

5

User interface development

This chapter

- Demonstrates use of XSLT with Java servlets
- Compares pure J2EE user interface development with an XML approach
- Shows how to build multidevice and multilocale user interfaces
- Covers XML web publishing frameworks

Creating a robust presentation layer for your J2EE application is a challenging endeavor. This is so because the vast majority of J2EE applications are web-based, also known as “thin-client”, applications. In this chapter, we examine some emerging challenges in J2EE user interface design and discuss ways you might use XML technology to overcome them.

We begin by exploring the nature of thin-client user interface development and the challenges you are likely to face when building and maintaining an advanced presentation layer. We then examine the “pure” J2EE approach to building such a layer to see where the current J2EE architecture is lacking.

The remainder of the chapter focuses on overcoming the limitations of the pure J2EE approach using XSLT technology. First we develop an XSLT-based presentation layer from scratch. Then we use XSLT from within a web publishing framework to discover the benefits and drawbacks of using a third party API.

The goal of this chapter is not to convince you that one architecture or product is superior to another. We wish only to make you aware of the available options, see them in action, and understand the positive and negative aspects of taking each approach.

5.1 *Creating a thin-client user interface*

In this chapter, we focus almost exclusively on web-based, thin-client distributed applications. Before diving into the details of overcoming challenges associated with these types of applications, we should take a moment to discuss what they are and why building interfaces for them is so difficult.

DEFINITION A *thin-client application* is one in which the server side is responsible for generating the user interface views into the application. The client side usually consists only of generic rendering software, e.g., a web browser.

If you have built web-based applications before, you are no doubt familiar with the thin-client architecture. Until recently, the burden of generating the user interface for your application on the server side was not too difficult. Two relatively recent developments, however, are now making the development and maintenance of your presentation layer components more challenging.

5.1.1 *Serving different types of devices*

The first new challenge relates to the ongoing proliferation of web-enabled devices. It seems as though anything that runs on electricity these days now has a web browser on it. These “smart” devices include cell phones, PDAs, and refrigerators. (That only sounds like a joke.) The trouble with these smart devices is that they are usually quite dumb, virtually light years behind the current version of your favorite computer-based Internet browser. Some understand a subset of the HTML specification, and others require an entirely separate markup language. An example of the latter is the WAP-enabled cell phone, which requires web pages to be rendered using the Wireless Markup Language (WML).

DEFINITION WML is an XML-based presentation markup language specifically designed for web browsing on cellular telephones. These phones typically feature small display areas and very limited rendering capabilities. WML arranges content into a *deck* of *cards* (pages), each of which should contain very little data and uncomplicated navigation.

Creating a single, integrated user interface that can render content for various types of devices is difficult to do using the traditional J2EE (or ASP, or Cold Fusion, etc.) approach. User interface components in these technologies were not designed to change the entire structure of the generated content based on the type client requesting it. This means that, in order to serve web browser *and* cell phone client types from your J2EE web application, you must develop and maintain two separate presentation layers. One layer would generate an HTML interface, and the other a WML interface. Some updates to the application would require updates to *both* user interfaces. If you think managing two interfaces is scary, consider supporting three more types of client devices in the same application.

5.1.2 *Serving multiple locales*

The other unpleasant possibility concerns serving content to users in different locales. The obvious challenge here is to determine the end user’s preferred language and render the presentation in that language. This means having some mechanism to translate your application’s user interface between natural languages, either on the fly or offline. Offline translation can be expensive and time consuming, but its result is far less likely to be unintelligible. Real-time translation has been done poorly many times, but never done well as far as we know. Also consider that everything might need to be translated, including

text embedded in images and static text that might otherwise be embedded directly in your JSP.

The second, perhaps less obvious, dimension to the multiple locale problem has to do with cultural differences. Beyond the textual content of the page, it may be appropriate and advantageous to change the entire structure of your user interface for a new locale. You might want to change color schemes, page layouts, or even the entire navigational flow of the interface. The reasons for these types of changes have to do with psychology and marketing and other ungeeklike topics that we need not address here. In this chapter, we concentrate on how to satisfy these types of requirements, assuming away the reasons.

If you have tried to serve multiple languages and/or locales effectively using a pure J2EE presentation layer, you already appreciate the magnitude of the challenge. If not, we highlight its scariest features in the remainder of this chapter. What inevitably results from attempting to meet this requirement in the traditional approach is a large collection of mostly redundant, often misbehaved JSPs and servlets that are a nightmare to maintain and extend.

5.1.3 *An example to work through*

To really get at the heart of the matter and see how XML can help overcome the thin-client blues, let us look at some example requirements. For the sake of ubiquity and clarity, we examine a relatively simple user interface and a single point of functionality in a fictitious application. The application is a stock-trading web site, and the functional point to be explored is the rendering of a specific user's "watch list" of stock prices. Constraining our example to one small functional point allows us to concentrate on dynamically changing the user interface without bogging down in other details.

Here are the requirements that concern us in this chapter:

- The watch list page must be viewable via PC-based web browsers and WAP-enabled cell phones.
- The watch list page is rendered for two locales, the United States and Great Britain. Each locale requires its own specific interface with appropriate textual messages.
- The price for each stock listed on the watch list should be expressed in the user's local currency, USD (United States dollars) for the U.S. and GBP (British pounds) for the U.K.

You can see from the requirements above that our watch list user interface must consist of four pages. Figure 5.1 is an HTML page rendered for users in the United States.



Figure 5.1
Stock watch list
HTML page—
United States

Figure 5.2 is an HTML page rendered for users in Great Britain.



Figure 5.2 Stock watch list HTML page—Great Britain

Figure 5.3 is a WML page rendered for users in the United States.



Figure 5.3
Stock watch list WML page—United States

Figure 5.4 is a WML page rendered for users in Great Britain.



Figure 5.4
Stock watch list WML page—Great Britain

We chose the United States and Great Britain as the locales for this example to avoid the requirement of a second language on your part. In the remainder of this chapter, we create and recreate these four pages using a variety of techniques. In section 5.2, we use only the capabilities of J2EE and see where they fail us. In sections 5.3 and 5.4, we bring XML technology to the rescue and create a unified interface for this page that supports multiple device types and locales.

5.2 *The pure J2EE approach*

Before we discuss any new, XML-based architectural models for user interface creation, it is important to understand why we might need them. After all, we

do not want to fix something that is not broken nor overcomplicate an application unnecessarily. In this section we first explore what the pure J2EE approach has to offer and see some of its limitations when using it to develop our example pages.

5.2.1 The J2EE presentation tool kit

The various J2EE presentation layer components are summarized in table 5.1. All of these components, with the exception of applets, execute in the J2EE web container on the server side of a client/server application. These applications are almost always thin-client, web-based ones. In such an architecture the J2EE server-side components are designed to collaborate with one another in an adaptation of the Model-View-Controller (MVC) software design pattern.

Table 5.1 J2EE presentation framework components

J2EE component	Purpose
Applets	Java components downloaded and run in client browsers.
Servlets	Java components invoked through a client request to perform server-side processing and generate a response.
Java Server Pages (JSPs)	Response page templates that allow in-line inclusion of Java code with static content.
JavaBeans	Component framework for creating reuse in your applications.
JSP custom tags	Classes that enable complex logic to be referenced by a simple tag within a JSP. Often used to iterate over processing result sets and dynamically generate UI structures like HTML tables.
Filters	Server side components that intercept requests for pre and post processing.

The MVC presentation architecture is intended to decouple application data (*model*) from its rendering (*view*) to produce a more modular system. The *controller* is a traffic cop of sorts, managing the relationship between the views and the model. The two main goals of the MVC are:

- To enable multiple views of the same information
- To reuse the components used to generate the interface

When MVC is implemented correctly, individual components are independent of one another. A clock is a perfect example of this. The model, or mechanism for keeping time, functions the same regardless of the use of a digital or analog view.

In a J2EE presentation layer, one or more servlets usually act as the controller. They accept requests from clients, perform some processing on the model, and return an appropriate view. In some simple and other poorly designed systems, the servlet itself may render the view. This involves embedding static content like HTML tags directly in the servlet code, and is frowned upon by all of us for its shortsightedness.

A far superior approach is to have your controller servlet select an appropriate JSP as the view and forward processing to it. JSPs are far more maintainable and can make use of custom tags and JavaBeans to decouple much of the logic from the static content in your pages. This is the preferred presentation layer architecture referenced in the JSP specification (where it is called Model 2). Figure 5.5 illustrates the overall concept.

