JSP Examples and Best Practices

ANDREW PATZER

Apress™

JSP Examples and Best Practices

Copyright ©2002 by Andrew Patzer

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

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

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

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 information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

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.

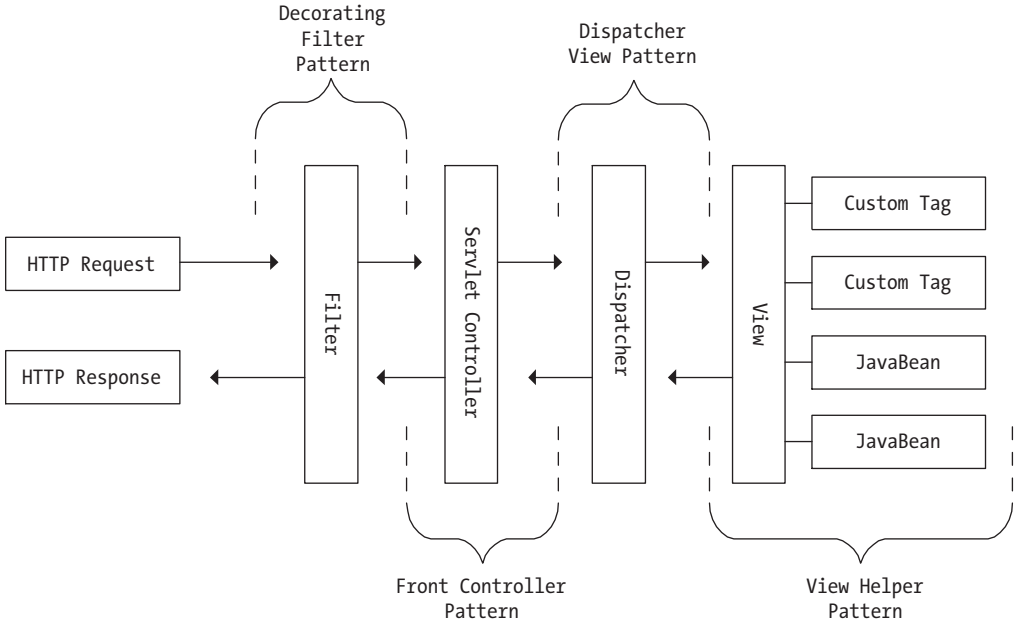

Figure 5-1. Presentation patterns

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).

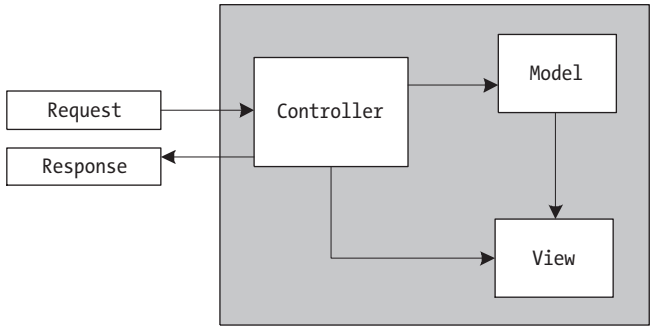

Figure 5-2. MVC architecture

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).

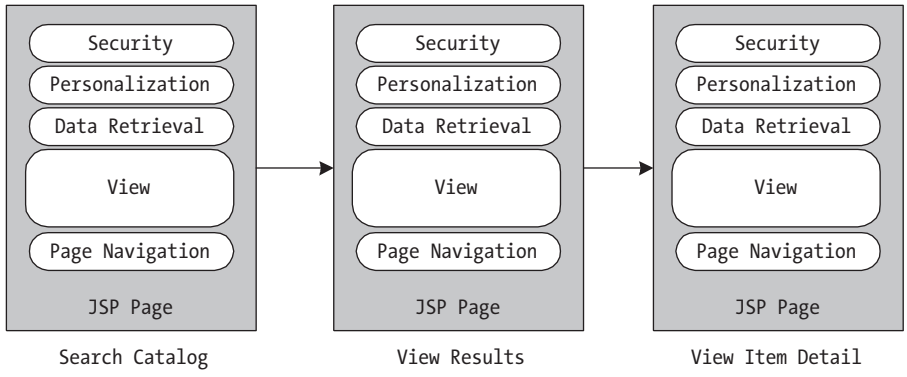

Figure 5-3. Simple catalog application (without MVC)

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.

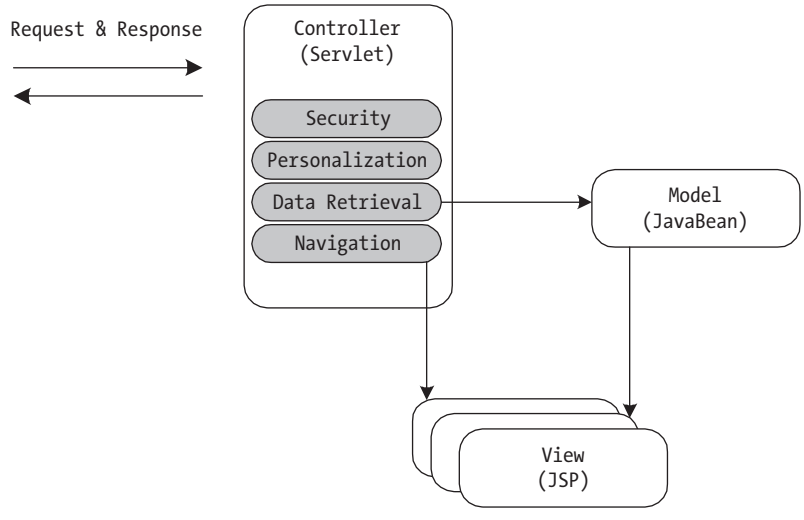

Figure 5-4. Simple catalog application (with MVC)

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.

