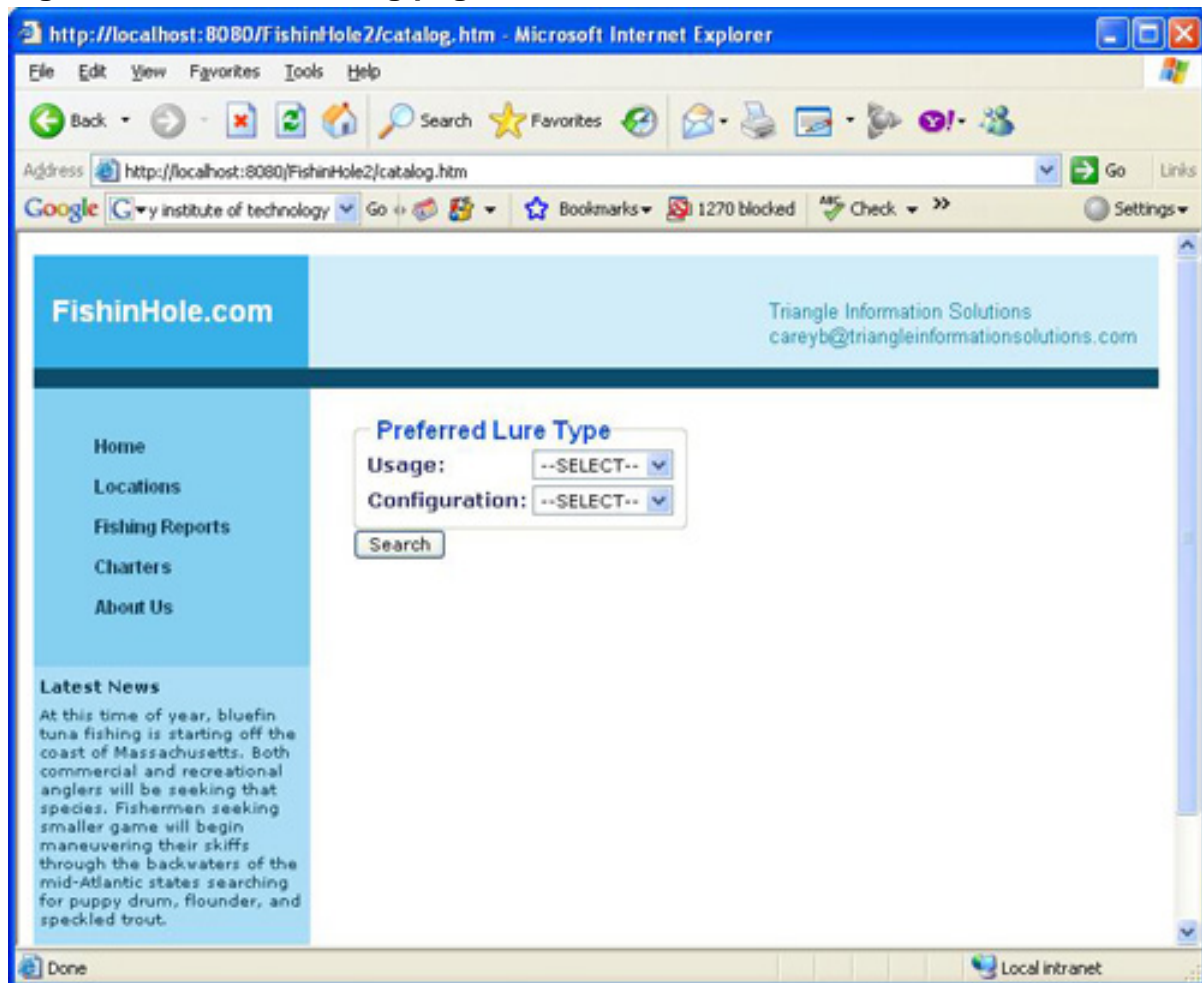
The deployment procedure is identical to the procedure described in the first iteration except that in this case, you will be deploying `FishinHole2.war`. As before,

the standard rules and procedures for deploying an application to a Tomcat application server apply. You will not need to worry about "un-deploying" the WAR file from the first iteration.