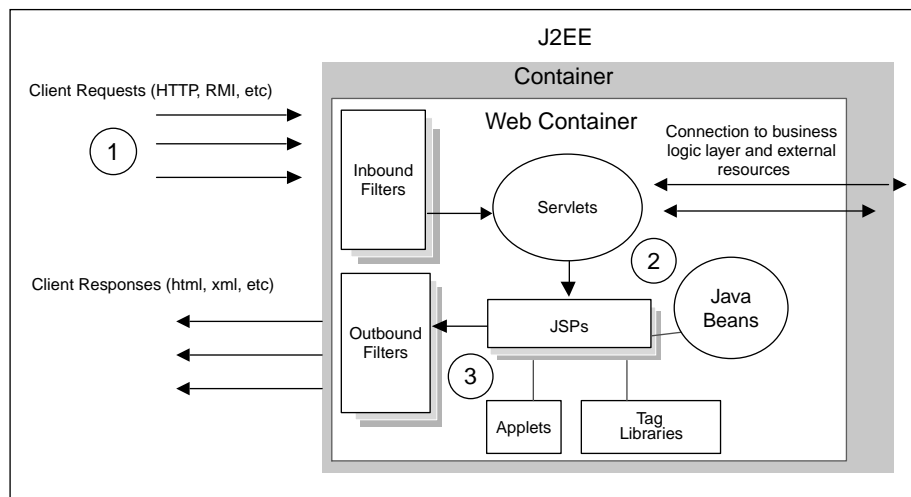


Figure 5.5 The J2EE MVC architecture

5.2.2 *Issues in J2EE MVC architecture*

The J2EE presentation framework is an adaptation of MVC for thin-client applications. In most cases, however, it does not succeed in fully decoupling presentation logic from data in your user interface.

DEFINITION *Presentation logic* refers to the programming elements that control the dynamic generation of the user interface. This term includes the programming variables, statements, and control structures used in your JSP pages.

In this section, we explore some of the goals and challenges common to virtually all pure J2EE presentation layer implementations, one by one.

Enforcing the separation of logic and content

Servlets and JSPs can be implemented in many different ways. In figure 5.5, we saw the Model 2 architecture in which JavaBeans are used as the model, servlets as the controller, and JSPs as the view.

Consider the following request-response flow:

- 1 A web request is intercepted by an inbound filter for preprocessing, authentication, or logging and then redirected to a servlet.
- 2 The servlet performs lightweight processing and interacts with your application's business logic layer, finally forwarding execution to a JSP for rendering.
- 3 The JSP uses a combination of JavaBeans and custom tags to render the appropriate presentation and then forwards the response to an output filter for any postprocessing before the response is returned to the user.

The success of any MVC implementation relies on the clear separation of roles (model, view, and controller) among components. When strictly enforced, the reusability and maintainability of each individual component is greatly enhanced. When not enforced, the attempted MVC architecture makes an already complicated system even harder to maintain and extend.

Limitations of template languages

JSPs are dynamic page templates that contain a combination of logic and data. They were modeled in a similar fashion to the other dynamic template languages, such as Active Server Pages and Cold Fusion, as an enabler to develop dynamic HTML pages. Completely separating code and data in such template languages is difficult, and at times impossible. For example, even the following three-line JSP contains page structure, static content, and code all mixed together.

```
<html>
<h1>Hello <%=request.getParameter(uName)%> </h1>
</html>
```

When JSP was originally developed, it was heavily criticized for embedding Java code directly into HTML pages (as done above). When data and code are collocated, there is inevitably contention between the HTML author and the JSP developer trying to work on the same source file. For enterprise

applications, there is usually an obvious need to separate the roles of UI designer and code hound. To solve this problem, the JSP specification evolved to include the use of custom tag libraries and JavaBeans. These additions freed JSPs of much Java code by encapsulating complex presentation logic within tags and data structures within JavaBeans. XML tags referring to these other components now replace much of the old JSP code.

While the amount of code present in JSPs decreased, the complexity of the overall architecture increased. In particular, tags often contain a combination of presentation markup tags and Java code, making some pages more difficult to maintain. The page designer cannot be expected to modify and recompile tags to accomplish a simple HTML change; and finding the code that needs to change can be tedious. A secondary issue with tags and JavaBeans in JSP is one of enforcement. Developers often opt for the path of least resistance, continuing to embed Java code directly in JSPs.

Code redundancy

As we discussed in section 5.1, the advent of multidevice and internationalization requirements for J2EE applications puts much more strain on the pure J2EE architecture. As we demonstrate in the next section, building our stock quote example pages requires between two and four individual JSP pages. As we increase either the number of locales or the number of client device types being serviced, the number of JSP pages proliferates. Consider the burden of developing and maintaining a set of 100 JSP pages to satisfy five functional requirements. Think about making the same change in 20 different JSP pages, testing them individually, and then regression testing the entire application. If the change is to logic, you hopefully have encapsulated it in a tag or a JavaBean. If the change is to the presentation layout or flow, you are in a world of hurt.

5.2.3 Building our example in J2EE

Enough discussion. It is time to see the issues in action. To service requests for our example watch list page using only J2EE, we require the following components:

- A servlet to accept the client request and interact with our application logic layer to obtain a list of stock quotes.
- A JSP for each device type for which we need to render the list.

Handling the request

We require a servlet to accept watch list user requests, obtain the watch list, and select a view with which to render the list. In this J2EE-only scenario, our servlet will forward the watch list to one of four JSPs, depending on the client device type the user is connecting from and the user's locale.

The first step our servlet will take is to obtain the user's stock quote list. We assume that our application has stored the user's unique identifier as a `String` in the `HttpSession` object. After retrieving that identifier, we make a call to the application logic layer to obtain the stock quote list as a `JDOM` value object.

```
HttpSession session = request.getSession(false);
ListBuilder builderInterface = ...
// ... validate session object exists

String userId = (String) session.getAttribute("userId");
org.jdom.Document quoteList = builderInterface.getWatchList(userId);
```

We then wrap the `quoteList` in a `JavaBean` object and store it in the request, for later retrieval by the JSP component. We go over the bean code later in this section.

```
XMLHelperBean helper = new XMLHelperBean(quoteList);
session.setAttribute(helper, helper);
```

The final step is the most involved. We must determine which of our four JSPs will render the response for the user. To do this, we must determine the user's device type and locale preference. First we develop a method to determine the device type, using the `User-Agent` HTTP header.

```
private String getOutputType(HttpServletRequest request) {
    String userAgent = request.getHeader("User-Agent");
    // compare to list of WAP User-Agents
    // for simplicity, we'll only try one here
    if (userAgent.indexOf("UP.Browser") >= 0)
        return "wml";
    return "html";
}
```

In a real-world scenario, this servlet would maintain a dictionary of known user-agents and their associated MIME content types. For the example, we only output WML if someone is using the `UP.Browser` phone browser. Calling the above method will set the output format. Now we require a method to choose a JSP based on the output format and the user's locale. For that, we use the `Accept-Language` HTTP header(s), which contains an ordered list of preferred locales from the user's browser settings.

```

private String getForwardURL(
    HttpServletRequest request, String outputType) {
    String result = null;
    if (outputType.equals("html"))
        result = "/watchlist/watchlist.html.en_US.jsp";
    else
        result = "/watchlist/watchlist.wml.en_US.jsp";
    Enumeration locales = request.getHeaders("Accept-Language");
    while (locales.hasMoreElements()) {
        String locale = (String) locales.nextElement();
        if (locale.equalsIgnoreCase("en_GB"))
            if (outputType.equals("html"))
                return "/watchlist/watchlist.html.en_GB.jsp";
            else
                return "/watchlist/watchlist.wml.en_GB.jsp";
    }
    return result;
}

```

The foregoing method will choose the appropriate JSP from the four we will develop shortly. Now all that is left to do is make use of these methods in our servlet's request handling method.

```

String outputType = getOutputType(request);
String forwardURL = getForwardURL(request, outputType);
context.getRequestDispatcher(forwardURL).forward(request, response);

```

The complete code for our new servlet is shown in listing 5.1.