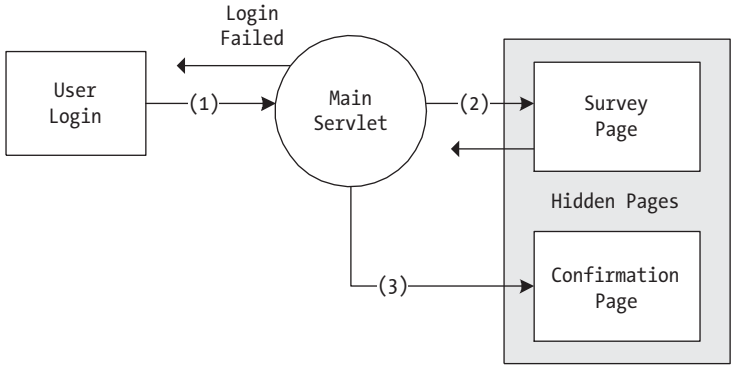

Figure 5-5. Simple survey application

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.

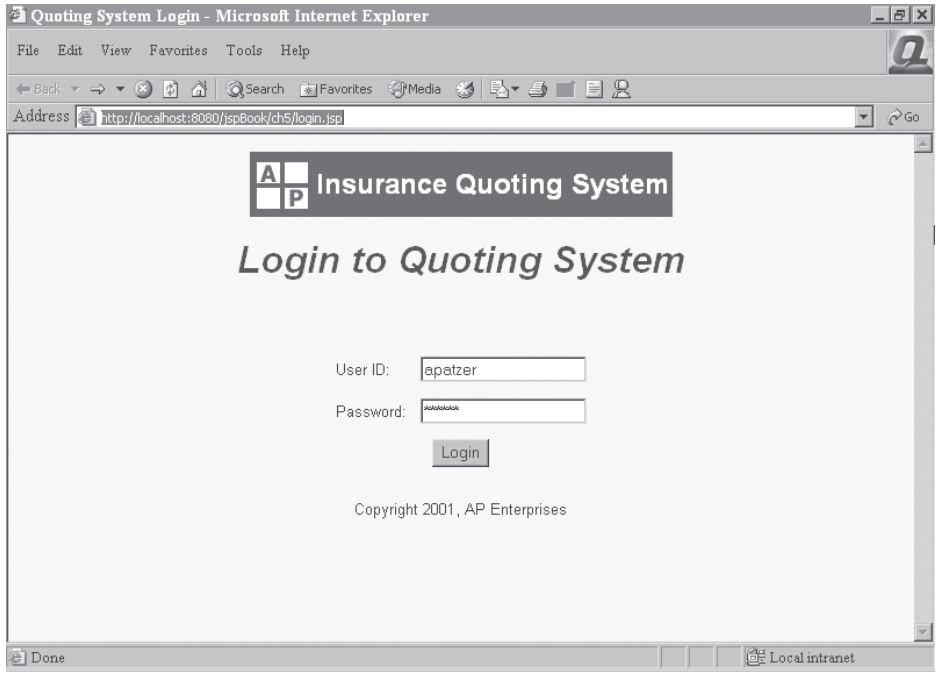

Figure 5-6. Login page

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.

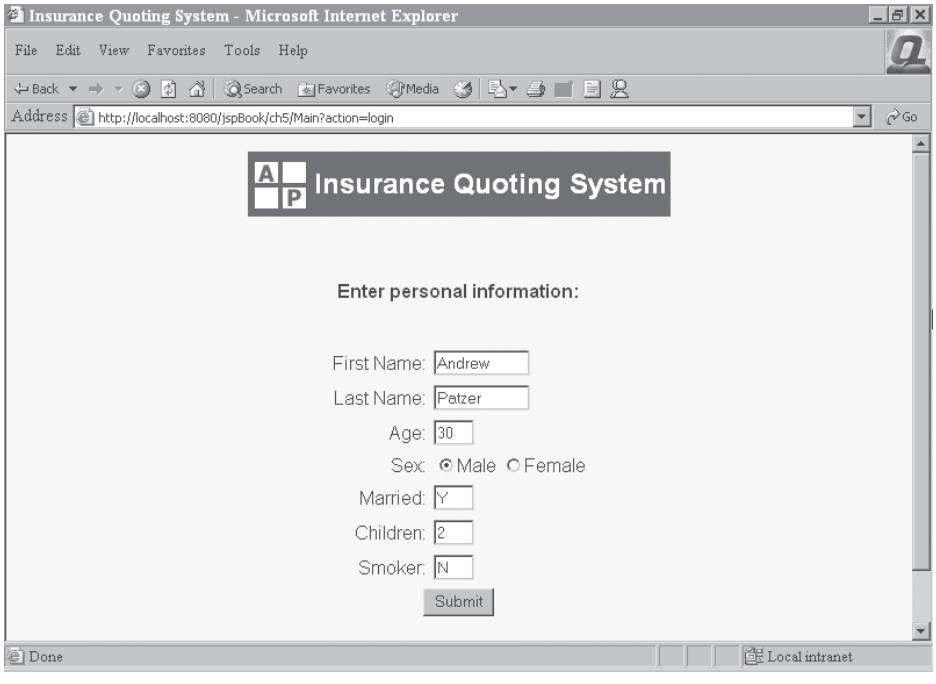

Figure 5-7. Survey page

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.

