JSP Examples and Best Practices

ANDREW PATZER

Apress™

JSP Examples and Best Practices

ISBN (pbk): 1-59059-020-1

Printed and bound in the United States of America 12345678910

Technical Reviewer: David Czarnecki

Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski

Managing Editor: Grace Wong

Project Manager: Alexa Stuart

Copy Editor: Kim Wimpsett

Production Editor: Kari Brooks

Compositor: Impressions Book and Journal Services, Inc.

Indexer: Carol Burbo

Cover Designer: Tom Debolski

Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710.

Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 5

Development Using Patterns

A KEY ADVANTAGE TO USING JAVA TECHNOLOGY is that it's an Object-Oriented (OO) language. This enables you to write code that is reusable and highly scalable. As you become more accustomed to OO development, you might recognize a few *best practices* that you follow when developing solutions of a particular class. For instance, you may find that for every data-entry application you work on, you tend to code the data validation routines in a similar way. If you were to formalize this best practice and abstract away some of the implementation details, it's conceivable that others could use it as a roadmap to jumpstart their own development efforts by implementing an already proven technique for data validation. This eliminates a lot of design effort as well as numerous testing iterations.

Published best practices have come to be known as *design patterns*. These originated in the OO world and have been published in various forms specific to their implementations in C++ and Java, as well as several general studies. In particular, the book *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995) has become the definitive guide to OO design patterns. Recently, the concept of design patterns has influenced the J2EE developer community, prompting Sun to publish a *J2EE Patterns Catalog* (`http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/`). These patterns address typical design issues and how you can apply the various J2EE technologies to solve such issues.

This chapter is the first of several that will deal with enterprise design patterns. I'll discuss the merits of using patterns, review the *J2EE Patterns Catalog*, highlight the patterns relevant to the presentation tier (and therefore the subject matter of this book), and finish with a discussion of the Model-View-Controller (MVC) pattern upon which most of the J2EE patterns are based.

Why Use Patterns?

I'll begin the patterns coverage by answering this question: Why should you look to design patterns for help with your own development efforts? The answer to this question is simple. Design patterns are proven techniques that can be reused and applied to many similar problems. They also provide you with a common vocabulary when discussing application design.

They're Proven Techniques

When designing an application, many problems need to be solved. In most cases, these problems are not unique to the specific application you're designing. If you were to design and implement a custom solution to the problem, then that piece of code will need to undergo perhaps several iterations of testing and subsequent coding until it's exactly what you need for your particular application.

If you were to take the previous scenario and use a design pattern instead of a custom solution, then you would greatly reduce development and testing time. The design pattern has already undergone many iterations of testing and development to produce an industry-wide best practice. Obviously, you'll still need to do some custom development to implement the pattern, but now you only need to test the implementation-specific code and not the entire piece of functionality.

They're Reusable

In the spirit of OO design, enterprise design patterns are intended to be reused across projects. Each pattern provides a proven solution for a specific class of problems. These problems tend to exist in many different applications. Rather than reinvent the wheel each time, it makes more sense to apply a design pattern requiring minimal modifications.

It's a Common Vocabulary

When speaking of application design, it helps to have a common vocabulary to communicate your options to the rest of the development team. For instance, a common OO design pattern is the factory pattern. This pattern is useful when an object needs to be instantiated at runtime, but the class of that object is not known at compile-time. So, when discussing design options, you might say something such as, "Well, if we implement a factory pattern in the reporting module, we can add new reports in the future without modifying the application framework." If everyone on the team understands the factory pattern, they can all envision the solution based on the given statement.

Introducing the *J2EE Patterns Catalog*

The architects at Sun have compiled a series of design patterns and published them as the *J2EE Patterns Catalog* available at the Java Developer Connection website (`http://developer.java.sun.com`). These patterns address common

application problems through the application of J2EE technologies. The patterns catalog groups the various patterns into the following tiers:

- **Presentation tier**: Whatever is required to present application data and user interface elements to the user is inside of the presentation tier of the application. Key technologies in use are JavaServer Pages (JSP) and Java Servlets.

- **Business tier**: The business tier is where all the business processing takes place. The primary J2EE technologies in use for this tier are Enterprise JavaBeans (EJBs).

- **Integration tier**: The integration tier provides connections to the resource tier. The resource tier includes things such as message queues, databases, and legacy systems. The J2EE technologies in use at the integration tier are the Java Message Service (JMS), Java Database Connectivity (JDBC), and the Java Connector Architecture (JCA).

Because this is a JSP book, I'll mostly present those patterns that deal with the presentation tier. I won't attempt to describe each pattern in detail; the patterns catalog does a fine job of that. The goal of this book is to provide best practices and examples. To that end, I'll provide enough definition to enable you to apply these patterns to common development tasks using JSP pages and Java servlets.

Looking at Presentation Design Patterns

The patterns I'll discuss in this book are commonly known as the Decorating Filter, Front Controller, Dispatcher View, and View Helper patterns. There are a few more presentation patterns in the J2EE catalog that I won't discuss. These four patterns are sufficient to illustrate the examples and best practices that I'll cover.

These patterns each cover a different *layer* of the presentation logic. As the request comes in, it can pass through a filter prior to the actual processing of the request (Decorating Filter pattern). It could then go to a centralized servlet to be processed (Front Controller Pattern). Once it has been processed, the servlet could then dispatch the results to a specific JSP page (Dispatch View Pattern). Finally, the JSP page could make use of custom tags or JavaBeans to help include the data in the HTML output (View Helper Pattern). Figure 5-1 illustrates the relationship between these patterns.