Listing 5.1 The watch list JSP servlet

```

import org.jdom.*;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * The stock watchlist servlet with JSP
 */
public class WatchListJSPServlet extends HttpServlet {

    private ListBuilder builderInterface = new ListBuilder();
    private ServletConfig config;
    private ServletContext context;

    public WatchListJSPServlet() { super(); }

    public void init(ServletConfig config)
        throws ServletException {
        this.config = config;
        this.context = config.getServletContext();
    }

```

```

}

public void doGet(HttpServletRequest request,
                 HttpServletResponse response)
    throws ServletException, IOException {
    // get userid from HttpSession
    HttpSession session = request.getSession(false);
    if (session == null) {
        context.getRequestDispatcher("/login.jsp")
            .forward(request, response);
        return;
    }

    String userId = (String) session.getAttribute("userId");
    Document quoteList = builderInterface.getWatchList(userId);

    XMLHelperBean helper =
        new XMLHelperBean(quoteList);
    request.setAttribute("helper", helper);

    String outputType = getOutputType(request);
    String forwardURL =
        getForwardURL(request, outputType);

    context.getRequestDispatcher(forwardURL)
        .forward(request, response);
}

private String getOutputType(HttpServletRequest request) {
    String userAgent = request.getHeader("User-Agent");
    // compare to list of WAP User-Agents
    // for simplicity, we'll only compare one here
    if (userAgent.indexOf("UP.Browser") >= 0)
        return "wml";
    return "html";
}

private String getForwardURL(HttpServletRequest request,
                             String outputType) {
    String result = null;
    if (outputType.equals("html"))
        result = "/watchlist/watchlist.html.en_US.jsp";
    else
        result = "/watchlist/watchlist.wml.en_US.jsp";
    Enumeration locales = request.getHeaders("Accept-Language");
    while (locales.hasMoreElements()) {
        String locale = (String) locales.nextElement();
        if (locale.equalsIgnoreCase("en_GB"))
            if (outputType.equals("html"))
                return "/watchlist/watchlist.html.en_GB.jsp";
            else
                return "/watchlist/watchlist.wml.en_GB.jsp";
    }
}

```

User must be logged in and have an HttpSession

Store the document in a JavaBean for later retrieval

Find the right JSP to render the view

Forward the request and response for rendering

```
        return result;
    }

    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Obtaining and using XML data

Our watch list servlet makes use of a `ListBuilder` component, the code for which you can find on the web site for this book (<http://www.manning.com/Gabrick>). The `ListBuilder` returns a JDOM document containing a list of stock quotes in XML format. Listing 5.2 shows the XML data set we are using in the example. It contains three price quotes for stocks on my watch list. Note that multiple `price` nodes are contained within each `quote` element, each for a different currency. This will enable us to display prices in the user's native currency when the page is generated.

Listing 5.2 Stock quote XML data set

```
<?xml version="1.0"?>
<quote-list date="Nov. 4, 2001" time="9:32 AM EST">
  <customer first-name="Kurt" last-name="Gabrick" id="9999"/>
  <quote symbol="SRMC" name="Sierra Monitor Corporation">
    <price amount="2.00" currency="USD"/>
    <price amount="1.05" currency="GBP"/>
    <price amount="4000.00" currency="MXP"/>
  </quote>
  <quote symbol="IBM" name="International Business Machines">
    <price amount="135.00" currency="USD"/>
    <price amount="67.75" currency="GBP"/>
    <price amount="230000.00" currency="MXP"/>
  </quote>
  <quote symbol="ORCL" name="Oracle Corporation">
    <price amount="15.00" currency="USD"/>
    <price amount="7.75" currency="GBP"/>
    <price amount="30000.00" currency="MXP"/>
  </quote>
</quote-list>
```

After retrieving the JDOM document, we wrap it in a JavaBean component. The reason for this is to keep the XML manipulation code out of our JSP. By

storing this JavaBean in the `HttpRequest` object, we make it available to whichever JSP renders the view. The code for this bean is given in listing 5.3. Notice that the bean is also a custom tag, extending `javax.servlet.jsp.tagext.TagSupport` and implementing the `doStartBody` method. This allows us to dynamically generate the watch list presentation markup within the bean and eliminate all code from our JSPs.

Listing 5.3 The XMLHelper JavaBean/tag

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.util.*;

import org.jdom.*;

public class XMLHelperBean extends TagSupport {

    private String firstName;
    private String lastName;
    private String quoteTime;
    private String quoteDate;
    private Vector quotes = new Vector();

    private boolean useLinks=false;
    private String currency = "USD"

    public XMLHelperBean(Document doc) {
        Element root = doc.getRootElement();
        this.quoteTime = root.getAttributeValue("time");
        this.quoteDate = root.getAttributeValue("date");
        Element customer = root.getChild("customer");
        this.firstName = customer.getAttributeValue("first-name");
        this.lastName = customer.getAttributeValue("last-name");
        // build quote list
        List quoteElements = root.getChildren("quote");
        Iterator it = quoteElements.iterator();
        while (it.hasNext()) {
            Element e = (Element) it.next();
            Quote quote = new Quote();
            quote.symbol = e.getAttributeValue("symbol");
            quote.name = e.getAttributeValue("name");
            List priceElements = e.getChildren("price");
            Iterator it2 = priceElements.iterator();
            while (it2.hasNext()) {
                Element pe = (Element) it2.next();
                quote.prices.put(
                    pe.getAttributeValue("currency"),
```

JavaBeans
propertiesCustom tag
attributes ;

```

        pe.getAttributeValue("amount")
    );
    }
}

public String getFirstName() {
    return firstName;
}

public String getLastName() {
    return lastName;
}

public String getQuoteTime() {
    return quoteTime;
}

public String getQuoteDate() {
    return quoteDate;
}

// supplied only for consistency with
// JavaBeans component contract - inoperative
public void setFirstName(String s) {}
public void setLastName(String s) {}
public void setQuoteTime(String s) {}
public void setQuoteDate(String s) {}

public void setUseLinks(String yesno) {
    if (yesno.equalsIgnoreCase("yes"))
        useLinks = true;
    else
        useLinks = false;
}

public boolean getUseLinks() { return useLinks; }

public void setCurrency(String currency) {
    this.currency = currency;
}

public String getCurrency() { return currency; }

public int doStartTag() {
    try {
        JspWriter out = pageContext.getOut();
        out.print("<tr><th>Stock Symbol</th>");
        out.print("<th>Company Name</th><th>Last Price</th>");
        if (useLinks)
            out.print("<th>Easy Actions</th>");
        out.print("</tr>");
        for (int i = 0; i < quotes.size(); i++) {
            Quote q = (Quote) quotes.get(i);
            out.print("<tr><td>");

```

← Dynamically builds
the watch list table in
either HTML or WML

```

        out.print(q.symbol);
        out.print("</td><td>");
        out.print(q.name);
        out.print("</td><td>");
        out.print(q.prices.get(currency));
        out.print(" ");
        out.print(currency);
        out.print("</td>");
        if (useLinks) {
            out.print("<td><a href=\"http://www.exampleco.com/
buyStock?symbol=");
            out.print(q.symbol);
            out.print("\>buy</a>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;");
            out.print("<a href=\"http://www.exampleco.com/sellStock?symbol=");
            out.print("\>sell</a></td>");
        }
        out.print("</tr>");
    }
} catch(IOException ioe) {
    System.out.println("Error in XMLHelperBean.doStartTag(): " + ioe);
}
return(SKIP_BODY);
}

private class Quote {
    Hashtable prices = new Hashtable();
    String symbol;
    String name;
}
}

```

Using the class from listing 5.3, we can eliminate all Java code from our JSP. Hopefully even this trivial example gives you an appreciation of the amount of design and development work necessary to make your JSP code-free.

Rendering the output

Now we need JSPs. We could develop one enormous JSP with a bunch of control structures in it. Such a page could handle the two output formats and two locales in our example, but it would be quite complex and difficult to maintain over time. Such a page would hardly be independent of the data either.

We could develop two JSPs instead. One would produce HTML and the other WML. If the page layout does not change across locales, we could put selection logic for all the locale-specific text and images into our custom tag. But, we would still have to deal with the locales with branching statements.

Instead, we decided to write four very specific and simple JSPs, one for each permutation of output formats and locales we need to service. See listings 5.4 through 5.7. Each JSP will obtain a handle to the `XMLHelper` JavaBean that we stored in the request object to render its output.

Listing 5.4 The U.S. English, HTML watch list JSP

```
<jsp:useBean name="helper" scope="page"
             class="examples.chapter5.XMLHelperBean" />
<%@ taglib uri="example.tld" prefix="helperTag" %>
<html>
<head><title>Your Watch List</title></head>
<body>
<h1>Your Stock Price Watch List</h1>
<h3>
  Hello,
  <jsp:getProperty name="helper" property="firstName" />!
</h3>
<h3>
  Here are the latest price quotes for
  your watch list stocks.
</h3>
<p><i>
  Price quotes were obtained at
  <jsp:getProperty name="helper" property="quoteTime" />
  on
  <jsp:getProperty name="helper" property="quoteDate" />.
</i></p>
<table cellpadding="5" cellspacing="0" border="1">
<helperTag:printData useLinks="yes" currency="USD" />
</table>
</body>
</html>
```

Recoveres the JavaBean from the request object

Specifies prefix for use as a tag

Invokes the `doStartTag()` of our `XMLHelperBean`

- ① Because we specify `useLinks=yes`, our custom tag prints an HTML table consisting of four columns with links to buy and sell individual stocks. Setting `currency=USD` will print all stock prices in U.S. dollars.

The JSP that produces the U.S. English, HTML version of our page is shown in listing 5.4, with its British counterpart given in listing 5.5. Note both the

similarities and the differences between the two. The U.S. page uses a bit less formal language and expresses prices in USD. The British version is more formal and shows pricing in British pounds. However, they both use the identical data source and neither has any Java code in it.

