

Mastering Jakarta Struts: MVC with Struts

Generally speaking, developers create ugly HTML, and Web designers are allergic to Java. Sure, there are exceptions to these rules, but how do you get these two groups of people working together on the same project? Struts--with its set of custom tags and emphasis on getting scriptlets out of JSP--allows both sides to live in harmony.

Struts is an open source, Model-View-Controller (MVC) framework developed by The Apache Software Foundation as part of its Jakarta project. Struts is built on top of JSP, servlets, and tag libraries. While Sun was printing up the J2EE Blueprints, Apache was making them come to life. After reading the first J2EE Blueprints from Sun with their explanation of MVC and how to accomplish it with custom tags, servlets, and JSP, one can clearly see that Struts is the incarnation of Sun's J2EE MVC vision.

In many ways, Struts has led to the further development of J2EE with regard to the J2EE Web presentation tier. It's clear that Struts and Apache's TagLib project have influenced JavaServer Pages Standard Tag Library (JSTL) and JavaServer Faces. Struts pushed the MVC vision to its limits and discovered issues not covered by JSP and servlets. JSTL and JavaServer Faces were created to address the issues Struts discovered by pushing the limits of JSP/servlet based MVC. Struts is another example of the powerhouse Jakarta projects like Tomcat, Ant, and many others.

Struts is a growing, evolving project. It would be impossible to cover all of Struts in one chapter, so this chapter focuses on giving you a solid foundation and practical code examples for using Struts. If you need more information, we suggest James Goodwill's *Mastering Jakarta Struts*, also published by John Wiley & Sons.

Where to Get Struts

Struts 1.1 was developed by the Apache Software Foundation as part of its Jakarta project. Struts is distributed with the Apache Software License version 1.1, and you can download it at <http://jakarta.apache.org/ant/index.html>.

This chapter starts out with a brief description of Struts. Then, we present a common Model 1-style JSP application. Throughout most of this chapter, we reshape the Model 1 into a Model 2 and an MVC application, using and explaining Struts along the way.

Note the code for the Model 1 and the MVC application is focused on showing the concepts; it is not robust code. For example, the exception handling is kept short to make it easier to follow. To get the most out of this chapter, you should have a background with JavaBeans, JSP, and Java Database Connectivity (JDBC). You can utilize your knowledge of JSP to learn Struts in a Rosetta Stone fashion, as we compare the Model 1 versus the MVC way of writing the sample application.

Overview of Struts

Applications written using the Struts framework provide JSP pages for the View and custom actions and configuration files for the Controller. They endeavor to separate the Model so that the Model is not aware that it is being used for a Web application. In other words, the Model code is divorced from the presentation code. A typical Struts application has JSPs with very few and very relatively small JSP scriptlets.

Many control features are provided by the Struts framework and are highly configurable. The Struts Controller servlet dispatches and coordinates interaction between Actions and JSPs. Actions are subclasses of `org.apache.struts.action.Action`. The Struts Controller uses a configuration file that specifies ActionMappings. An ActionMapping maps path information to Actions, and specifies interaction between the Action and other resources and properties (such as forwarding information). Typically, an Action (as part of the Controller) interacts with the Model and then forwards to a JSP to display and get further user input. The Action will typically map domain objects (JavaBeans) into request, session, or application scope, and the JSP uses these objects to display information and collect user input.

Common areas of concern, such as form validation, are handled by the framework. Applications use ActionForms to validate data and populate HTML forms. ActionForms are a special type of JavaBean that subclasses `org.apache.struts.action.ActionForm`. Struts automatically stores the data for HTML form data in these strongly typed form beans. You can even nest form properties by having composite beans. Actions are mapped to ActionForms. Actions receive the results of form submissions via ActionForms that have been validated by the framework. Action can use to ActionForms to populate HTML forms with data. The life cycle of ActionForms is managed by the Struts framework, and ActionForms are notified when to do form validation.

The Struts framework comes with a set of custom tags that manage the interaction between the View and the Controller. One custom tag manages the ActionForm population from the HTML forms.

Just as you can write non-object-oriented code using Java if you try hard enough, you can write non-MVC code with Struts. Generally speaking, Struts should be used only for managing the View and the Controller. The Actions are like a glue between the presentation logic and your business logic and domain objects--that is, your Model. Actions should typically invoke your API set, which may consist of JavaBeans or Enterprise JavaBeans (EJBs). The JavaBeans that the Action uses should not refer to classes under the `javax.servlet` or the `org.apache.struts` package. For example, you would not expect to see JDBC calls in an Action, but the Action may call a bean that in turns calls JDBC (refer to Chapter 10 for more details). You may have additional mediator classes between your true Model facade and your Actions.