The main difference in testing this iteration versus the previous iteration is that the context is different. This time, instead of the context being `FishinHole`, it is now `FishinHole2`. This name change distinguishes it from the previous version.

Now, open a browser and point your URL to `http://localhost:8080/FishinHole2/catalog.htm`. You should see *almost* exactly what you saw when you did this during the first iteration (see Figure 3).

Figure 3. The new Catalog page

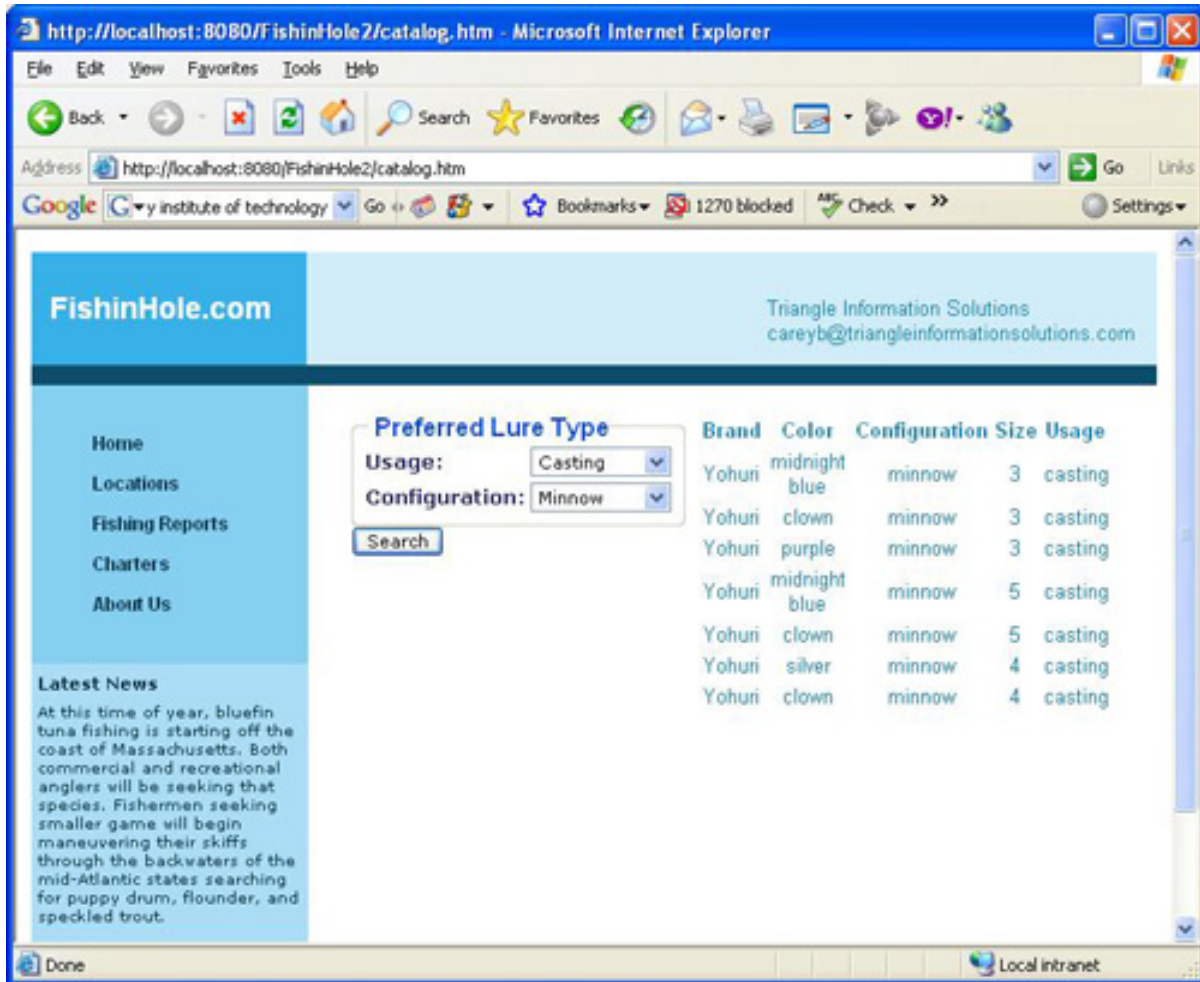


The subtle difference is that there are no headers on the initial page. That's just a preference here.