Listing 5.5 The British English, HTML watch list JSP

```
<jsp:useBean name="helper" scope="page"
             class="examples.chapter5.XMLHelperBean" />
<%@ taglib uri="example.tld" prefix="helperTag" %>
<html>
<head><title>Your Watch List</title></head>
<body>
<h1>Your Stock Price Watch List</h1>
<h3>
  Greetings, Mr.
  <jsp:getProperty name="helper" property="lastName" />!
</h3>
<h3>
  Here are the latest prices on
  your stocks of interest.
</h3>
<p><i>
  Price quotes as of
  <jsp:getProperty name="helper" property="quoteTime" />
  on
  <jsp:getProperty name="helper" property="quoteDate" />.
</i></p>
<table cellpadding="5" cellspacing="0" border="1">
  <helperTag:printData useLinks="yes" currency="GBP" />
</table>
</body>
</html>
```

Now all that remains is to develop the WML versions of the watch list page. These pages contain a bit less textual information and do not have links to directly buy individual stocks. If you need to refresh your memory on what each of these pages looks like, flip back to figures 5.1 through 5.4.

Listing 5.6 The U.S. English WML watch list JSP

```

<jsp:useBean name="helper" scope="page"
  class="examples.chapter5.XMLHelperBean"/>
<%@ taglib uri="example.tld" prefix="helperTag" %>

<wml>
  <card id="main" title="Your Watch List">
    <do type="accept" name="do-back" label="Back">
      <go href="http://www.exampleco.com/home.wml" />
    </do>
    <do type="accept" name="do-buy" label="Buy Shares">
      <go href="http://www.exampleco.com/buyShares" />
    </do>
    <do type="accept" name="do-sell" label="Sell Shares">
      <go href="http://www.exampleco.com/sellShares" />
    </do>
    <p>
      <b>
        Hello,
        <jsp:getProperty name="helper" property="firstName"/>!
      </b>
    </p>
    <p>Here are your latest watch list price quotes:</p>
    <table columns="3">
      <helperTag:printData useLinks="no" currency="USD" />
    </table>
  </card>
</wml>

```

Prints the WML-version
without links

Listing 5.7 The British English WML watch list JSP

```

<jsp:useBean name="helper" scope="page"
  class="examples.chapter5.XMLHelperBean"/>
<%@ taglib uri="example.tld" prefix="helperTag" %>

<wml>
  <card id="main" title="Your Watch List">
    <do type="accept" name="do-back" label="Back">
      <go href="http://www.exampleco.com/home.wml" />
    </do>
    <do type="accept" name="do-buy" label="Buy Shares">
      <go href="http://www.exampleco.com/buyShares" />
    </do>
    <do type="accept" name="do-sell" label="Sell Shares">
      <go href="http://www.exampleco.com/sellShares" />
    </do>
    <p><b>Greetings, Mr.
      <jsp:getProperty name="helper" property="lastName"/>!
    </b></p>

```

```
<p>Here are the latest prices on your stocks of interest.</p>
<table columns="3">
  <helperTag:printData useLinks="no" currency="GBP" />
</table>
</card>
</wml>
```

5.2.4 *Analyzing the results*

We think you will admit that building a multidevice, multilocale presentation layer in J2EE is not easy. Imagine how much worse things could have been if we had chosen some less-complimentary output formats or wildly different locales. Also consider how much effort would be involved in extending our example code to accommodate more languages, locales, or device types.

This is the area in which an XML-based presentation layer can come to the rescue to some extent. While serving substantially different views of the same application is always challenging, some XML tools have recently emerged to make the process a bit easier for you. That is, after you learn to use them.

5.3 *The J2EE/XML approach*

The first XML architectural alternative we examine involves the combined use of XSLT with the J2EE presentation components. In chapter 2, we talked briefly about what XSLT is and how to use it via JAXP. You will recall that XSLT provides a general way to transform XML into virtually any output format. This comes in very handy when generating thin-client user interfaces like the one we have been working with in our example.

The output format of an XSLT process is determined by the transformation rules defined within an XSL stylesheet. In this section the desired output formats are HTML and WML. And to emphasize the capabilities of XSLT, we use it to generate PDFs from XML in this section too.

If you are still a bit fuzzy on XSLT concepts, you can learn more of the basics from the XSLT references in the bibliography or via an online tutorial. An excellent introduction to XSLT can be found online at <http://www.zvon.org>.

5.3.1 *Adding XSLT to the web process flow*

In section 5.2, we created a controller servlet, a custom tag/JavaBean component, and four JSPs to render the watch list page. Adding XSLT processing to the mix has the following impact on our design:

- The JSPs are no longer necessary.
- The custom tag/JavaBean component is no longer necessary.
- We require a new, outbound filter component to handle the XSLT process.
- We must modify our `WatchListJSPServlet` to remove the JSP forwarding code.

Filters are the latest addition to the J2EE presentation framework. They are useful for chaining together a web processing workflow. A filter can be applied to a specific type of request or globally across your entire application. Filters can perform preprocessing of a request (before it reaches the servlet) or postprocessing. In our example case, we are interested in postprocessing any request that is handled by our Controller servlet.

The XSLT request handling flow now begins when our controller receives a request for a stock watch list. The servlet interacts with the application logic layer via the `ListBuilder` component, which still returns its result in the form of a `JDOM Document`. The view selection logic is now handled by our new filter component, which selects among XSL stylesheets instead of JSP pages.

This new architecture is an implementation of the Decorating Filter J2EE pattern, which you can learn more about in appendix A. Figure 5.6 depicts our implementation graphically.

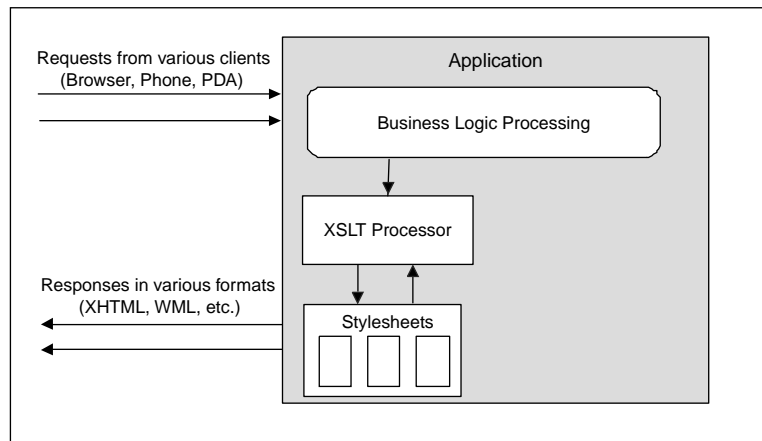


Figure 5.6 XSLT web request handling

The XSLT filtering process

We begin our XSLT example by developing the filter that will manage the stylesheet selection and XSLT transformation process. Here is an overview of how the filter works:

- 1 Each web request for the watch list page is intercepted by the filter and passed off to our controller servlet for processing.
- 2 The `JDOM Document` is returned to the filter via the `HttpRequest` object.
- 3 The filter determines the device type and preferred locale of the requesting client, just as our original servlet used to do.
- 4 The filter selects the most appropriate XSL stylesheet and invokes an XSLT processor to transform the JDOM results into a target output format
- 5 The filter sends the result of the XSLT transformation back to the user's device, where it is rendered for display.

For the sake of clarity, we oversimplify the view selection and XSLT transformation process. As discussed in chapter 2, special attention must be paid to precompiling and caching stylesheets in a production application, due to the performance characteristics of XSLT. In this version of our example, we concentrate on the interesting part, which is the transformation itself.

Modifying the servlet

Because our filter will select the appropriate stylesheet and no JSPs need to be invoked, our modified `WatchListServlet` component becomes quite simple. Its source code is shown in listing 5.8. The servlet now interacts with the `ListBuilder` interface and stores the `JDOM Document` in the `HttpRequest` object.