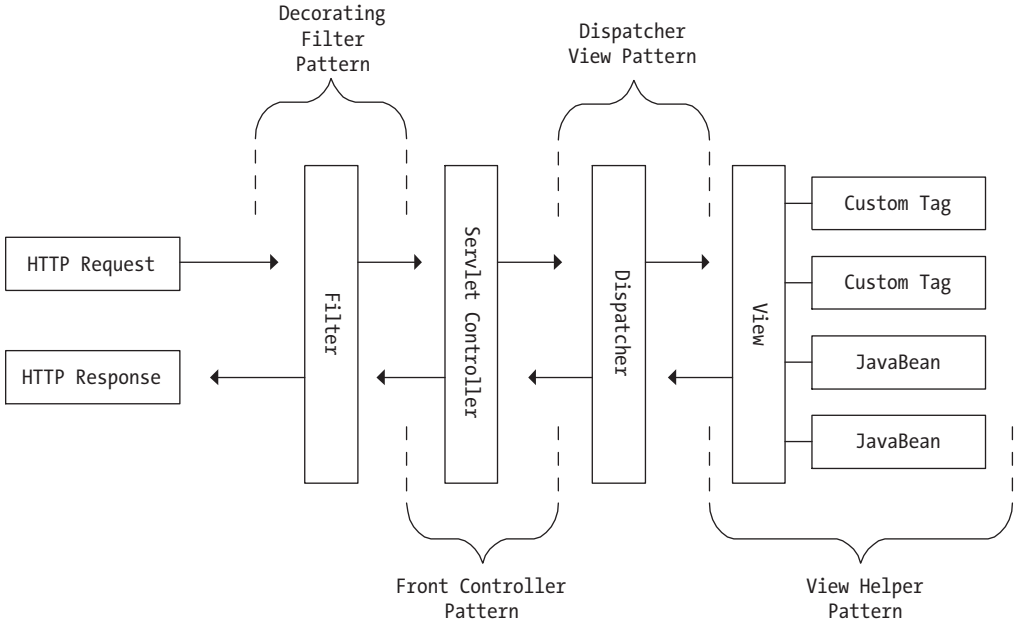

Figure 5-1. Presentation patterns

Here's a preview of each pattern I'll be discussing:

- **Decorating filter**: This pattern applies a kind of filter to either the request or response object as it passes in and out of the web container. You can use filters as a common place to log transactions, authenticate users, and even format data.

- **Front Controller pattern**: The Front Controller pattern is built upon the concept of the MVC pattern (see the next section). This pattern suggests the use of a single servlet to handle each and every request as opposed to embedding controller code inside of each JSP page.

- **Dispatcher view**: Inside of the controller, a piece of code exists that determines where the processed request should go to be displayed. In other words, it applies some kind of strategy to figure out which view, or JSP page, to use to display the current data.

- **View helper**: Once the specific view has been chosen, the JSP makes use of several "helpers" to adapt the data to the final outputted content. These helpers consist of either custom tags or JavaBeans.

Understanding Model-View-Controller (MVC)

The presentation patterns in the J2EE catalog are all based upon the MVC architecture. MVC is applied to software development projects in an effort to separate the application data from the presentation of the data. This separation enables the interface, or view, to take many different forms with little modification to the application code. For instance, using an MVC pattern, a user interface can be presented as both an HTML page (for web browsers) and a WML page (for mobile devices), depending on the device requesting the page. The controller would recognize the source of the request and apply the application data to the appropriate view (see Figure 5-2).

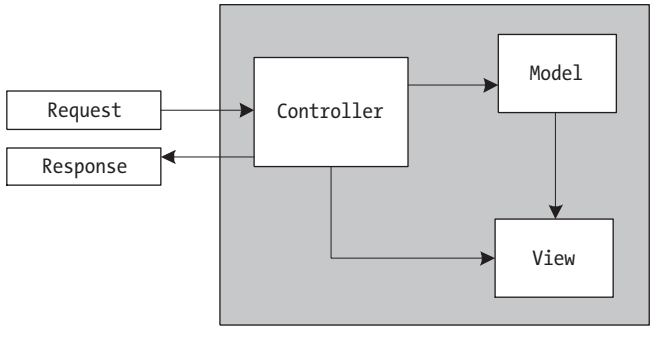

Figure 5-2. MVC architecture

The idea of separating presentation logic from the data and managing it with a controller has its roots in graphical user interface (GUI) development. Take, for instance, a user interface consisting of many different user controls. These controls contain the data, the formatting instructions, and the code that fires an event when the control is activated. This makes the user interface platform-specific and coupled with the application code itself. By applying an MVC pattern and separating each of these components, the user interface becomes lightweight, pluggable, and transportable across platforms. The Java Swing API illustrates this best.

You can apply the MVC pattern to other areas of software development besides client/server GUIs. Web development has benefited from this idea by clearly separating presentation code from the application data and the controller code that ties the two together. Let's take, for example, a simple web application that displays catalog pages. Typically, this would consist of a search page, a results page, and an item detail page. Each page has the responsibility of authenticating the user, retrieving user preferences, retrieving the requested data, and managing page navigation (see Figure 5-3).