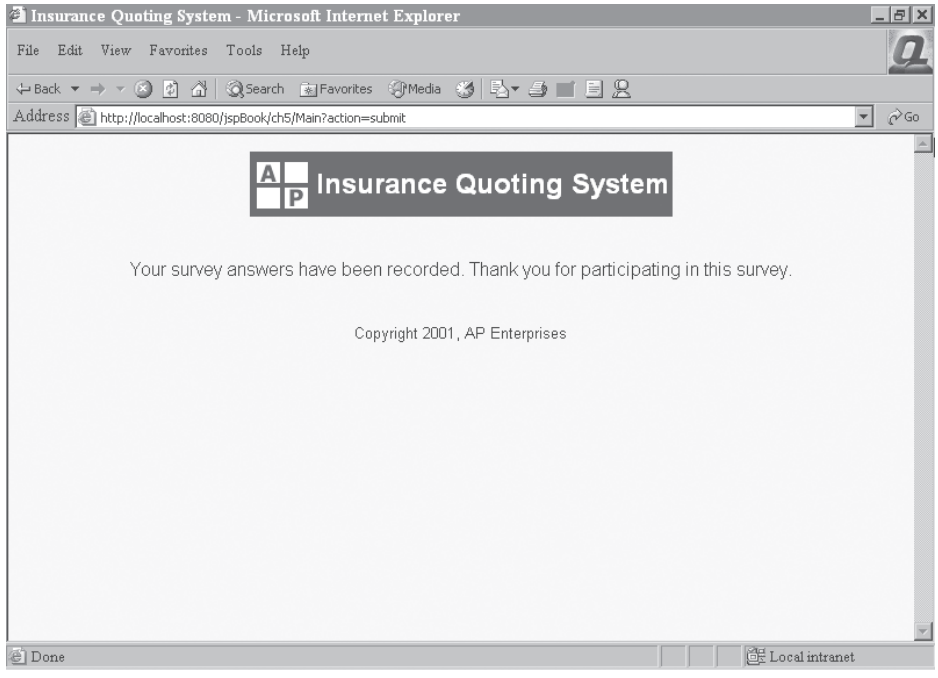

Figure 5-8. Confirmation page

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

Numbers and Symbols

404 error message, 111
<%! and %> tags, for declaring variables and methods, 13

A

Action interface
  creating, 156–157
  defining, 160
ActionFactory, code example, 157, 161, 250
Action.java, code example, 248–249
Ant tasks, using to account for different platforms when building scripts, 229
Ant tool
  advantages of using to build procedures, 229
  from the Apache-Jakarta Project, 215–216
  automating the build process with, 220–230
  built-in tasks for operations on the file system, 221
  creating a framework's build script with, 255–257
  integrating CVS with, 225–226
  for Java development, 219
Apache-Jakarta Project
  Ant build tool by, 215–216
  downloading Tomcat server from, 16
AppConstants.java, code example, 237–238
application deployment techniques, 215–230
application frameworks, 231–258
  adding unit tests to, 207–210
  designing, 231–232
application server
  choosing to host your JSP and servlet applications, 16–17
  popular commercial vendors for, 16
  setting up for online catalog, 263–264
  as web application component, 3–4
application-specific behaviors, implementing, 161–163
applications, testing for performance, 210–214
assertTrue method, code example, 207–210
AuthenticationFilter.java, code example for, 241–244
automated build procedure, 219
automated unit tests, 203

B

BEA WebLogic
  choosing as your application server, 16
  J2EE-compliant application server, 5
best practice, defined by a development pattern, 7
body tag, seeing one in action, 93–97
BodyContent object, obtaining a JspWriter through, 92
BodyTag, example of a custom JSP tag using, 91–92
BodyTag interface
  default implementation, 84
  lifecycle methods and return values for, 85
  use of, 84
bodyTagExample.jsp, code example, 96–97
BodyTagSupport, default implementation for BodyTag interface, 84
Bugzilla, website address, 219
build procedure, automated, 219
build process, automating with Ant, 220–230
build script
  creating a simple script with Ant, 220–223
  global variable you can use in, 221
build_cvs.xml file, for moving source code into a working directory, 226
build.xml file
  code example, 255–257
  for compiling all of the source code for this book, 222–223
  naming your project in, 220

build.xml file *(continued)*
script for building your online catalog application, 266–267
business tier, in *J2EE Patterns Catalog*, 107
ByteArrayPrintWriter class, enclosure of writer and stream within, 144