Listing 5.8 The modified watch list servlet

```
import org.jdom.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * The stock watchlist servlet with XSLT
 */
public class WatchListServlet extends HttpServlet {

    private ListBuilder builderInterface = new ListBuilder();
    private ServletConfig config;
    private ServletContext context;
```

```

public WatchListServlet() { super(); }

public void init(ServletConfig config)
    throws ServletException {
    this.config = config;
    this.context = config.getServletContext();
}

public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    HttpSession session = request.getSession(false);
    if (session == null) {
        context.getRequestDispatcher("/login.jsp")
            .forward(request, response);
        return;
    }
    String userId = (String) session.getAttribute("userId");
    Document quoteList = builderInterface.getWatchList(userId);
    request.setAttribute("quoteList", quoteList);
}

public void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
}

```

← No need to wrap the Document anymore

Building the filter

Our filter implementation is a class that implements the `javax.servlet.Filter` interface. The J2EE web container invokes the filter based on URL pattern matching, as it does with servlets. When a filter matching a specific URL pattern is found, the container invokes its `doFilter` method, the signature of which is as follows:

```

public void doFilter(ServletRequest request,
    ServletResponse response,
    FilterChain chain)
    throws IOException, ServletException;

```

The `FilterChain` parameter is a representation of all the processing to be done during this request, including the servlet(s), JSP(s), and other filter(s) that may be invoked. Since our filter is a postprocessing one, the first thing we do in our `doFilter` method is execute the `FilterChain`:

```

chain.doFilter(request, response);

```

This, in effect, calls the `WatchListServlet` via the web container, which places the `JDOM Document` in which we are interested into the request object. We then access the `Document` in the `doFilter` method.

```
HttpServletRequest httpRequest = (HttpServletRequest) request;
Document outputDoc = (Document) httpRequest.getAttribute("quoteList");
```

Next, we call some helper methods to determine the appropriate stylesheet for the transformation.

```
String outputFormat = getOutputFormat(httpRequest);
String locale = getLocaleString(httpRequest);
String stylesheetPath = getStylesheet(outputFormat, locale);
```

You can see the bodies of these methods in listing 5.9. They are similar to those of our earlier `WatchListJSPServlet` component. Now that we have the XML document and know which stylesheet we want to use, we perform the transformation via the JAXP API.

```
TransformerFactory myFactory = TransformerFactory.newInstance();
Transformer myTransformer = myFactory.newTransformer(new
    StreamSource(stylesheetPath));
JDOMResult result = new JDOMResult();
myTransformer.transform( new JDOMSource( outputDoc ), result );
```

Now, all that is left to do is write the XSLT output back to the client, via the `HttpResponse` object.

```
Document resultDoc = result.getDocument();
XMLOutputter xOut = new XMLOutputter();
if (outputFormat.equals("wml"))
    response.setContentType("text/vnd.wap.wml");
PrintWriter out = response.getWriter();
xOut.output( resultDoc, out );
```

Listing 5.9 provides the complete implementation of our `XSLTFilter` class.

Listing 5.9 XSLT filter code

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import org.jdom.*;
import org.jdom.output.*;
import org.jdom.transform.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
```

```
public class XSLTFilter implements Filter {
    private FilterConfig filterConfig;

    public void init(FilterConfig filterConfig)
        throws ServletException {
        this.filterConfig = filterConfig;
    }

    public FilterConfig getFilterConfig() {
        return this.filterConfig;
    }

    public void setFilterConfig(FilterConfig filterConfig) {
        this.filterConfig = filterConfig;
    }

    public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain)
        throws IOException, ServletException {
        try {
            chain.doFilter(request, response);
            HttpServletRequest httpRequest
                = (HttpServletRequest) request;
            Document outputDoc
                = (Document) httpRequest.getAttribute("quoteList");
            if (outputDoc == null) return;

            String outputFormat = getOutputFormat(httpRequest);
            String locale = getLocaleString(httpRequest);
            String stylesheetPath
                = getStylesheet(outputFormat, locale);

            TransformerFactory myFactory
                = TransformerFactory.newInstance();
            Transformer myTransformer
                = myFactory.newTransformer(
                    new StreamSource(stylesheetPath));

            JDOMResult result = new JDOMResult();
            myTransformer.transform(
                new JDOMSource( outputDoc ), result );
            Document resultDoc = result.getDocument();
            XMLOutputter xOut = new XMLOutputter();
            if (outputFormat.equals("wml"))
                response.setContentType("text/vnd.wap.wml");
            PrintWriter out = response.getWriter();
            xOut.output( resultDoc, out );

        } catch (Exception e) {
            System.out.println("Error was:" + e.getMessage());
        }
    }
}
```

```
private String getOutputFormat(HttpServletRequest request) {
    String userAgent = request.getHeader("User-Agent");
    // this is where your robust user-agent lookup should happen
    if (userAgent.indexOf("UP.Browser") >= 0)
        return "wml";
    return "html";
}

private String getLocaleString(HttpServletRequest request) {
    Enumeration locales = request.getHeaders("Accept-Language");
    while (locales.hasMoreElements()) {
        String locale = (String) locales.nextElement();
        if (locale.equalsIgnoreCase("en_GB"))
            return "en_GB";
    }
    return "en_US";
}

private String getStylesheet(String outputFormat, String locale) {
    if (locale.equals("en_US")) {
        if (outputFormat.equals("html"))
            return "watchlist.html.en_US.xsl";
        else
            return "watchlist.wml.en_US.xsl";
    } else {
        if (outputFormat.equals("html"))
            return "watchlist.html.en_GB.xsl";
        else
            return "watchlist.wml.en_GB.xsl";
    }
}

public void destroy() {}
}
```

Developing the stylesheets

The final four pieces of our new, XSLT-enabled solution are the four XSL stylesheets used to transform the XML into output. We need to convert the JSPs, JavaBeans, and custom tag code developed in section 5.2 into four sets of XSLT transformation rules. Although there are a few different ways to develop an XSL stylesheet, the most straightforward is the template-based approach. XSL stylesheets developed in this manner most closely resemble the JSP templates with which you are already familiar.

Listing 5.10 contains the XSL stylesheet used to convert the quotes-list XML document into HTML format for U.S. users. Note the resemblance of this file to an HTML source file. The major differences are the wrapping of the entire document in an `<xsl:stylesheet>` element and a global `<xsl:template>` ele-

ment. The `<xsl:stylesheet>` element identifies this file as a set of transformation rules, and the `<xsl:template>` element is our global transformation rule that will be applied to the root node of the XML source document.

A thorough analysis of XSLT development is beyond the scope of this book. Note, however, the XML-based control structures (e.g., `<xsl:for-each>`) and variable substitution (e.g., `<xsl:value-of>`) that can be accomplished in these stylesheets. XSLT is a powerful tool for transforming XML in a variety of ways.

Listing 5.10 The U.S. English HTML-producing stylesheet

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:output method="html"
    doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
    />

  <xsl:template match="/">
    <html>
    <head><title>Your Watch List</title></head>
    <body>
    <h1>Your Stock Price Watch List</h1>

    <h3>
      Hello,
      <xsl:value-of select="/customer[@first-name]"/>
    </h3>
    <h3>
      Here are the latest price quotes for
      your watch list stocks.
    </h3>
    <p><i>
      Price quotes were obtained at
      <xsl:value-of select="/quote-list[@time]"/>
      on
      <xsl:value-of select="/quote-list[@date]"/>
    </i></p>

    <table cellpadding="5" cellspacing="0" border="1">
    <tr>
      <th>Stock Symbol</th>
      <th>Company Name</th>
      <th>Last Price</th>
```

Allows us to use special HTML entities like `<` and ` `;

The beginning of our only global transformation rule

Uses XPath to select customer first name

Gets the generation time from the quotes-list document

Gets the generation date from the quotes-list document

ment the XSL and the developer can focus on the processing that generates the XML. These tasks can be performed relatively independently of each other.

The separation of roles advantage is not without its own challenges, however. For example, developing a user interface in XSL is far more difficult than using standard HTML and requires a programming skill-set that is uncommon among graphic designers. Graphical tools are expected to alleviate this problem somewhat in the future. Another challenge for the practicality of this architecture is the current lack of XSLT skill-sets in the industry. Though this will change over time, it has been an adoption hurdle for integrating XML into the presentation layer.

As we stated in chapter 2, performance will be the major factor in enabling the widespread deployment of XSLT-based presentation layers. It may produce an architecturally superior solution, but it must perform well to be adopted. All indications point toward a steady increase in XSLT performance as the technology matures.

5.3.3 *Extending to binary formats*

Before leaving the subject of interface rendering with XSLT, we should highlight the abilities of XSLT formatting objects. A popular requirement for advanced web applications today is to dynamically generate binary files that are more difficult or impossible to manipulate once generated.

DEFINITION *XSLT formatting objects (FO)* is a portion of the XSLT specification that defines the manner in which XML documents can be transformed into binary output via XSLT.

Using an implementation of XSLT formatting objects, you can transform your dynamic XML data into a binary format like PDF on the fly. To prove the point and see how it is done, we extend our watch list example to generate PDF files rather than HTML documents in this section.

For an implementation of XSL formatting objects, we turn once again to the Apache Software Foundation. ASF has a project called FOP that is currently a partial implementation of formatting objects. You can download the binaries, source, and documentation from <http://xml.apache.org/fop>.

For the sake of discussion, let us suppose that we need to output the stock watch list page in PDF format instead of HTML for web-based clients. Perhaps we have a fraud concern that someone might download the HTML source, modify it, and then claim that our pricing information was incorrect and cost

them money. Since it is difficult to steal and modify WML from a mobile device, we are only concerned with traditional, HTML web browsers.