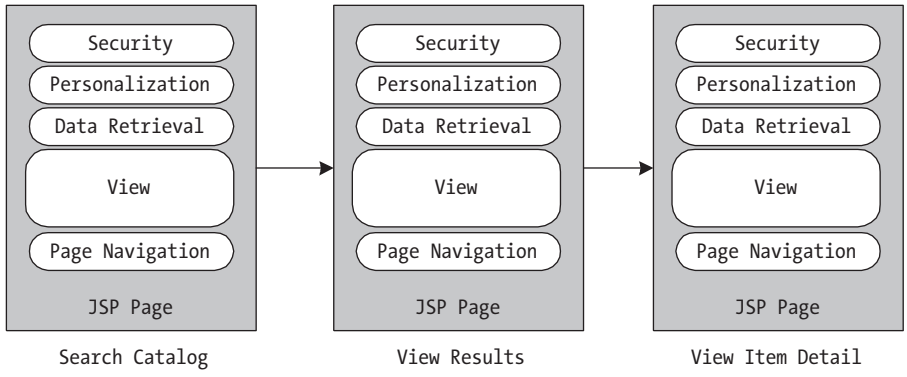

Figure 5-3. Simple catalog application (without MVC)

Looking at this application, it's easy to see that a lot of redundant code is being used to display each page. Not only does this introduce the potential for errors, but it also ties the application to the presentation by including many non-presentation functions inside of the presentation code. When you apply an MVC pattern to this application, the common functions move to a controller on the server. The presentation code is now only responsible for rendering the application data in a format that's appropriate for a particular device (typically a web browser). See Figure 5-4 for an MVC approach to this application.

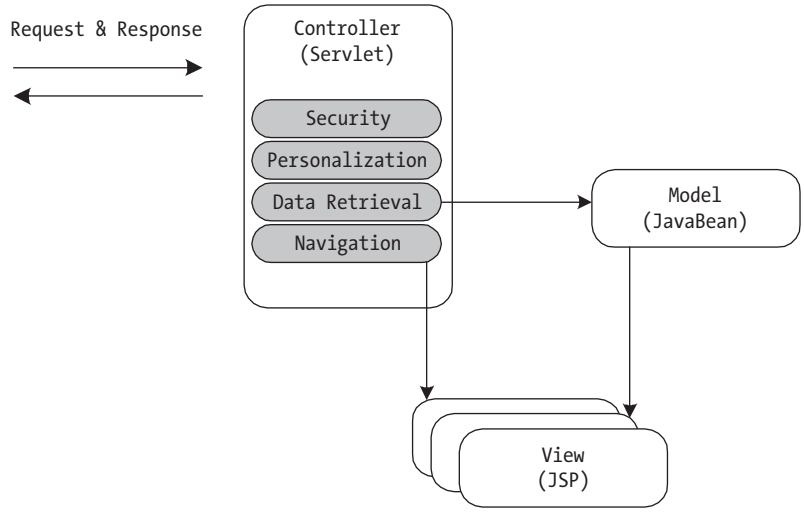

Figure 5-4. Simple catalog application (with MVC)

Seeing MVC in Action

To illustrate the MVC pattern, you're going to build a simple web application that collects data and stores it in a database. The data you'll be collecting is health information that will be stored in the customer table of our quoting database. In addition to collecting the data, the application will require the user to login to the system before accessing any of the pages.

This example is a simple one, but it illustrates some of the benefits of using an MVC architecture. You'll see how to centralize application security by giving the user a single access point into the application. You'll also standardize and share the database connection using a connection pooling mechanism built into the application server. In the next few chapters, I'll use this example (among others) to introduce new patterns. With that in mind, this example is basic at this point. You'll add features such as field validation and improved error handling later.

The application starts with a login page and then moves to a main servlet that will act as the controller (see Figure 5-5). The servlet will determine whether to proceed to the next page based upon the success of the login procedure. Once the user has logged in, they'll go to a survey page where they'll enter their information and submit it. Once again, the servlet will interrogate the request and move the user to the next page. If the data is successfully recorded, the user is taken to a confirmation page.

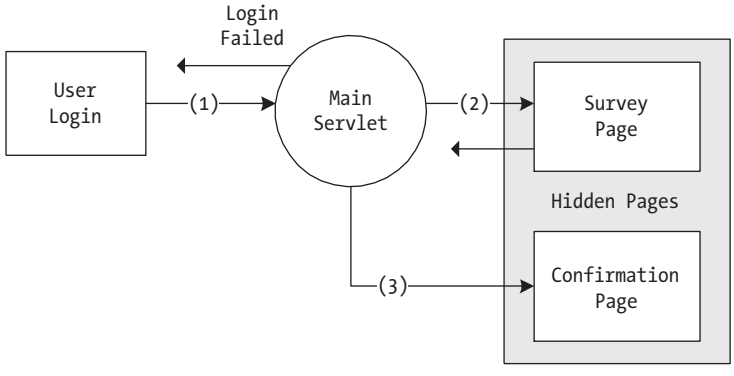

Figure 5-5. Simple survey application

Another benefit of using a servlet as a single entry point is that it enables you to hide your JSP pages from the outside world. This helps to secure the system by not allowing direct access to your application. The only page the user can access directly is the login page. If they were to type in the name of another page, the server would return a 404 error ("not found"). You accomplish this by "hiding"

your JSP pages inside of the \WEB-INF directory. By definition, everything underneath this directory is inaccessible by direct access from the user. However, the servlet that acts as our controller can access this directory and therefore is allowed to forward requests to pages that reside there. Here's what your directory structure will look like when you're done with this example:

```
webapps\jspBook\ch5\login.jsp
webapps\jspBook\ch5\myError.jsp
webapps\jspBook\ch5\myHeader.htm
webapps\jspBook\ch5\myFooter.htm
webapps\jspBook\ch5\images\logo.gif
webapps\jspBook\WEB-INF\jsp\ch5\census.jsp
webapps\jspBook\WEB-INF\jsp\ch5\thankyou.jsp
```