C

CartAction class, for accessing the online catalog shopping cart, 286–288
CartAction.java, code example, 286–288
CartBean class, code example, 289–292
CartBean.java, code example, 289–292
cart.jsp, code example for displaying the shopping cart, 292–294
catalog application, simple with and without MVC, 110
catalog database, creating and adding records, 53–54
CatalogBean class, for scrolling through and accessing catalog items, 274–279
CatalogBean.java, code example, 274–279
CatalogDB DataSource, code example for using, 238
CatalogItem.java, code example, 282–286
catalogtags.tld, code example, 281
census.jsp, code example, 120–122, 174–175
change management, using bug tracking facilities for, 219
checkout argument, using with the cvs command, 224–225
code example
ActionFactory, 157
ActionFactory.java, 161, 250
Action.java, 160, 248–249
for add method routine, 206
for adding a resource entry into your web.xml file, 113–114
for adding records to a catalog database, 54
AppConstants.java, 237–238
for AuthenticationFilter.java, 241–244
of base methods Action interface defines, 156–157
for basic concepts of input form, 38
for bodyTagExample.jsp, 96–97
for build script to compile and deploy a framework, 255–257
for building a URL, 207
for building the controller, 124
for building the ProductBean table, 63
build.xml file, 255–257
CartAction.java, 286–288
CartBean class, 289–292
cart.jsp page, 292–294
for casting request or response objects to HTTP equivalents, 135
for catalog item tag handler, 282–286
CatalogBean.java, 274–279
CatalogItem.java, 282–286
catalogtags.tld, 281
census.jsp, 172–174
for changing the edit link, 68–69
for connecting to the ProductBean database, 56
Controller.java, 169–171, 245–247
for converting Windows syntax to Unix-style syntax, 229–230
for creating a catalog database, 54
for creating a directory and copying files into it, 221
for creating, adding tables to, and exiting a database, 18
for creating database tables, 18
for creating navigational links, 63
of a custom JSP tag using a BodyTag, 91–92
for custom tag for formatting dates and numbers, 251–254
CustomerBean.java, 70–76, 165–168
for customerDetail.jsp, 77–80
customers.jsp file after adding page directive, 30–32
customers.jsp file with try-catch blocks, 23–25
for cvs command for a Windows and Unix system, 224
of cvs commands and tasks, 225
for DBHelper.java, 239–240
of declarations followed by XML syntax, 13
for declaring a JavaBean, 62
for declaring a ResultSet object, 21
for declaring a tag library descriptor, 179–180
for declaring and referencing a JavaBean, 179
for declaring the FileOutputStream at the class level, 140
for executing the doFilter method of the FilterChain, 141
for expressions, 14
FormatHelper.jsp, 187–189
FormatTag.java, 184–187, 251–254
FormattingModel.java, 183–184

for a forward tag, 41
for generating rows of customers, 22–23
for getting JavaBean properties, 51
for the GrocerOrder tag handler, 100–102
GroceryItem.java, 99–100
groceryList.jsp, 102–103
hello.jsp, 51–52
helpers.tld file, 182–183
for HomeAction class for online catalog system, 271–274
home.jsp page, 279–280
of how a controller servlet might look, 153–154
HtmlUtils.java, 94–95
of include directive, 13, 33
of including a file at runtime, 35
of including a file with parameters at runtime, 36
for the interface all action objects must implement, 248–249
for isErrorPage page directive, 28
Java filter object basic template, 136
<jsp:usebean> tag, 50
ListHelper.jsp, 197–199
for ListTag.java, 195–197
login action, 162–163
for login page, 118–119
login.jsp, 171–172, 270–271
Main.java, 125–129
MenuHelper.jsp page, 193–194
MenuModel.java, 190–191
MenuTag.java, 191–192
for a minimal request helper strategy, 155–156
for moving source code into a working directory, 226
myFooter.html, 34
myHeader.html, 34, 268
for obtaining a JspWriter through the BodyContent object, 92
for overriding jspInit() and jspDestroy() methods, 12
of page directive, 12–13
for populating databases, 18–19
for populating the ProductBean cache, 56–57, 62
for ProductBean.java, 58–61
for productList.jsp, 63–65
for putting your JSP environment all together, 23–25
for reading the content using a Reader, 92
for referencing a tag inside of a JSP page, 180
for removing an attribute from a session, 43
for request handling, 245–250
of request helper object, 159
RequestLoggingFilter class, 141–143
ReqUtility.java, 247–248
ResponseLoggingFilter.java, 145–147
to retrieve contents of a last_name input field, 37
for retrieving and storing customer table records, 21
for retrieving job field and hobbies list field, 39–40
for retrieving the name of an error page, 28
server.xml, 113
for sessionExample.html, 44
for sessionExample.jsp, 44–45
for sessionExamplePage1.jsp, 45
for sessionExamplePage2.jsp, 45
for setting a JavaBean property, 50–51
for a simple error page, 28–29
of a simple scriplet followed by its XML equivalent, 15
SimpleBean.Java, 49
simpleTagExample.jsp, 90–91
for SimpleTag.java, 87–88
simple.tld, 88–89
for skeleton code for a TestCase, 204–206
SubmitAction.java, 164–165
for <tag> entries, 98
for target, 221
for test code for add method routine, 206
thankyou.jsp, 123, 174–175
of a typical J2EE web application directory and WAR file, 227
for updating the survey application database, 112
for using a Factory pattern inside a request helper, 156
for using a request object inside of the doPost method, 155
for using getParameterNames method, 37
for using getParameterValues method, 37
using getSession method for obtaining a session, 43
using Java reflection to create a new action class instance, 249–250
for using the CatalogDB DataSource, 238
of using the cvs command with arguments, 224