Modifying our XSLT version of the example to generate PDF instead of HTML involves two steps:

- We must modify our HTML-producing stylesheets to produce an FOP formatting object tree instead of an HTML page.
- We must add a final step to the XSLT filter component to invoke the FOP API and convert the formatting object tree to a PDF document.

DEFINITION A *formatting object tree* is a specialized XSL stylesheet that contains print formatting instructions. The FOP `Driver` component uses these instructions to create a PDF file from an XML document.

The modified process flow for generating PDF instead of HTML is depicted in figure 5.7.

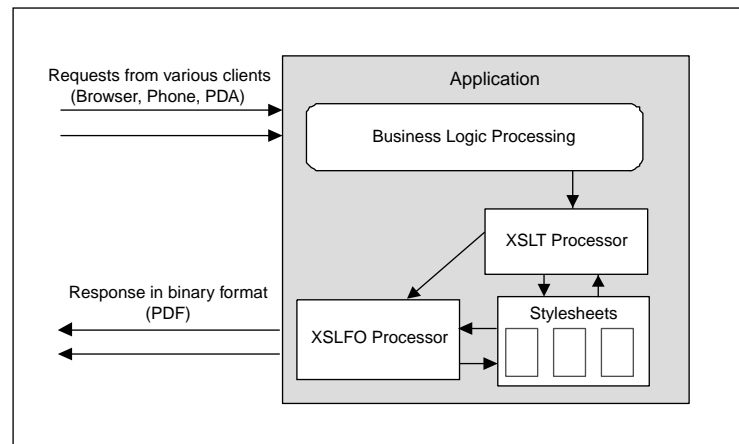


Figure 5.7 Dynamic PDF generation process flow

Creating a formatting tree

The basic structure of a formatting tree stylesheet is depicted in figure 5.8. The tree consists of two main components: layouts and page sequences. A *layout* describes a page template to be applied to one or more page sequences. Each *page sequence* defines the actual content that will appear in the PDF, including all of its formatting information. For our example, we create one layout (a master template)

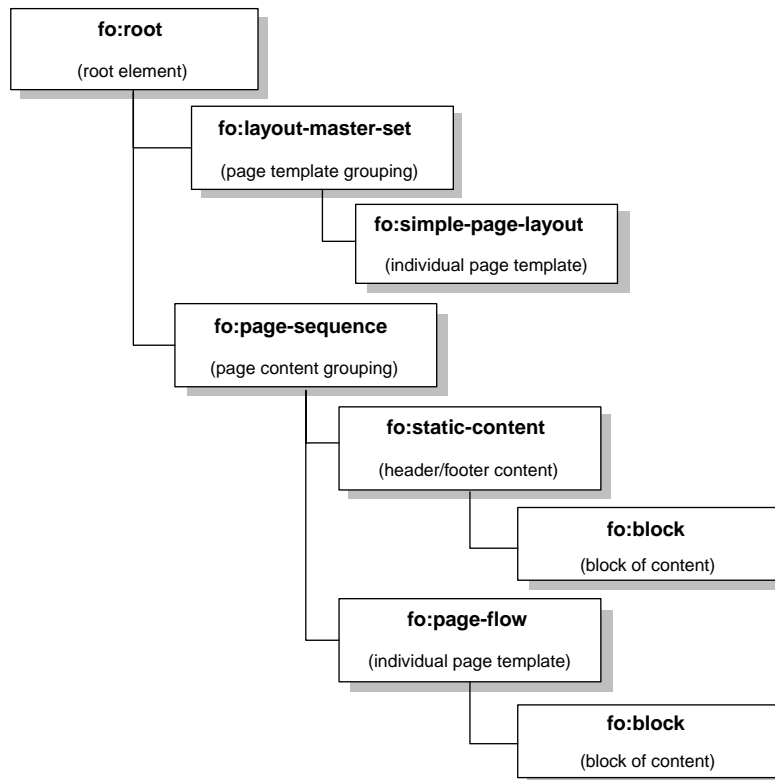


Figure 5.8 Logical structure of a FO formatting tree

and one page sequence. Our page sequence will contain a single page showing the same information as our HTML page did.

Listing 5.11 contains the complete formatting tree XSL stylesheet to produce the U.S. English version of the watch list page in PDF format.

Listing 5.11 Formatting tree for U.S. English stock watch list

```

<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  xmlns:fo="http://www.w3.org/1999/XSL/Format"
>
<xsl:template match = "/">

```

← Identifies the XSLT
FO XML
namespace

```

<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <!-- define page layout -->
  <fo:layout-master-set>
    <fo:simple-page-master master-name="simple"
      page-height="29.7cm"
      page-width="21cm"
      margin-top="1.5cm"
      margin-bottom="2cm"
      margin-left="2.5cm"
      margin-right="2.5cm">
      <fo:region-body margin-top="3cm"/>
      <fo:region-before extent="1.5cm"/>
      <fo:region-after extent="1.5cm"/>
    </fo:simple-page-master>
  </fo:layout-master-set>

  <!-- define the content -->
  <fo:page-sequence master-name="simple">

    <fo:static-content flow-name="xsl-region-before">
      <fo:block text-align="end"
        font-size="10pt"
        font-family="serif"
        line-height="14pt">
        Watch List - Customer
        <xsl:value-of
          select="./quote-list/customer/@id"/>
      </fo:block>
    </fo:static-content>

    <fo:flow flow-name="xsl-region-body">
      <fo:block font-size="16pt"
        font-family="sans-serif"
        font-weight="bold"
        line-height="26pt"
        space-after.optimum="12pt"
        background-color="blue"
        color="white"
        text-align="center">
        Your Stock Watch List
      </fo:block>
      <fo:block font-size="12pt"
        font-family="sans-serif"
        font-weight="bold"
        line-height="18pt"
        space-after.optimum="10pt"
        start-indent="10pt">
        Hello,
        <xsl:value-of

```

Defines a page template

Defines a page sequence, bound to our template above

Defines a global page header for the sequence

Prints customer id from the data document

Defines document contents

```

        select="./quote-list/customer/@first-name"/>
</fo:block>
<fo:block font-size="10pt"
  font-family="sans-serif"
  font-style="italic"
  line-height="18pt"
  space-after.optimum="10pt"
  start-indent="15pt">
  Prices were obtained at
    <xsl:value-of select="./quote-list/@time"/>
  on
    <xsl:value-of select="./quote-list/@date"/>
</fo:block>
<fo:table>
  <fo:table-column column-width="3cm"/>
  <fo:table-column column-width="7cm"/>
  <fo:table-column column-width="3cm"/>
<fo:table-header font-size="10pt"
  line-height="14pt"
  font-family="sans-serif">
<fo:table-row font-weight="bold">
  <fo:table-cell text-align="start">
    <fo:block>SYMBOL</fo:block>
  </fo:table-cell>
  <fo:table-cell text-align="start">
    <fo:block>COMPANY NAME</fo:block>
  </fo:table-cell>
  <fo:table-cell text-align="start">
    <fo:block>SHARE PRICE</fo:block>
  </fo:table-cell>
</fo:table-row>
</fo:table-header>
  <fo:table-body font-size="10pt"
  line-height="16pt"
  font-family="sans-serif">
    <xsl:for-each select="//quote">

      <fo:table-row>
        <fo:table-cell>
          <fo:block text-align="start" >
            <xsl:value-of select="@symbol"/>

          </fo:block>
        </fo:table-cell>
        <fo:table-cell>
          <fo:block text-align="start" >
            <xsl:value-of select="@name"/>

          </fo:block>
        </fo:table-cell>
        <fo:table-cell>
          <fo:block text-align="start" >

```

← Constructs the watch list table, just as we did in textual formats

```

        $ <xsl:value-of
            select="./price[@currency='USD']/@amount"/>
        </fo:block>
    </fo:table-cell>
</fo:table-row>
</xsl:for-each>
</fo:table-body>
</fo:table>
</fo:flow>
</fo:page-sequence>
</fo:root>
</xsl:template>
</xsl:stylesheet>

```

To make use of the FO stylesheet, we must invoke the Apache FOP API. Since we do not want to tie our filter implementation to a specific FO implementation, we chose to wrap the use of FOP with an adapter object called `PDFWriter`. This component takes a formatting tree stylesheet path and an XML input source and writes the PDF to a specified output stream. To do its work, the `PDFWriter` uses both Apache FOP and the JAXP API for XSLT. The code for this adapter is given in listing 5.12.