Now I must confess that I am not a purist. I believe in the simplest possible solution that will work. There are times when you must use MVC, and there are times when you can avoid it. If your project is not a prototype or a one-off solution, then you should use MVC. If you need to get Web designers and developers working together, then you should use MVC. Struts is a good vehicle for getting things done fairly quickly and still using MVC. We've worked on large projects using Struts, and we've found Struts to be very useful, as well as a good vehicle for learning a lot about JSP and custom tags.

A Typical Model 1-Style JSP Application

JSP was originally Sun's answer to the success of Microsoft's Active Server Pages (ASP). Much of the syntax and style is the same as what you'd find in ASP. In fact, this was intentional; tools suited for Web designers could already recognize the special ASP tags, and JSP copied these so it could also work with these tools. Along with ASP's success, JSP also inherited ASP's biggest "flaw": It allows, in fact almost encourages, developers to write sloppy code. JSP scriptlets are quick and dirty, but always rough and ready.

You can still get a lot of mileage out of Model 1-style development, but it can be a hindrance. For example, there is a lot of mixing of Java code and HTML, which makes the HTML harder for Web designers to change, not to mention harder to maintain as the logic gets spread over many pages.

In this section, we present a simple application created in Model 1-style JSP development. The domain will be separated from the JSP pages, but there will be a lot of scriptlets in the JSP.

The application lists all of the departments in a fictional company. When the users clicks on a department, the browser shows all of the employees in that department in the Employee Listing view. The Employee Listing view allows the user to add, edit, or delete employees for the given department.

The Department Model

First, let's cover the domain objects for this application. There are only two domain objects for this example: Dept (which represents a department) and Employee (which represents an employee). Both the Dept and the Employee can read themselves out of the database. The Employee object can also, update, add, and remove itself. Listing 13.1 shows the class for the Dept domain object, and Listing 13.2 shows the class for the Employee domain object. These classes will be used unchanged in both the Model 1 and MVC versions of the application. Skim Listings 10.1 and 10.2 before we go into the Model 1 application's JSP.

```
package bean;
import java.sql.*;
import java.util.*;
import javax.naming.*;
import javax.sql.DataSource;

public class Dept implements java.io.Serializable {

    private String name;
    private int id;

    private static final String select = "select deptid, name " +
                                         "from dept";
    private static final String selectEmployee =
        "select empid, fname, lname, phone, deptid" +
        " from employee where deptid=? " +
        " order by lname";

    public Dept() {
    }

    public Dept(ResultSet rs) throws SQLException{
        init(rs);
    }
}
```

```
public void init(ResultSet rs) throws SQLException{
    this.name = rs.getString("name");
    this.id = rs.getInt("deptid");
}

public String getName() {
    return this.name;
}

public void setName(String name) {
    this.name = name;
}

public int getId() {
    return this.id;
}

public void setId(int id) {
    this.id = id;
}

static Connection getConnection(){
    try {
        Context initCtx = new InitialContext();
        Context envCtx = (Context) initCtx;
        lookup("java:comp/env");
        DataSource ds = (DataSource) envCtx.lookup("jdbc/emp");
        return ds.getConnection();
    }catch (Exception e){
        e.printStackTrace();
    }
    ...
}
catch (Exception e){
    ...
}
}

public static Dept [] getDepartments()throws SQLException{

    Connection connection = null;
    Statement statement = null;
    ResultSet resultSet = null;
    ArrayList depts = new ArrayList();

    try{

        connection = getConnection();
        statement = connection.createStatement();
        resultSet = statement.executeQuery(select);

        while(resultSet.next()){
            Dept dept = new Dept(resultSet);
            depts.add(dept);
        }
    }finally {
        if (resultSet!=null) resultSet.close();
        if (statement!=null) statement.close();
    }
}
```

```
        if (connection!=null) connection.close();
    }

    Dept [] arrayDepts = new Dept[depts.size()];
    return (Dept []) depts.toArray(arrayDepts);
}

public Employee [] getEmployees() throws SQLException{

    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    ArrayList emps = new ArrayList();

    try{

        connection = getConnection();
        statement = connection.prepareStatement(selectEmployee);
        statement.setInt(1, this.id);

        resultSet = statement.executeQuery();

        while(resultSet.next()){
            Employee emp = new Employee(resultSet);
            emps.add(emp);
        }

    }finally {
        if (resultSet!=null) resultSet.close();
        if (statement!=null) statement.close();
        if (connection!=null) connection.close();
    }

    Employee [] arrayEmps = new Employee[emps.size()];
    return (Employee []) emps.toArray(arrayEmps);
}
}
```

Listing 13.1

The Dept domain object.