Setting Up the Application

Before you begin coding, you need to add a table to the database and then modify your application server configuration to accommodate the use of DataSources. The table you need to add is a user table containing the user ID, password, and a customer ID. The customer ID creates a customer record that corresponds to the user. Ideally, this field would be dynamically generated, but for these purposes you're just going to hard-code this value. Here's the script to update the database:

```
createUsers.sql (c:\mysql quoting < createUsers.sql)

DROP TABLE IF EXISTS user;
CREATE TABLE user (id varchar(10) not null, pwd varchar(10), cust_id int);
INSERT INTO user VALUES ('apatzer', 'apress', 6);
```

The next task you need to do is modify your configuration to use DataSources. The J2EE specification allows a DataSource to be defined inside of the application server itself. Servlets and JSP pages can locate and use these DataSources using Java Naming and Directory Interface (JNDI). A key advantage to accessing a database this way is that the connection information is stored in one place outside of the application code. Also, most application servers have a built-in connection pooling mechanism you can take advantage of by accessing your database using a DataSource. To set this up, you'll need to be sure your

application server supports this capability. Before modifying the appropriate configuration files, be sure to add your database drivers to a directory accessible to the application server (for Tomcat 4.0.1, put the drivers in the `\common\lib` directory). To create a DataSource in your application server, you'll need to add a description of it to the `server.xml` file. This description goes inside of your context definition like the one seen in Listing 5-1 (see the J2EE specification for more details).

Listing 5-1. `server.xml`

```
<Context path="/jspBook"
  docBase="jspBook"
  crossContext="false"
  debug="0"
  reloadable="true" >

  <Logger className="org.apache.catalina.logger.FileLogger"
             prefix="localhost_jspBook_log." suffix=".txt"
          timestamp="true"/>

  <Resource name="jdbc/QuotingDB" auth="SERVLET"
            type="javax.sql.DataSource"/>

    <ResourceParams name="jdbc/QuotingDB">
      <parameter>
        <name>driverClassName</name>
        <value>org.gjt.mm.mysql.Driver</value>
      </parameter>
      <parameter>
        <name>driverName</name>
        <value>jdbc:mysql://localhost:3306/quoting</value>
      </parameter>
    </ResourceParams>

</Context>
```

Now that you've described the `DataSource` to the application server, you need to tell your application about it. You do this by adding a resource entry into your `web.xml` file. Listing 5-2 shows what should go into this file (inside of your `<web-app>` tags).

Listing 5-2. web.xml

```
<resource-ref>
  <description>
    Resource reference to a factory for java.sql.Connection
    instances that may be used for talking to a particular
    database that is configured in the server.xml file.
  </description>
  <res-ref-name>
    jdbc/QuotingDB
  </res-ref-name>
  <res-type>
    javax.sql.DataSource
  </res-type>
  <res-auth>
    SERVLET
  </res-auth>
</resource-ref>
```

Finally, to use the DataSource, you need to replace any code that gets a database connection with the following piece of code (in this example's servlet you execute this code once inside of your init method):

```
try {
  Context initCtx = new InitialContext();
  Context envCtx = (Context) initCtx.lookup("java:comp/env");
  DataSource ds = (DataSource) envCtx.lookup("jdbc/QuotingDB");
  dbCon = ds.getConnection();
}
catch (javax.naming.NamingException e) {
  System.out.println("A problem occurred retrieving a DataSource object");
  System.out.println(e.toString());
}
catch (java.sql.SQLException e) {
  System.out.println("A problem occurred connecting to the database.");
  System.out.println(e.toString());
}
```

Defining the Model

Before walking through the controller or the views, you need to define the model. The model is responsible for storing data that will be displayed by one or more views. Typically, a model exists as an Enterprise JavaBean (EJB) or simply a

regular JavaBean. For this example, you'll just use a JavaBean. You might recall from Chapter 3 that you used a JavaBean to model the customer data. You'll reuse some of that and add a few additional methods to suit these purposes.

Aside from removing some unnecessary code, you'll need to add two new methods to the `CustomerBean` you built back in Chapter 3. The first method you'll add is the `populateFromParameters` method. This method takes an `HttpServletRequest` object as a parameter. The method is responsible for reading the input fields from the request object and populating the bean properties with their values. Also, the user ID is pulled out of the user's session and stored in the bean for later use. The other new method you'll be adding to this bean is the `submit` method. This method takes a `Connection` object as a parameter and is responsible for updating the database with the stored data residing in the properties (fields) of the bean. Listing 5-3 shows the updated code for the CustomerBean.