code example *(continued)*
utils.tld file, 93–94
of a war task, 227
web.xml file, 89, 93
for writing an attribute to a session, 43
for writing the session attribute and displaying hyperlinks, 44–45
code reviews
example of code review form for, 218
importance of, 217–218
command and controller strategy, 156–158
commit argument, using with cvs command to commit changes, 224–225
confirmation page
code example, 123
displayed after recording of survey data, 124
controller, code example for building, 124
controller servlet
code example for, 125–129
code example of how it might look, 153–154
Controller.java
code example, 245–247
code example for building the controller, 169–171
conversion.jsp file, calling, 36
createCatalogDB.sql script, adding user and product tables to a database with, 261–262
createProducts.sql script, for adding records to a catalog database, 54
custom filter strategies, disadvantages of, 133
custom list formats, creating, 194–199
custom tag helper strategy, code example for declaring a tag library descriptor, 179–180
custom tags
looking at a simple example, 86–91
processing at runtime, 85
for reading a list of link items and outputting a list of hyperlinks, 190–191
role separation with, 83–103
using, 83–86
CustomerBean.java
code example, 70-76, 165–168
defining a model in, 114–118
customerDetail.jsp file, code example, 77–80
customerList.jsp file
changing the edit link in, 68–69
example code, 34–35
customers.jsp file
after adding page directive and removing try-catch blocks, 30–32
code example for, 23–25
modifying to include standard header and footer, 34–35
CVS
integrating with Ant, 225–226
website address for downloading, 223–224
cvs command
code example for a Windows and Unix system, 224
code example of using with arguments, 224
using checkout argument with, 224–225
using import argument with, 224
CVS repository
creating, 224–225
setting up for your code, 223–224
cvs task
code example, 225
using to interact with the CVS repository, 223–224
cvsRoot argument, specifying the location of the CVS repository with, 225

D

daemons, 3–4
database
creating for your online catalog, 260–261
running after creating, 19
selecting for your JSP environment, 17–19
database connection, establishing, 21
database helper, building, 237–240
database management (JDBC), defined by J2EE specification, 5
Database Management Systems (DBMS), selecting for your JSP environment, 17–19
database server, as web application component, 3–4
datacollection page, code example, 120–122
DataSources
code changes needed to use, 113–114
creating in your application server, 112–113

DBHelper.java, code example for, 239–240
debug messages, logging, 235–237
declarations, 13
declaring, filters, 137
Decorating Filter, presentation design pattern, 107–108
Decorating Filter pattern, 131–150
  applying, 134–150
  defining, 132
  strategies for applying, 133–134
deployment techniques, 215–230
Design Patterns (Gamma, Helm, Johnson, Vlissides), definitive guide to OO design patterns, 105
design patterns
  published best practices known as, 105
  reasons to use for development efforts, 105–106
development environment, choosing when setting up a JSP environment, 15–16
development framework, defined, 216–217
development patterns, for web applications, 7–9
development process, managing, 216–219
directives. *See* JSP directives
directory structure, creating in Tomcat, 17
Dispatcher View, presentation design pattern, 107–108
distributed object handling (RMI-IIOP), defined by J2EE specification, 5
documenting web application framework, 233–234
doFilter method
  code example for wrapping a response inside of, 144
  executing to continue processing remaining filters, 140–141

E

edit link, code example for changing, 68–69
EJB containers. *See also* Enterprise JavaBean (EJB) containers
  provided by a J2EE-compliant application server, 5
  understanding, 6–7
Enhydra, open-source J2EE-compliant application server, 5
Enterprise JavaBean (EJB) containers. *See* EJB containers
enterprise patterns, introduction to, 9
error and debug messages, logging, 235–237
error handling, for JSP pages, 27–32
error messages, logging, 235–237
error page
  creating, 27–29
  creating for your online catalog system, 268
errorPage page directive, code example, 29
evaluation copies, availability of for application servers, 16
exceptions, coding pages to forward all uncaught, 29–32
expressions
  defined, 14
  XML syntax for, 14

F

Factory pattern
  adding new behavior to your request-handling code with, 156–158
  code example for using inside a request helper, 156
  function of, 7
FileOutputStream, declaring at the class level, 140
filter chain, code that executes after return from processing, 144
filter class
  creating, 135–136
  implementing the javax.servlet.Filter interface to write, 140
filter manager, requests passed through, 132
filter strategies
  developing custom, 133
  filtering with J2EE, 135
  using standard, 134
filters
  declaring, 137
  entering form data to test, 147–149
  for logging HTTP parameters, 138
  mapping to a URL, 137
  potential uses for, 132
  for pre-processing of HTTP requests, 132
  using for integrated security, 131
  using to log HTTP requests, 137–143
  using to log HTTP responses, 143–150
  using with the Front Controller pattern, 175
form data, processing, 36–40

form handling
building the JavaBean for, 69–76
creating the solution, 67–68
implementing a solution, 68–80
patterns for, 36
standardizing, 66–82
steps required to process form data, 68
using the solution, 80–82
validating input, 80–82
FormatHelper.jsp, code example, 187–189
FormatTag.java, code example, 184–187, 251–254
formatting text, 182–183
FormattingModel.java, code example, 183–184
forward tag, code example for, 41
framework packages, for building a web application framework, 233
frameworks. *See also* application frameworks
creating build scripts for, 255–257
deploying, 255–258
designing, 231–232
using, 258
Front Controller, presentation design pattern, 107–108
Front Controller pattern
applying, 158–175
defining, 151–152
developing strategies, 152–158
using filters with, 175