As you can see in Listing 13.1, the Dept class has a static method called `getDepartments()`, which returns a Dept array that contains every department in the company. The Dept class also has a method called `getEmployees()`, which returns an array of all the employees in the department. The `getConnection()` helper method gets access to a JDBC connection pool using JNDI to access the component's naming context. The Employee class also uses `getConnection()` to get its JDBC connection.

```
package bean;
import java.sql.*;

public class Employee implements java.io.Serializable{

    private String firstName;
    private String lastName;
    private String phone;
```

```
private int id;
private int deptId;

private static final String insert =
    "insert into employee (fname, lname, phone, deptid) values "
        + "(?,?,?,?)" ;

private static final String delete =
    "delete from employee where empid=" ;

private static final String select =
    "select empid, fname, lname, phone, deptid " +
    " from employee where empid=" ;

public Employee() {
}

public Employee(ResultSet rs) throws SQLException{
    init(rs);
}

public void init(ResultSet rs) throws SQLException{
    this.firstName = rs.getString("fname");
    this.lastName = rs.getString("lname");
    this.phone = rs.getString("phone");
    this.id = rs.getInt("empid");
    this.deptId = rs.getInt("deptid");
}

public Employee(int id) throws SQLException{
    load(id);
}

public String getFirstName() {
    return this.firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return this.lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public String getPhone() {
    return this.phone;
}

public void setPhone(String phone) {
    this.phone = phone;
}
```

```
public int getId() {
    return this.id;
}

public void setId(int id) {
    this.id = id;
}

public int getDeptId() {
    return this.deptId;
}

public void setDeptId(int deptId) {
    this.deptId = deptId;
}

public void remove() throws Exception{
    Connection connection = null;
    Statement statement = null;

    try{

        connection = Dept.getConnection();
        statement = connection.createStatement();

        System.out.println(delete+id);
        int worked = statement.executeUpdate(delete+id);
        if (worked != 1)
            throw new RuntimeException
                ("Unable to delete the employee");

    }finally {
        if (statement!=null) statement.close();
        if (connection!=null) connection.close();
    }
}

public void add() throws SQLException{
    Connection connection = null;
    PreparedStatement statement = null;

    try{

        connection = Dept.getConnection();
        statement = connection.prepareStatement(insert);

        //fname, lname, phone, deptid
        statement.setString(1, this.firstName);
        statement.setString(2, this.lastName);
        statement.setString(3, this.phone);
        statement.setInt(4, this.deptId);

        int worked = statement.executeUpdate();
        if (worked != 1)
            throw new RuntimeException("Unable to add employee");
    }
}
```

```
        }finally {
            if (statement!=null) statement.close();
            if (connection!=null) connection.close();
        }
    }

    public void load() throws SQLException{
        load(this.id);
    }

    public void load(int id) throws SQLException{

        Connection connection = null;
        Statement statement = null;
        ResultSet resultSet = null;

        try{

            connection = Dept.getConnection();
            statement = connection.createStatement();

            resultSet = statement.executeQuery(select+id);

            if(resultSet.next()){
                init(resultSet);
            }else{
                throw new RuntimeException("Unable to load " +
                    employee id=" + id);
            }

        }finally {
            if (resultSet!=null) resultSet.close();
            if (statement!=null) statement.close();
            if (connection!=null) connection.close();
        }

    }

    public void edit() throws Exception{
        remove();
        add();
    }

    public String toString(){
        StringBuffer buf = new StringBuffer(100);
        buf.append("firstName=");
        buf.append(this.firstName);
        buf.append(", lastName=");
        buf.append(this.lastName);
        buf.append(", phone=");
        buf.append(this.phone);
        buf.append(", deptid=");
        buf.append(this.deptId);
        return buf.toString();
    }
}
```

Listing 13.2**The Employee domain object.**

Listing 13.2 shows the Employee class. This class can create, load or remove an employee from the database, or update an employee's record in the database. The code is standard JDBC, and it is assumed you have some JDBC experience. Even if you don't, however, the sample code is straightforward enough for you to follow along.

The Employee class has three constructors: One constructor takes a result set and is used by the Dept class to load a whole department at a time. Another constructor takes an employee ID, which is used to load an employee. The empty Employee constructor is used for serialization.