Listing 5-3. `CustomerBean.java`

```
package jspbook.ch5;

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

public class CustomerBean implements java.io.Serializable {

  /* Member Variables */
  private String lname, fname, sex;
  private int age, children;
  private boolean spouse, smoker;

  /* Helper Variables */;
  private String uid ;

  /* Constructor */
  public CustomerBean() {
    /* Initialize properties */
    setLname("");
    setFname("");
    setSex("");
    setAge(0);
    setChildren(0);
    setSpouse(false);
    setSmoker(false);
  }
```

```
public void populateFromParms(HttpServletRequest _req) {
  // Populate bean properties from request parameters
  setLname(_req.getParameter("lname"));
  setFname(_req.getParameter("fname"));
  setSex(_req.getParameter("sex"));
  setAge(Integer.parseInt(_req.getParameter("age")));
  setChildren(Integer.parseInt(_req.getParameter("children")));
  setSpouse((_req.getParameter("married").equals("Y")) ? true : false);
  setSmoker((_req.getParameter("smoker").equals("Y")) ? true : false);
  // Get session and populate uid
  HttpSession session = _req.getSession();
  uid = (String) session.getAttribute("uid");
}

/* Accessor Methods */

/* Last Name */
public void setLname(String _lname) {lname = _lname;}
public String getLname() {return lname;}

/* First Name */
public void setFname(String _fname) {fname = _fname;}
public String getFname() {return fname;}

/* Sex */
public void setSex(String _sex) {sex = _sex;}
public String getSex() {return sex;}

/* Age */
public void setAge(int _age) {age = _age;}
public int getAge() {return age;}

/* Number of Children */
public void setChildren(int _children) {children = _children;}
public int getChildren() {return children;}

/* Spouse ? */
public void setSpouse(boolean _spouse) {spouse = _spouse;}
public boolean getSpouse() {return spouse;}

/* Smoker ? */
public void setSmoker(boolean _smoker) {smoker = _smoker;}
public boolean getSmoker() {return smoker;}
```

```
public boolean submit(Connection _dbCon) {

  Statement s = null;
  ResultSet rs = null;
  String custId = "";
  StringBuffer sql = new StringBuffer(256);

  try {
    // Check if customer exists (use uid to get custID)
    s = _dbCon.createStatement();
    rs = s.executeQuery("select * from user where id = '" + uid + "'");
    if (rs.next()) {
      custId = rs.getString("cust_id");
    }

    rs = s.executeQuery("select * from customer where id = " + custId);
    if (rs.next()) {
      // Update record
      sql.append("UPDATE customer SET ");
      sql.append("lname='").append(lname).append("', ");
      sql.append("fname='").append(fname).append("', ");
      sql.append("age=").append(age).append(", ");
      sql.append("sex='").append(sex).append("', ");
      sql.append("married='").append((spouse) ? "Y" : "N").append("', ");
      sql.append("children=").append(children).append(", ");
      sql.append("smoker='").append((smoker) ? "Y" : "N").append("'");
      sql.append("where id='").append(custId).append("'");
    }
    else {
      // Insert record
      sql.append("INSERT INTO customer VALUES(");
      sql.append(custId).append(",'");
      sql.append(lname).append("', '");
      sql.append(fname).append("', ");
      sql.append(age).append(", '");
      sql.append(sex).append("', '");
      sql.append((spouse) ? "Y" : "N").append("', ");
      sql.append(children).append(", '");
      sql.append((smoker) ? "Y" : "N").append("')");
    }
    s.executeUpdate(sql.toString());
  }
```

```
    catch (SQLException e) {
      System.out.println("Error saving customer: "
                          + custId + " : " + e.toString());
      return false;
    }
    return true;
  }

}
```

Setting the View

The presentation logic of the application is stored in three JSP files. The first one, `login.jsp`, is accessible to the public, and the other two are only accessible from the controller servlet. The login page is a simple user and password entry screen that submits its data to the `Main` servlet. You'll notice that you add a parameter to the servlet called `action`. This tells the servlet what it needs to do. In this case, the action is `login`. If there's an error while attempting to log in, the servlet will add an attribute to the session indicating a problem and then return the user to the login page. Because of this, the login page checks the session for the appropriate attribute and displays corresponding error message if it finds it. Listing 5-4 shows the complete listing of the login page.

Listing 5-4. `login.jsp` *(*`\webapps\jspBook\ch5\login.jsp`*)*

```
<%@ page
      errorPage="myError.jsp?from=login.jsp"
%>

<html>
<head>
  <title>Quoting System Login</title>
</head>

<body bgcolor="#FFFF99">

<%@ include file="myHeader.html" %>

<form method="post" action="Main?action=login">
```

```
<p align="center">
  <font face="Arial, Helvetica, sans-serif" size="6" color="#003300">
    <b><i>Login to Quoting System</i></b>
  </font>
</p>

<p> </p>

<%  String loginError = (String) session.getAttribute("loginError");
    if (loginError != null && loginError.equals("y")) {
%>
<center>
  <font color="#ff0000">Invalid login, please try again.</font>
</center>
<%  }
%>

<table width="199" border="0" align="center" cellpadding="5">
  <tr>
    <td>
      <font face="Arial, Helvetica, sans-serif" size="2">User ID:</font>
    </td>
    <td><input type="text" name="UID"></td>
  </tr>
  <tr>
    <td><font face="Arial, Helvetica, sans-serif" size="2">Password:</font></td>
    <td><input type="password" name="PWD"></td>
  </tr>
  <tr align="center">
    <td colspan="2"><input type="submit" name="Submit" value="Login"></td>
  </tr>
</table>

</form>

<%@ include file="myFooter.html" %>

</body>
</html>
```

Figure 5-6 shows the login page.