G

GET request, function of, 2
getEnclosingWriter method, obtaining a JspWriter through the BodyContent object with, 92
getParameter() method, retrieving the name of an error page with, 28
getParameterNames method, code example for using, 37
getParameterValues method, returning a array of values with, 37
getReader method, getting content returned as a Reader with, 92
getSession method, code example for obtaining a session, 43
GroceryItem tags, contained in GroceryOrder tag, 98
GroceryItem.java, code example for, 99–100
groceryList.jsp, code example for, 102–103
GroceryOrder tag, GroceryItem tags contained in, 98
GroceryOrder tag handler, code example for, 100–102

H

hello.jsp, code example, 51–52
helpers, advantages of using, 178
helpers.tld file
code example, 182–183
full tag descriptor for, 193
HomeAction.java, code example, 272–274
home.jsp page, code example, 279–280
HTML form
for accepting a single field containing a person's name, 43–46
using to collect data, 36
HTML header, creating for your online catalog system, 268
HtmlUtils tag handler
writing, 94–95
writing the JSP for, 96–97
HtmlUtils.java, code example, 94–95
HTTP (HyperText Transfer Protocol), understanding, 2–3
HTTP requests, using filters to log, 137–143
HTTP responses
manipulating content of, 143
using filters to log, 143–150
HTTP sessions, using to manage user data, 42–46
HTTP sniffer, examining the contents and headers of a response object with, 137–138
HTTP tracer program, log file example, 3
HttpServletResponseWrapper, using, 144

I

IBM WebSphere, J2EE-compliant application server, 5
IllegalStateException, reasons thrown, 43
include directive, code example, 13, 33
input form
code example of basic concepts, 38
example of, 39
insurance quoting system, building a simple, 19–25
integration tier, in *J2EE Patterns Catalog*, 107

Intercepting Filter pattern, defining, 132
isErrorPage page directive, code example, 28

J
J2EE architecture, 5
J2EE compliant, definition of, 5
J2EE-compliant application, structure of, 7
J2EE-compliant application servers, popular, 5
J2EE Patterns Catalog (Sun)
  for detailed pattern definitions, 131
  introduction to, 106–107
  website address for, 105
J2EE specification services, 5
J2EE web applications, developing, 5–7
<jar> tag, running the Jar utility with, 222
Java code, using Macromedia HomeSite to insert, 15
Java Developer Connection website, *J2EE Patterns Catalog* available at, 106–107
Java filter object, basic template of, 136
Java Integrated Development Environment (IDE), choosing for JavaBeans, Servlets, and EJBs, 16
Java Servlets, function of, 5–6
JavaBean, code example for declaring and referencing, 179
JavaBean properties
  code example for getting, 51
  code example for setting, 50–51
JavaBeans
  accessing properties, 50–52
  building a simple, 48–49
  code example for declaring, 62
  conversion of datatypes within, 51
  creating for caching data, 53–54
  creating one containing a hashtable of link items, 190–191
  handling large sets of data within a single page, 53–66
  introduction to, 47–52
  and JSP, 48
  role separation with, 47–82
  using in a JSP page, 50
<javac> tag, running the Java compiler with, 222
Javadoc comments, formatting of, 233–234
Javadoc syntax, code example for descriptive comments, 233–234
java.net.URL object, using to connect to the URL, 207
JavaServer Pages (JSP). *See* JSP (JavaServer Pages)
javax.servlet.Filter interface, methods defined by, 135
JBoss
  for hosting EJBs, 16
  open-source J2EE-compliant application server, 5
JDBC. *See* database management (JDBC)
JDBC driver, obtaining to access your database, 19
jEdit, website address, 16
JMeter load testing tool
  from the Apache Group, 210–214
  viewing the graph results, 213–214
  website address for downloading, 210
JMeter start screen, 211
JMS, JavaMail. *See* messaging (JMS, JavaMail)
JNDI. *See* naming services (JNDI)
JSP (JavaServer Pages)
  foundations, 1–25
  function of, 6
  handling errors, 27–32
  and JavaBeans, 48
  learning the basics of, 10–15
  processing, 10–12
  processing steps, 11
  structure of, 12–15
  using, 27–46
JSP (JavaServer Pages) applications
  building simple, 19–25
  choosing an application server for, 16–17
JSP directives, function of, 12–13
JSP environment
  choosing a development environment, 15–16
  puttting it all together, 23–25
  setting up, 15–19
<jsp:forward> tag vs. <jsp:include> tag, 41
JSP front vs. servlet front strategy, 153–155
JSP Model 1, moving to, 8
JSP Model 2, moving to, 9
JSP pages
  building to implement the form handler, 76–80
  code example for FormatHelper.jsp, 187–189
  creating for your online catalog, 259–260