The Department Database Schema

Listing 13.3 shows the database schema that the Dept and Employee can access. This code is taken from the build script included with this sample application. The build script is similar in form and function to the build scripts described in Chapter 20, "Developing Web Applications with Ant and XDoclet."

```
CREATE TABLE DEPT (
    DEPTID      INT          IDENTITY      PRIMARY KEY,
    NAME        VARCHAR (80)
);

CREATE TABLE EMPLOYEE (
    EMPID       INT          IDENTITY      PRIMARY KEY,
    FNAME       VARCHAR (80),
    LNAME       VARCHAR (80),
    PHONE       VARCHAR (80),
    DEPTID      INT,
    CONSTRAINT  DEPTFK FOREIGN KEY (DEPTID)
               REFERENCES DEPT (DEPTID)
);
```

Listing 13.3**The Dept application schema.**

The Dept and Employee classes make up the model for this application. Neither the model nor the database schema will change when we switch to using Struts and MVC.

The JSPs for the Model 1 Department Application

The Model 1 consists of the following JSPs:

- DeptListing.jsp
- EmployeeUpdate.jsp
- EmployeeDelete.jsp
- EmployeeForm.jsp

- EmployeeListing.jsp
- Footer.jsp
- Header.jsp

DeptListing.jsp lists all the departments in the company. DeptListing.jsp includes Header.jsp and Footer.jsp, as shown here:

```
...
<jsp:include page="Header.jsp">
  <jsp:param name="header" value="Dept Listing"/>
</jsp:include>

...

<%@include file="/Footer.jsp" %>
```

Header.jsp and Footer.jsp provide a common header and footer for all the main pages. Note that the HTML is kept as simple as possible so you can focus on the concepts.

DeptListing.jsp uses the Dept class to load all the departments in the system:

```
<%@page import="bean.*" %>

...
<% Dept[] depts = Dept.getDepartments(); %>
```

DeptListing creates an HTML table of departments. Each row in the table contains the Dept name in one column and a link to the employees in that department in the other column. DeptListing iterates over the array of departments with JSP scriptlets, as shown here:

```
<%
  for (int index=0; index < depts.length; index++){
%>
  <tr>
    <td>
      <%=depts[index].getName()%>
    </td>

    <td>
      <a href="EmployeeListing.jsp?deptid=<%=depts[index].
      getId()%>">
        show
      </a>
    </td>
  </tr>
<%}%>
```

For each iteration, DeptListing creates a link to EmployeeListing.jsp and passes the deptid as a parameter on the link's query string. So far, you will notice that this solution uses a lot of JSP scriptlets. When a user clicks on the Employee listing link, EmployeeListing.jsp lists all the employees for that department. Listing 13.4 shows the complete listing for DeptListing.jsp. As you can see in Listing 13.5, EmployeeListing.jsp is similar to DeptListing.jsp in both structure and content.

```
<%@page import="bean.*" %>

<jsp:include page="Header.jsp">
  <jsp:param name="header" value="Dept Listing"/>
</jsp:include>
```

```

<% Dept[] depts = Dept.getDepartments(); %>
    <table>
<%
for (int index=0; index < depts.length; index++){
%>
    <tr>
        <td>
            <%=depts[index].getName()%>
        </td>
        <td>
            <a
href="EmployeeListing.jsp?deptid=<%=depts[index].getId()%>">show</a>
        </td>
    </tr>
<%}%>
    </table>
<%@include file="/Footer.jsp" %>

```

Listing 13.4

DeptListing.jsp.

```

<%@page contentType="text/html"%>
<%@page import="bean.*" %>
<jsp:include page="Header.jsp">
    <jsp:param name="header" value="Employee Listing"/>
</jsp:include>
<jsp:useBean id="dept" scope="page" class="bean.Dept" />
<jsp:setProperty name="dept" property="id" param="deptid"/>
<a href="EmployeeForm.jsp?add=true&deptid=<%=dept.getId()%>">add</a>
<% Employee[] emps = dept.getEmployees(); %>
    <table>
<%
for (int index=0; index < emps.length; index++){
%>
    <tr>
        <td>
            <%=emps[index].getFirstName()%>
        </td>
        <td>
            <%=emps[index].getLastName()%>
        </td>
        <td>
            <%=emps[index].getPhone()%>
        </td>
        <td>
            <a href="EmployeeForm.jsp?id=<%=emps[index].
getId()%>&edit=true">

```

```

                edit
            </a>
        </td>

        <td>
            <a href="EmployeeDelete.jsp?id=<%=emps[index].
getId()%>">
                delete
            </a>
        </td>
    </tr>
<}%%>
</table>

<%@include file="/Footer.jsp" %>

```

Listing 13.5

EmployeeListing.jsp.