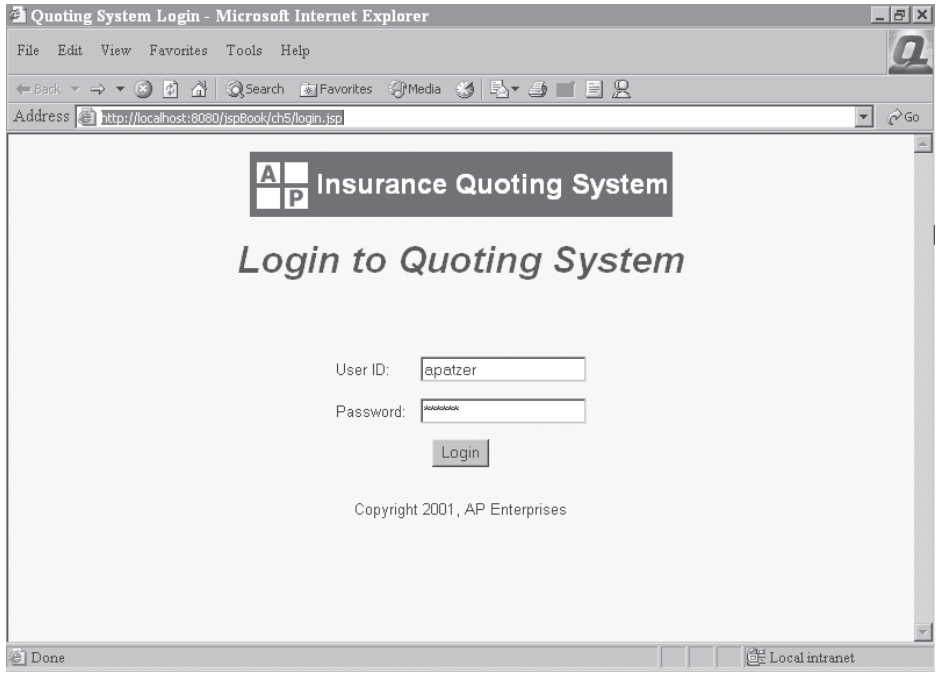

Figure 5-6. Login page

The survey page (`census.jsp`) collects data from the user and submits it to the Main servlet. The action parameter is set to `submit` to indicate to the servlet that you want to submit data to the database. This page is a good example of one that needs to be enhanced to include field validation and error handling. You'll do this in future chapters as you explore other presentation patterns. See Listing 5-5 for the complete code of the simple data collection page.

Listing 5-5. `census.jsp` *(*`\WEB-INF\jsp\ch5\census.jsp`*)*

```
<!-- JSP Directives -->
<%@ page
      errorPage="myError.jsp?from=census.jsp"
%>

<html>
<head>
  <title>Insurance Quoting System</title>
</head>

<body bgcolor="#FFFF99">

<basefont face="Arial">
```

```
<%@ include file="/ch5/myHeader.html" %>

<form action="Main?action=submit" method="post">

<br><br>

<%  String submitError = (String) session.getAttribute("submitError");
    if (submitError != null && submitError.equals("y")) {
%>
<center>
  <font color="#ff0000">Error recording survey data, please try again.</font>
</center>
<br><br>
<%  }
%>

<center><b>Enter personal information:</b></center>
<br><br>
<table cellspacing="2" cellpadding="2" border="0" align="center">
<tr>
    <td align="right">First Name:</td>
    <td><input type="Text" name="fname" size="10"></td>
</tr>
<tr>
    <td align="right">Last Name:</td>
    <td><input type="Text" name="lname" size="10"></td>
</tr>
<tr>
    <td align="right">Age:</td>
    <td><input type="Text" name="age" size="2"></td>
</tr>
<tr>
    <td align="right">Sex:</td>
    <td>
      <input type="radio" name="sex" value="M" checked>Male</input>
      <input type="radio" name="sex" value="F">Female</input>
    </td>
</tr>
<tr>
    <td align="right">Married:</td>
    <td><input type="Text" name="married" size="2"></td>
</tr>
<tr>
```

```
    <td align="right">Children:</td>
    <td><input type="Text" name="children" size="2"></td>
</tr>
<tr>
    <td align="right">Smoker:</td>
    <td><input type="Text" name="smoker" size="2"></td>
</tr>
<tr>
    <td colspan="2" align="center"><input type="Submit" value="Submit"></td>
</tr>
</table>

<br><br>

</form>
<%@ include file="/ch5/myFooter.html" %>

</body>
</html>
```

Figure 5-7 shows the survey page.