JSP pages *(continued)*
designing, 20
establishing a database connection, 21
including a file at compile-time vs. runtime, 32
including files at compile-time, 33–35
including files at runtime, 35–36
modifying for your controller, 171–175
precompiling, 228–229
refreshing the model, 57–61
session ID created on the server at first request for, 42
tools for development, 15–16
using a JavaBean in, 50
using your tag library in, 90–91
writing for the HtmlUtils tag handler, 96–97
JSP Quick Reference card, website address for, 25
JSP tags
example using a BodyTag, 91–92
using Macromedia HomeSite to insert, 15
jspc tool
provided by Tomcat server, 228–229
running to convert a JSP file, 228
jspDestroy() method, overriding, 11–12
<jsp:getProperty> tag, getting a JavaBean property with, 51
<jsp:include> tag vs. <jsp:forward> tag, 41
jspInit() method
loading of, 11
overriding, 11–12
_jspService() method, function of, 11
<jsp:setProperty> tag, setting JavaBean properties with, 50–51
<jsp:usebean> tag, code example, 50
JTA. *See* transaction management (JTA)
JUnit architecture, 203–204
JUnit package
using, 203–206
website address for, 203

L

list helper example, showing formatted lists, 199
listeners. *See* daemons
ListHelper.jsp, code example, 197–199
ListTag.java, code example for, 195–197
load testing, applications for performance, 210–214
log file
for response filter, 149–150
for request filter, 148–149
log4j package, from Apache Group, 235–237
Logger.java file, wrapping of the log4j functionality in, 235–237
logging, error and debug messages, 235–237
Logical Resource Mapping strategy, function of, 158
login page
code example, 118–119
creating for your online catalog system, 269–271
LoginAction.java, code example, 162–163
login.jsp, code example, 118–119, 171–172, 270–271

M

Macromedia Dreamweaver, using to develop a JSP page, 15–16
Macromedia HomeSite, code editor, 15
Main.java, code example, 125–129
manual unit test, use of in regression testing, 202–203
mapping, filters to a URL, 137
Menu helper example, 193
MenuHelper.jsp page, code example, 193–194
MenuModel.java, code example, 190–191
menus, creating, 190–194
MenuTag.java, code example, 191–192
messaging (JMS, JavaMail), defined by J2EE specification, 5
MIME type, 3
model, defining for your survey application, 114–118
model separation strategy, implementing, 181–182
Model-View-Controller (MVC) architecture, 109
role of JavaBeans in, 48
understanding, 109–129
Model-View-Controller (MVC) pattern, implementation of by JSP model 2, 9
Multiplexed Resource Mapping strategy, function of, 158
MVC pattern. *See also* Model-View-Controller (MVC) architecture
revisiting, 158–175
seeing it in action, 111–112
myError.jsp, code example, 28–29, 268
myFooter.html, code example, 34
myHeader.html, code example, 34, 268

MySQL
creating a database and adding some tables to, 18
downloading, 17
MySQL website, downloading the JDBC driver from, 19

N
naming services (JNDI), defined by J2EE specification, 5
navigational links, code example for creating, 63
network resources, limiting access to, 4

O
online catalog system
creating a simple catalog system with learned techniques, 259–295
illustration of, 260
online catalog with shopping cart
accessing the shopping cart, 286–288
adding the user and product tables to, 261–262
changing the startup script for, 265
code for describing CatalogItem tag, 281
creating application resources for, 267–268
creating login page for, 269–271
creating the database for, 260–261
designing the application, 259–260
directory structure for, 266
displaying the shopping cart, 292–294
HomeAction.java code for, 272–274
home.jsp page, 279–280
installing and configuring the framework, 264–265
logging in to the application, 269–271
modifying the startup script for, 265
setting up the application, 260–268
setting up the application server, 263–264
setting up the development environment, 266–267
viewing the home page for, 271–274
OO design patterns, definitive guide to, 105
outlineTag, code example for accessing, 97

P
page directive
code example using XML syntax, 13
for importing java.sql package for database code, 21
page navigation, controlling with JSP, 40–41
parent tag, obtaining an instance of, 98
patterns. *See* development patterns; web application development; web development patterns
development using, 105–129
Front Controller, 151–176
reasons to use for development efforts, 105–106
Physical Resource Mapping strategy, function of, 158
platform differences, accounting for when building scripts, 229–230
populate method, using to populate the ProductBean cache, 62
ports, specifying in a URL, 4
POST request, function of, 2
presentation design patterns, looking at, 107–108
presentation layer, using helpers when developing, 178
presentation patterns, relationships between, 108
presentation tier, in *J2EE Patterns Catalog*, 107
ProductBean
code example for building the table, 63
code example for connecting to the database, 56
creating, 54–55
declaring the class and implementing the Serializable interface, 55
declaring the fields in, 55
initializing properties for, 55
providing accessor methods for fields, 55
ProductBean class, using to page through a set of records, 65–66
ProductBean.java, code example, 58–61
productList.jsp, code example, 63–65
prop.file.dir system property, modifying the Tomcat server startup script to specify, 238–239
<property> tag, for defining properties in a build script, 221