Now, from the **Usage** drop-down list, select **Casting**, and from the **Configuration** list, select **Minnow**. Click **Search** and see what happens.

For starters, you may notice that it takes a few seconds for the results to be displayed. This is because, as promised, the `ajax` function is actually invoking a request: It's just doing it behind the scenes. Requests take a few seconds because they are network intensive and the JSP page needs to be compiled during the first access (see Figure 4).

Figure 4. The new Results page



Your page should look similar to what is shown here. The results should actually be identical, because it is assumed that you are working with the XML file provided with this tutorial.

You should notice a couple of things immediately. First, the application didn't refresh the entire page. Recall that the marketing department specifically asked that you provide an application that does not do this. So, this is good!

Next, note the values in the drop-down lists. They haven't changed! This is a side effect of not refreshing the page with an entirely new request. This is also good.

Notice one more thing. If you select another usage type, like **Trolling**, and click

Search, you will see the results change right before your eyes almost instantaneously. The JSP page has already been compiled, so the results are returned a lot faster.

Before the jury

The good folks in marketing take a look at your second iteration. Do the results display without a page reload? Check. Do the user's previous drop-down selections stay in place? Check. They shake your hand and congratulate you on a job well done.

As you go back to your desk, you overhear people saying very nice things about you to their co-workers. You're off to a great start.

Section 6. Summary

Now, it's time for you to do what developers do best when they want to learn: tinker. Play around with the code that has been provided in this tutorial. Add some drop-down options. Make changes to the XML. Play with the XQuery code. Recompile and re-deploy.

Using Ajax with XQuery can be a powerful means of providing dynamic server-side content that appears to the casual user to be nothing more than really slick dynamic client-side content. Using the two together, you can still follow best practices and conventional design patterns. You can implement solutions that contain robust queries against XML documents. You can format those queries into attractive HTML.

Downloads

Description	Name	Size	Download method
XML and XQuery files	FishinHole.zip	28KB	HTTP

[Information about download methods](#)

Resources

Learn

- "[An introduction to XQuery](#)" (developerWorks, June 2001) takes a look at the W3C's proposed standard for an XML query language.
- "[Use XQuery from a Java environment](#)" (developerWorks, April 2008) explains how to search documents with XQuery from Java applications.
- I find Wikipedia a good place to start learning about almost any subject. [Ajax](#) is no exception.
- "[Build apps using asynchronous JavaScript with XML \(AJAX\)](#)" (developerWorks, November 2005) is a great starting point for Ajax development.
- Read the [W3C's XML standard](#).
- The [developerWorks Web development zone](#) is packed with tools and information for Web 2.0 development.
- [developerWorks technical events and Webcasts](#): Stay current with the latest technology.
- [Technology bookstore](#): Browse for books on these and other technical topics.

Get products and technologies

- Get [Spring downloads](#) necessary to compile and run the code presented in this tutorial.
- [Apache Tomcat](#) is the application server on which these code samples were tested.
- Obtain Sun's standard edition of the [Java programming language](#).
- "[An XQJ Tutorial: Introduction to the XQuery API for Java](#)" provides instructions for downloading and using XQJ.

Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the author

Brian M. Carey

Brian Carey is an information systems consultant who specializes in the architecture,

design, and implementation of Java enterprise applications. You can follow Brian on Twitter at <http://twitter.com/brianmcarey>, and his tweets are publicly available.

Trademarks

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

IBM and developerWorks are registered trademarks of International Business Machines Corp in the United States, other countries, or both.