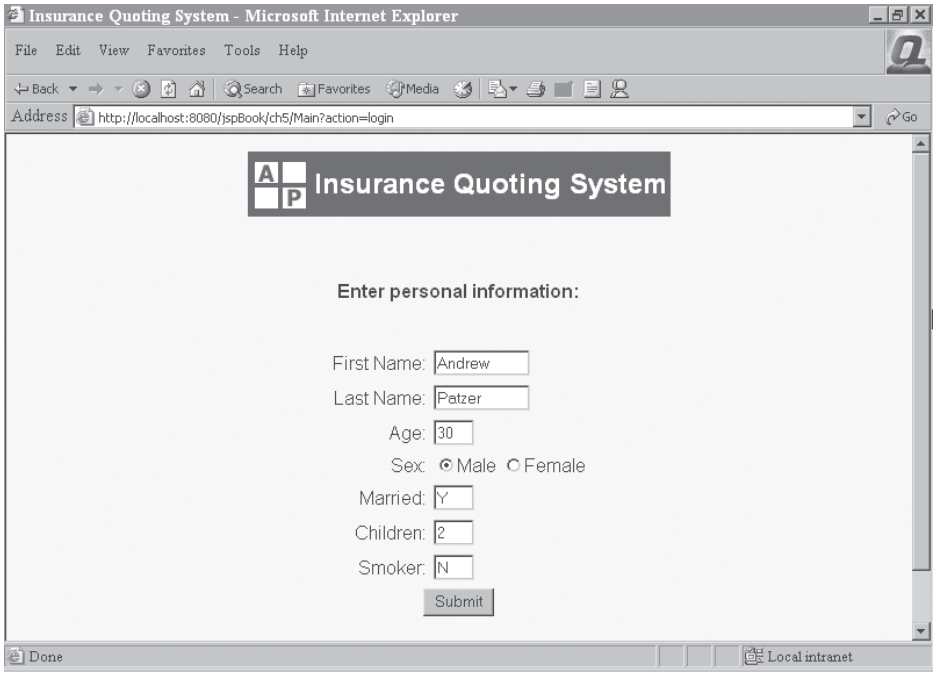

Figure 5-7. Survey page

Finally, once the data has been submitted, the request is forwarded to a confirmation page (`thankyou.jsp`). This is a simple page confirming that the data has been accepted. If there were an error trying to submit the data, control would return to the survey page (`census.jsp`) and an error message would appear at the top (similar to what you did with the login page). See Listing 5-6 for the confirmation page.

Listing 5-6. `thankyou.jsp` *(*`\WEB-INF\jsp\ch5\thankyou.jsp`*)*

```
<!-- JSP Directives -->
<%@ page
       errorPage="myError.jsp?from=thankyou.jsp"
%>

<html>
<head>
  <title>Insurance Quoting System</title>
</head>

<body bgcolor="#FFFF99">

<basefont face="Arial">

<%@ include file="/ch5/myHeader.html" %>

<br><br>

<center>
Your survey answers have been recorded.
Thank you for participating in this survey.
</center>

<br><br>

<%@ include file="/ch5/myFooter.html" %>

</body>
</html>
```

See Figure 5-8 for the confirmation page that's displayed upon successfully recording the survey data.

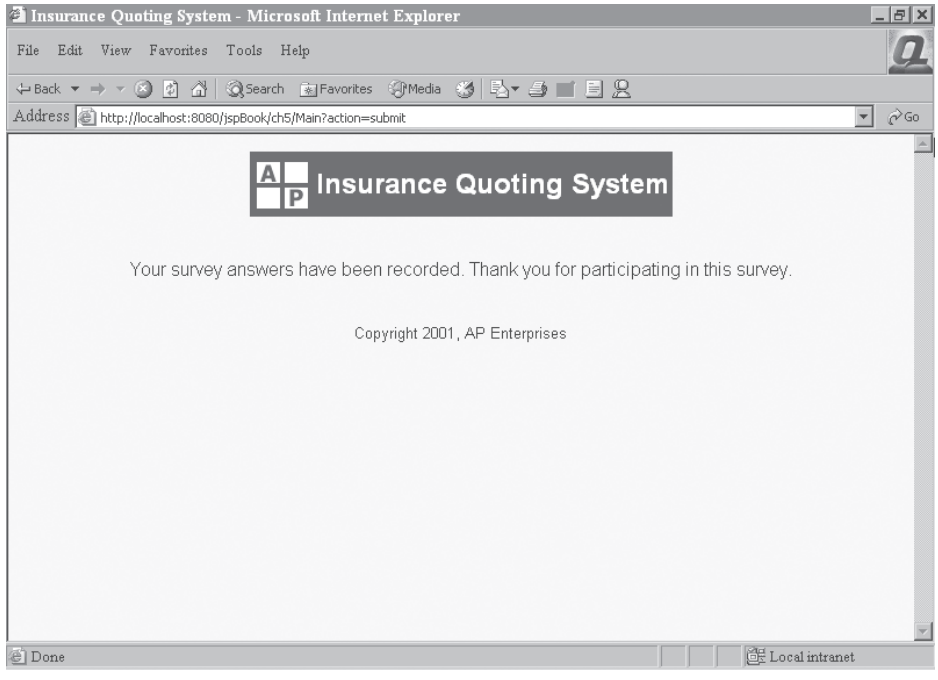

Figure 5-8. Confirmation page

Building the Controller

You'll be using a servlet as your controller (`Main`). To make this accessible, add the following entry to your `web.xml` file (inside of the `<web-app>` tags):

```
<servlet>
    <servlet-name>
        Main
    </servlet-name>
    <servlet-class>
        jspbook.ch5.Main
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>
        Main
    </servlet-name>
    <url-pattern>
        /ch5/Main
    </url-pattern>
</servlet-mapping>
```

The `init` method obtains your database connection using the `DataSource` you created earlier. This database connection is closed in the `destroy` method at the end of the servlet's lifecycle. Each request is serviced by the `doPost` method. Inside of there, the action is determined by checking the parameter `action`. The first time through, the `login` action directs the servlet to the `authenticate` method. If the login is successful, the user is taken to the `census.jsp` page.