R
regression testing
breaking into units, 202–203
understanding, 202

removeAttribute method, for removing an attribute from a session, 43
repeatable process, for software development, 215
request filter, log file for, 148–149
request handling, simplified, 244–250
request-handling framework
  defining a standard path for the request to follow, 152–153
  using Front Controller pattern to build, 151–176
request helper object, building a simple, 159
request helper strategy, 155–156
request object
  using inside of the doPost method, 155
  using methods on, 37
RequestLoggingFilter, web.xml file that describes and maps, 138–140
RequestLoggingFilter class, code example, 141–143
ReqUtility.java, code example, 247–248
ReqUtil.java, code example, 159
resource mapping strategies, functions of, 157–158
ResponseLoggingFilter.java, code example, 145–147
ResultSet object, declaring in a JSP declaration tag, 21
RMI-IIOP. *See* distributed object handling (RMI-IIOP)
role separation
  with custom tags, 83–103
  with JavaBeans, 47–82
rows of customers, generating, 22–23

S

scriplets, use of in JSP, 14–15
scripting, a build procedure, 219
server-side programs, processing data with, 36
servers, chaining of through port mapping, 4
server.xml, code example, 113
server.xml file, modifying for your online catalog, 263–264
servlet applications, choosing an application server for, 16–17
servlet controller, accessing, 154–155
servlet front vs. JSP front strategy, 153–155
servlet model, introduction to, 8
session ID, created on the server at first request for a JSP page, 42
sessionExample.html, code example for, 44
sessionExample.jsp, code example for, 44–45
sessionExamplePage1.jsp, code example for, 45
sessionExamplePage2.jsp, code example for, 45
setAttribute method, code example for writing an attribute to a session, 43
SimpleBean class
  code example illustrating use of, 51–52
  using in a JSP page, 52
SimpleBean.Java, code example, 49
simpleForm.html, code example, 38
simpleForm.jsp, code example, 39–40
simpleTagExample.jsp, code example, 90
SimpleTag.java, code example, 87–88
simple.tld, code example, 88–89
sniffer. *See* HTTP sniffer
software development process, importance of source code control in, 217
source code control
  importance of in software development process, 217
  integrating with Ant, 223–226
standard filter strategies, advantages of, 134
static model, containing a hashtable of link items, 190–191
stub, inserting wherever dynamic data is required, 20
submit action, code example, 164–165
survey application
  setting up, 112–114
  simple using MVC architecture, 111–112
survey page (census.jsp), data collected by, 120–122

T

tag descriptor
  for creating custom list formats, 194–195
  for helpers.tld file, 193
<tag> entries, code example for, 98
tag handler, locating in your JSP, 84

tag handler class
definition, 87–88
implementing, 86–88
Tag interface
lifecycle methods and return values for, 85
use of, 84
tag library
declaring, 89
using in a JSP page, 90–91
tag library descriptor file
creating, 88–89, 93–94
modifying, 93
taglib directive
declaring a tag with, 180
function of, 13
locating the tag handler in your JSP with, 84
tags, nesting, 97–103
target, code example, 221
target tag, for defining a set of actions to be performed, 220–221
TCP/IP, used by HTTP, 2
template text, defined, 12
TestCase, skeleton code for, 204–206
testing, importance of, 201–203
testing techniques, 201–214
thankyou.jsp, code example, 123, 174–175
thread group
adding a test to, 212
adding to the test plan, 212
ThreadGroup node, adding a web test to, 212–213
Tomcat
setting up, 16–17
using as a web container, 16–17
Tomcat server, jspc tool provided by, 228–229
toString() method, for converting an object value to a string, 29
transaction management (JTA), defined by J2EE specification, 5
<tstamp> tag, creating a timestamp with, 222

U

unit tests
adding to your application framework, 207–210
building a framework for, 203–210
use of in regression testing, 202–203
URL, code example for building, 207
user authentication, code example for, 241–244
users, authenticating, 241–244
utils.tld file, code example, 93–94

V

variable declarations, commenting, 234
View Helper
for formatting text, 182–189
presentation design pattern, 107–108
View Helper pattern
applying to your application, 182–199
defining, 177–178
implementing strategies for, 179–182

W

WAR files, building, 227
war task
code example, 227
website address for information about building WAR files, 227
web application development, patterns for, 7–9
web application framework
building, 232–254
building a database helper in, 237–240
commenting variable declarations, 234
designing, 232–233
documenting, 233–234
logging error and debug messages in, 235–237
web applications
architecture of, 4
components of, 3–4
developing, 1–4
web containers
provided by a J2EE-compliant application server, 5
understanding, 5–6
web development patterns, system level best practice provided by, 7–8
web server, as web application component, 3–4
\webapps directory, creating a directory structure underneath, 17
website address
for Bugzilla bug tracking program, 219
for downloading CVS, 223–224
for downloading MySQL DBMS, 17
for downloading the JDBC driver, 19
for downloading the Tomcat servlet container, 16

website address *(continued)*
for information on using Javadoc to document your code, 234
for a *J2EE Patterns Catalog* (Sun), 105
for Java Servlet specification, 7
for jEdit development tool, 16
for Sun JSP Quick Reference card, 25
web.xml file
adding a resource definition to, 263–264
code example, 89, 93
code example for adding a resource entry into, 113–114
code example for displaying GroceryOrder tag items in a table, 98
configuration information contained in, 7
for declaring the tag library, 89
modifying, 93