Listing 5.12 An adapter for Apache FOP

```

import java.io.*;

import org.xml.sax.InputSource;
import org.apache.fop.apps.Driver;
import org.apache.fop.apps.Version;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;

public class PDFWriter {
    protected Transformer transformer = null;
    public PDFWriter(StreamSource source)
        throws TransformerConfigurationException {
        TransformerFactory factory
            = TransformerFactory.newInstance();
        transformer = factory.newTransformer(source);
    }

    public PDFWriter(String xslFilePath)
        throws TransformerConfigurationException,
            FileNotFoundException {
        this(new StreamSource(new FileInputStream(xslFilePath)));
    }
}

```

Uses JAXP to transform the XML data into a FO tree

```

protected byte[] invokeFOP(InputSource foSource)
    throws Exception {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    Driver driver = new Driver(foSource, out);
    driver.run();
    return out.toByteArray();
}

public byte[] generatePDF(StreamSource xmlSource)
    throws Exception {
    ByteArrayOutputStream baos
        = new ByteArrayOutputStream();
    StreamResult foResult = new StreamResult(baos);
    transformer.transform(xmlSource, foResult);
    ByteArrayInputStream bais
        = new ByteArrayInputStream( baos.toByteArray() );
    return invokeFOP( new InputSource(bais) );
}

public static void createPDFFromXML(String xslFilePath,
    InputStream xmlIn,
    OutputStream pdfOut)

    throws Exception {
    PDFWriter writer = new PDFWriter(xslFilePath);
    byte[] PDFbytes
        = writer.generatePDF( new StreamSource(xmlIn) );
    pdfOut.write(PDFbytes, 0, PDFbytes.length);
}
}

```

Defines a set of page templates containing a single page named simple

Transforms XML to PDF and writes to an output stream

The last thing we need to do is modify our XSLT filter to use the `PDFWriter` when `html` is the output format. The modified `XSLTPDFFilter` class is shown in listing 5.13.

Listing 5.13 The modified XSLT filter class

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import org.jdom.*;
import org.jdom.output.*;
import org.jdom.transform.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;

public class XSLTFilter implements Filter {
    private FilterConfig filterConfig;

```

```
public void init(FilterConfig filterConfig)
    throws ServletException {
    this.filterConfig = filterConfig;
}

public FilterConfig getFilterConfig() {
    return this.filterConfig;
}

public void setFilterConfig(FilterConfig filterConfig) {
    this.filterConfig = filterConfig;
}

public void doFilter(ServletRequest request,
                    ServletResponse response,
                    FilterChain chain)
    throws IOException, ServletException {
    try {
        chain.doFilter(request, response);
        HttpServletRequest httpRequest
            = (HttpServletRequest) request;
        Document outputDoc
            = (Document) httpRequest.getAttribute("quoteList");
        if (outputDoc == null) return;

        String outputFormat = getOutputFormat(httpRequest);
        String locale = getLocaleString(httpRequest);
        String stylesheetPath
            = getStylesheet(outputFormat, locale);

        XMLOutputter xOut = new XMLOutputter();
        if (outputFormat.equals("html")) {
            ByteArrayOutputStream baos
                = new ByteArrayOutputStream();
            xOut.output(outputDoc, baos);
            ByteArrayInputStream bais
                = new ByteArrayInputStream( baos.toByteArray() );
            // get the response output stream
            response.setContentType("application/pdf");
            OutputStream out = response.getOutputStream();
            PDFWriter.createPDFFromXML( stylesheetPath, bais, out);
            out.close();
        } else {
            // wml format
            TransformerFactory myFactory
                = TransformerFactory.newInstance();
            Transformer myTransformer
                = myFactory.newTransformer(
                    new StreamSource(stylesheetPath));
            JDOMResult result = new JDOMResult();
            myTransformer.transform(
                new JDOMSource( outputDoc ), result );
        }
    }
}
```

Converts JDOM to an
input stream

Creates and
saves PDF

```

        Document resultDoc = result.getDocument();
        response.setContentType("text/vnd.wap.wml");
        PrintWriter out = response.getWriter();
        xOut.output( resultDoc, out );
    }
} catch (Exception e) {
    System.out.println("Error was:" + e.getMessage());
}
}
private String getOutputFormat(HttpServletRequest request) {
    String userAgent = request.getHeader("User-Agent");
    if (userAgent.indexOf("UP.Browser") >= 0)
        return "wml";
    return "html";
}
private String getLocaleString(HttpServletRequest request) {
    Enumeration locales = request.getHeaders("Accept-Language");
    while (locales.hasMoreElements()) {
        String locale = (String) locales.nextElement();
        if (locale.equalsIgnoreCase("en_GB"))
            return "en_GB";
    }
    return "en_US";
}
private String getStylesheet(String outputFormat, String locale) {
    if (locale.equals("en_US")) {
        if (outputFormat.equals("html"))
            return "watchlist.pdf.en_US.xsl";
        else
            return "watchlist.wml.en_US.xsl";
    } else {
        if (outputFormat.equals("html"))
            return "watchlist.pdf.en_GB.xsl";
        else
            return "watchlist.wml.en_GB.xsl";
    }
}
public void destroy() {}
}

```

Uses our FO stylesheets
instead of the HTML-
producing ones

Figure 5.9 shows the fruit of our labor, a dynamically generated PDF containing our stock watch list data.

SYMBOL	COMPANY NAME	SHARE PRICE
SRMC	Sierra Monitor Corporation	\$ 2.00
IBM	International Business Machines	\$ 135.00
ORCL	Oracle Corporation	\$ 15.00

Figure 5.9 The U.S. English PDF version of the stock watch list

5.4 XML web publishing frameworks

The architecture presented in the previous section requires custom code development that can be difficult and time-consuming. In this section, we explore a possible alternative to the custom integration work called web publishing frameworks.

DEFINITION A *web publishing framework* is a software suite design to speed the development of an XML-based presentation layer.

Web publishing frameworks combine Java and XML technologies into a cohesive and usable architecture. They are an out-of-the-box solution for generating your user interface. Using a web publishing framework, you can create a production-ready, XML-based presentation layer without having to write custom code that integrates XML into your architecture. These frameworks were built to solve the same challenge that we outlined earlier in this chapter—producing various views of the same content while maintaining a separation between presentation logic and style.

Web publishing frameworks that are based on XML technologies are still relatively new. Their reliability depends on the stability of underlying components, including the XML parser and XSLT processor. A few popular web publishing framework products include the following:

- Webmacro (<http://www.webmacro.org>)
- Enhydra (<http://www.enhydra.org>)
- Cocoon (<http://xml.apache.org/cocoon>)

For purposes of comparison with the XSLT approach from the previous section, we now explore how our watch list example could be developed within the Cocoon web publishing framework.

5.4.1 Introduction to Cocoon architecture

At the time of this writing, the Apache Software Foundation's Cocoon is one of the most stable and feature-rich XML web publishing frameworks. User interface development with Cocoon involves the creation of XSL stylesheets and XSP pages. Since XSP is a technology currently limited to Cocoon, we will concentrate on the more generic, XSLT capabilities of Cocoon in this section. Figure 5.10 depicts the Cocoon processing flow.

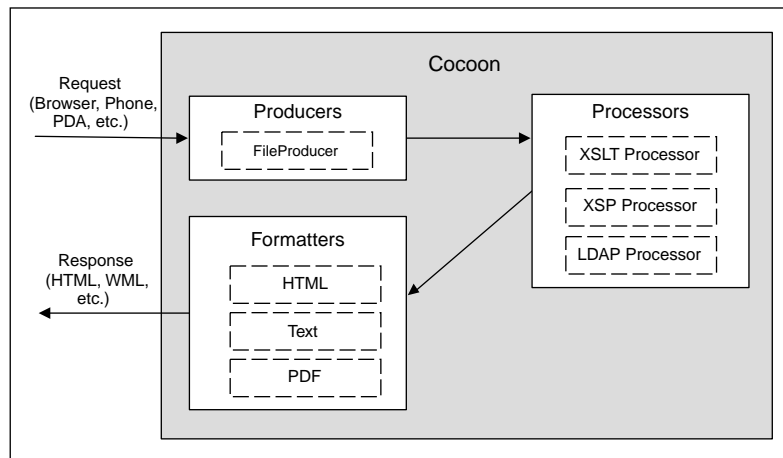


Figure 5.10 The Cocoon request processing flow

DEFINITION *XML Server Pages (XSP)* is a working draft specification for an XML-based program generation language. XSPs contain directives that control how a given XML data set is processed.

The simplest way to use Cocoon is to add special processing instructions to your XML data documents. These processing instructions allow Cocoon to process and format your data and deliver it to a requesting client. Supported output formats include WML, PDF, XML, and XHTML.

Cocoon producers

Producers are software components responsible for generating XML data. They are the equivalent of a servlet in that they receive and process an `HttpServletRequest`. This is just one of the areas in which Cocoon is extensible. You can implement your own producers to perform custom processing. Cocoon ships with a `FileProducer` that loads a requested file from the file system.

Cocoon processors

Once the data has been produced, it is available for processing. A *processor* is a component responsible for performing an operation such as an XSLT transformation on the XML data generated by a producer. Cocoon contains the following processors out-of-the-box:

- An XSLT processor
- An LDAP processor
- An XSP processor

Writing your own processor is similar to writing a JSP custom tag. The tag is created, associated with some behavior, and included in a page.

Cocoon formatters

Formatters are helper components that may be applied to a response before it is returned to the requesting client. Formatters are used to wrap output content with additional formatting information. Formatters do things such as placing tags around such markup content as HTML documents and creating a final PDF from a XSLFO formatting tree.

5.4.2 Using Cocoon to render the watch list page

Let us put Cocoon to work on our example XML document. We will use the standard Cocoon XSLT processor to perform an XSLT transformation on our quote-list XML. Fortunately for us, we already developed the XSL required to make this work in section 5.2.

The only modification we need to make to the XML document returned from the application logic layer (the `ListBuilder` interface in the example) is to

add a Cocoon processing instruction and references to our XSL stylesheets within the XML data document. The Cocoon directive is as follows:

```
<?cocoon-process type=xslt ?>
```

Then we add two processing instructions that describe the HTML and WML stylesheets to be applied to the data. For the U.S. locale, the instructions are as follows:

```
<?xml-stylesheet href=watchlist.html.en_US.xsl type=text/xsl ?>  
<?xml-stylesheet href=watchlist.wml.en_US.xsl type=text/xsl  
  media=wap ?>
```

The `media=wap` attribute in the second processing instruction tells Cocoon to select this stylesheet for WML-based clients. In other cases, the default stylesheet will be used.

Cocoon is designed to be accessed as a servlet, but can be invoked via an API call as well. In listing 5.14, our modified `WatchListServlet` adds the appropriate processing instructions to the XML data document and invokes the Cocoon engine to perform the transformation and delivery back to the client.

Listing 5.14 Invoking Cocoon from the watch list servlet

```
import org.jdom.*;  
  
import java.io.*;  
import java.util.*;  
import javax.servlet.*;  
import javax.servlet.http.*;  
  
import org.apache.cocoon.Engine;  
import org.apache.cocoon.util.CocoonServletRequest;  
  
public class WatchListServletWithCocoon extends HttpServlet {  
  
    private ListBuilder builderInterface = new ListBuilder();  
    private ServletConfig config;  
    private ServletContext context;  
  
    public WatchListServletWithCocoon() { super(); }  
  
    public void init(ServletConfig config)  
        throws ServletException {  
        this.config = config;  
        this.context = config.getServletContext();  
    }  
  
    public void doGet(HttpServletRequest request,  
        HttpServletResponse response)  
        throws ServletException, IOException {
```

```

// get userid from HttpSession
HttpSession session = request.getSession(false);
if (session == null) {
    context.getRequestDispatcher("/login.jsp")
        .forward(request, response);
    return;
}
String userId = (String) session.getAttribute("userId");
Document quoteList =
    builderInterface.getWatchList(userId);

String localeString = getLocaleString(request);
String document
= getOutputDocWithProcessingInstructions(quoteList,
                                       localeString);

try {
    Engine cocoonEngine = Engine.getInstance();
    CocoonServletRequest myReq
        = new CocoonServletRequest(document, request);
    cocoonEngine.handle(myReq, response);
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
}

private String getLocaleString(HttpServletRequest request) {
    Enumeration locales
        = request.getHeaders("Accept-Language");
    while (locales.hasMoreElements()) {
        String locale = (String) locales.nextElement();
        if (locale.equalsIgnoreCase("en_GB"))
            return "en_GB";
    }
    return "en_US";
}

private String
getOutputDocWithProcessingInstructions(Document document,
                                       String locale) {
    if (locale.equals("en_US")) {
        document.addContent(
            new ProcessingInstruction("xml-stylesheet",
                "href=\"watchlist.html.en_US.xsl\" type=\"text/xsl\""));
        document.addContent(
            new ProcessingInstruction("xml-stylesheet",
                "href=\"watchlist.wml.en_US.xsl\" type=\"text/xsl\"
                media=\"wap\""));
    } else {
        document.addContent(
            new ProcessingInstruction("xml-stylesheet",
                "href=\"watchlist.html.en_GB.xsl\" type=\"text/xsl\""));
    }
}

```

Adds processing instructions for either locale

Obtains a handle to the Cocoon Engine

Wraps document as an HttpRequest

Invokes Cocoon to perform the XSLT transformation

Adds processing instructions for specified locale ①

```

        document.addContent(
            new ProcessingInstruction("xml-stylesheet",
                "href=\"watchlist.wml.en_GB.xsl\" type=\"text/xsl\"
                media=\"wap\""));
        }
        return document.toString();
    }

    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}

```

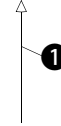


Figure 5.11 depicts our new presentation layer that combines Cocoon and the J2EE presentation layer components.

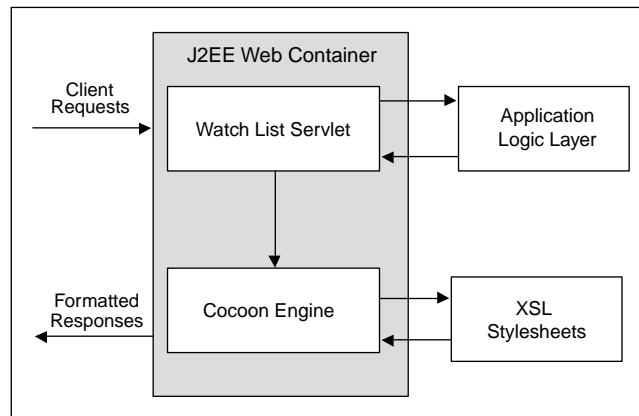


Figure 5.11
The J2EE Cocoon
presentation layer

5.4.3 Analyzing the results

The primary advantage of using a web publishing framework like Cocoon is the avoidance of writing a significant amount of Java code, which would otherwise be necessary to make the XSLT transformation process happen. From a development standpoint, you need only supply XML documents and XSL stylesheets to use the Cocoon framework. Web publishing frameworks also feature caching algorithms to optimize performance.

This convenience does come with a price. Despite precompiling stylesheets and caching at various levels, XSLT transformation does not perform as well as compiled templates, no matter what framework you use. Additionally, the

J2EE presentation framework is fast, reliable, and adequately satisfies most user interface needs.

This begs the question—should you replace your JSP infrastructure with XSLT, XSP, and Cocoon? At this point, we believe it is advisable to only augment your J2EE infrastructure with Cocoon and use XSP sparingly. Cocoon technology has a lot of potential, but is not yet mature enough to warrant redesigning your user interface. For now, the use of web publishing frameworks should be limited to the portion of your application that has advanced interface requirements like those described in this chapter.

5.5 *A word about client-side XSLT*

The architectures explored in this chapter perform XSLT transformations focused exclusively on the server side. This is because most J2EE applications use the thin-client model, wherein the server is required to do the work. Most web browsers still do not support client-side XSLT or they lack the processing power to perform transformations (e.g., wireless device browsers). This may change in the future, and a model in which the client side performs the transformation might become possible.

One of the potential advantages of the client-side approach is the relief of a heavy burden from the server-side, leading to faster responses and higher application throughput. Another advantage may be the ability to perform specialty processing such as conversion to voice or Braille. Client-side processing could be an integrated feature of the browser or implemented using applets. Currently, Internet Explorer 5 is the only popular browser with any real XML support.

5.6 *Summary*

This chapter was all about creating a more robust, flexible user interface using XML within the J2EE framework. We began by reviewing the common challenges involved in the development of a thin-client user interface to serve multiple types of users connecting via various types of devices. We detailed the difficulties involved in creating and maintaining a multilocale, multidevice presentation layer using J2EE alone. We then discussed two alternatives to the pure J2EE approach made available by the advent of XML technology.

The first alternative is to add XSLT processing to your J2EE request handling process. The recent addition of filters to the servlet API makes adding

XSLT much easier using the Decorating Filter pattern. Still, you have to roll your own code to take this approach.

The second alternative is to use XSLT via a web publishing framework like Apache Cocoon. This approach allows you to avoid custom coding of the XSLT extensions to your presentation layer and offers some potential performance benefits from caching algorithms and other means. XSLT is much slower than using compiled templates, but can be a reasonable alternative for large, complicated interfaces that have advanced requirements such as multilocale and/or multidevice support.

If you decide that a hybrid J2EE/XML approach is not right for your application, you should investigate other pure J2EE presentation frameworks to give your user interface development efforts a jump start. Two such popular frameworks are Struts and Velocity. Information on both can be found at <http://jakarta.apache.org>.

This chapter concludes our detailed discussion of using XML at each layer of your n-tier J2EE application. The next and final chapter summarizes the concepts and technologies from the first five chapters in the context of a cohesive case study.