The important thing to point out is that all security, database connectivity, and navigational control is centralized inside of this one servlet. You reuse code in several places. For instance, the navigational code goes into the `gotoPage` method. If you need to change this functionality, you only need to do it in one place. You'll see as you explore other patterns how useful this architecture really is. The goal of this example is simply to illustrate the basic idea of an MVC pattern. See Listing 5-7 for the controller servlet.

Listing 5-7. `Main.java`

```
package jspbook.ch5;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;
import javax.naming.*;
import javax.sql.*;

import jspbook.ch5.CustomerBean;

public class Main extends HttpServlet {


  DataSource ds;
  HttpSession session;

  /* Initialize servlet. Use JNDI to look up a DataSource */

  public void init() {

    try {
      Context initCtx = new InitialContext();
      Context envCtx = (Context) initCtx.lookup("java:comp/env");
      ds = (DataSource) envCtx.lookup("jdbc/QuotingDB");
    }
    catch (javax.naming.NamingException e) {
      System.out.println(
```

```
            "A problem occurred while retrieving a DataSource object");
        System.out.println(e.toString());
      }

    }

    public void doPost (HttpServletRequest _req, HttpServletResponse _res)
      throws ServletException, IOException {

      /* Refresh session attributes */
      session = _req.getSession();
      session.removeAttribute("loginError");
      session.removeAttribute("submitError");

      String action = _req.getParameter("action");

      /* Authenticate user if request comes from login page */
      if (action.equals("login")) {
        String uid = _req.getParameter("UID");
        String pwd = _req.getParameter("PWD");
        if (authenticate(uid, pwd)) {
          session.setAttribute("validUser", "y");
          session.setAttribute("loginError", "n");
          session.setAttribute("uid", uid);
          gotoPage("/WEB-INF/jsp/ch5/census.jsp", _req, _res);
        }
        /* If the user login fails, then return them to the login page to retry */
        else {
          loginError(_req, _res);
        }
      }

      /* Record the survey data if the request comes from the survey form */
      else if (action.equals("submit")) {
        /* Make sure the user has logged in before recording the data */
        String validUser = (String) session.getAttribute("validUser");
        if (validUser.equals("y")) {
          if (recordSurvey(_req)) {
            /* Reset validUser flag and forward to ThankYou page */
            session.removeAttribute("validUser");
            gotoPage("/WEB-INF/jsp/ch5/thankyou.jsp", _req, _res);
          }
```

```
      else {
        session.setAttribute("submitError", "y");
        gotoPage("/ch5/login.jsp", _req, _res);
      }
    }
    /* If the user did not login, then send them to the login page */
    else {
      loginError(_req, _res);
    }
  }

}

/* Send request to a different page */
private void gotoPage(String _page, HttpServletRequest _req,
  HttpServletResponse _res)
  throws IOException, ServletException {

  RequestDispatcher dispatcher = _req.getRequestDispatcher(_page);
  if (dispatcher != null)
     dispatcher.forward(_req, _res);

}

/* Set error attributes in session and return to Login page */
private void loginError(HttpServletRequest _req, HttpServletResponse _res)
  throws IOException, ServletException {

  session.setAttribute("validUser", "n");
  session.setAttribute("loginError", "y");
  gotoPage("/ch5/login.jsp", _req, _res);

}

/* Check if the user is valid */
private boolean authenticate(String _uid, String _pwd) {

  Connection dbCon = null;
  ResultSet rs = null;
  try {
    dbCon = ds.getConnection();
    Statement s = dbCon.createStatement();
    rs = s.executeQuery("select * from user where id = '"
```

```
            + _uid + "' and pwd = '" + _pwd + "'");
      return (rs.next());
    }
    catch (java.sql.SQLException e) {
      System.out.println("A problem occurred while accessing the database.");
      System.out.println(e.toString());
    }
    finally {
      try {
        dbCon.close();
      }
      catch (SQLException e) {
        System.out.println("A problem occurred while closing the database.");
        System.out.println(e.toString());
      }
    }

    return false;

  }

  /* Using the CustomerBean, record the data */
  public boolean recordSurvey(HttpServletRequest _req) {

    Connection dbCon = null;
    try {
      dbCon = ds.getConnection();
      CustomerBean cBean = new CustomerBean();
      cBean.populateFromParms(_req);
      return cBean.submit(dbCon);
    }
    catch (java.sql.SQLException e) {
      System.out.println("A problem occurred while accessing the database.");
      System.out.println(e.toString());
    }
    finally {
      try {
        dbCon.close();
      }
      catch (SQLException e) {
        System.out.println("A problem occurred while closing the database.");
        System.out.println(e.toString());
      }
    }
```

```
    return false;

  }


  public void destroy() {
  }

}
```

Summary

This chapter introduced you to the idea of using patterns to design your applications. Patterns are industry-wide best practices that have been tested and proven by many different developers. The *J2EE Patterns Catalog* contains several design patterns for enterprise Java development. This book covers four specific presentation-tier patterns that help to describe several best practices for JSP development.

Each of these patterns assumes an MVC architecture, which organizes your web application into three logical pieces. The model stores the application data, the view displays the application data, and the controller manages requests and handles navigation through the application. The next few chapters will explore J2EE patterns that extend each of these areas and applies strategies to maximize the efficiency of developing MVC-based web applications.

Index

C

D

E

F

G

H

I

L

M

S

T