EmployeeListing.jsp uses the Dept JavaBean by using the jsp:useBean action:

```
<jsp:useBean id="dept" scope="page" class="bean.Dept" />
```

The Employee listing features an add link at the top of the page so that a user can add more employees to this department:

```
<a href="EmployeeForm.jsp?add=true&deptid=<%=dept.getId()%>">add</a>
```

EmployeeListing.jsp then calls the getEmployees() method of the Dept JavaBean. It iterates over the employees and fills out columns in the table. Each row represents another employee. Each column represents a different property of the employee bean--except for the last two columns, which are links for editing and deleting the employee associated with that row. These last two action columns are defined as follows:

```

        <td>
            <a href="EmployeeForm.jsp?id=<%=emps[index].
getId()%>&edit=true">
                edit
            </a>
        </td>

        <td>
            <a href="EmployeeDelete.jsp?id=<%=emps[index].getId()%>">
                delete
            </a>
        </td>

```

Notice that both the edit link and the add link point to EmployeeForm.jsp. The add link passes the dept ID on the query string, while the edit link passes the employee ID. Also notice that the edit link passes the edit parameter, which is set to true. This means EmployeeForm.jsp will have to keep track of whether it is in add mode or edit mode, and it will have to populate the HTML form fields if it is in edit mode. Listing 13.6 shows the amount of Java code EmployeeForm.jsp needs to perform its tasks. It is scriptlet overkill. Later when we switch to Struts, it will handle add and edit modes transparently.

```

<%@page contentType="text/html"%>
<jsp:useBean id="employee" scope="page" class="bean.Employee" />
<jsp:setProperty name="employee" property="id" param="id" />

<jsp:include page="Header.jsp">
    <jsp:param name="header" value="Employee Form"/>
</jsp:include>

```

```

<%
    String firstName = "";
    String lastName = "";
    String phone = "";
    String sEdit = request.getParameter("edit");
    String deptID = "0";
    String mode = "";

    boolean edit = sEdit!=null;
    if (edit == true){
        mode = "edit";
        out.println("<h1> edit mode </h1>");
        employee.load();
        firstName = employee.getFirstName();
        lastName = employee.getLastName();
        phone = employee.getPhone();
        deptID = ""+employee.getDeptId();
    }
    else{
        mode="add";
        out.println("<h1> add mode </h1>");

        deptID=request.getParameter("deptid");
    }
%>

<form action="EmployeeUpdate.jsp" method="POST" >
    First Name <input type="TEXT" name="firstName" value="<%=
firstName%>"/>
    Last Name <input type="TEXT" name="lastName" value="<%=
lastName%>"/>
    Phone <input type="TEXT" name="phone" value="<%=phone%>"/>
    <input type="HIDDEN" name="deptId" value="<%=deptID%>"/>
    <input type="HIDDEN" name="id" value="<jsp:getProperty name=
"employee" property="id"/>"/>
    <input type="HIDDEN" name="mode" value="<%=mode%>"/>
    <input type="SUBMIT" />
</form>

<%@include file="/Footer.jsp" %>

```

Listing 13.6

EmployeeForm.jsp.

EmployeeForm.jsp has an HTML form that posts to EmployeeUpdate.jsp, shown in Listing 13.7. EmployeeUpdate.jsp could be written as a servlet since it is essentially nongraphical. Most of the logic for managing mode was done by EmployeeForm.jsp, so EmployeeUpdate.jsp is fairly simple. It maps the HTML form parameters to the Employee bean and then calls add() or edit(), depending on what mode the form was in.

```

<%@page contentType="text/html"%>
<html>
<head><title>Employee Add/Edit</title></head>
<body>

<jsp:useBean id="employee" scope="page" class="bean.Employee" />
<jsp:setProperty name="employee" property="*" />

<%

```

```
String mode=request.getParameter("mode");

if (mode.equals("add")){
    employee.add();

}else{
    employee.edit();
}
%>

Employee <%=employee.getLastName()%>

<a href="EmployeeListing.jsp?deptid=<%=employee.getDeptId()%>">
    back to listing </a>

</body>
</html>
```

Listing 13.7

EmployeeUpdate.jsp.

That's it. As you can see, it is a fairly simple application, but it proves how powerful JSP is. With a minimal amount of code, we implemented add, edit, list, and delete functionality for Employees. However, we have not done a good job of separating the presentation logic from the HTML. This code base would be difficult to maintain, and it would be hard to solicit the help of a Web designer. In the next section, we learn how to use Struts to get the Java out of our JSPs so that we can live in peace and harmony with our Web designer friends.

The MVC Struts Version of the Application

The Dept and Employee classes make up the Model for this application, too. The Model is unchanged from our previous example.

The MVC version consists of the following JSPs:

- DeptListing.jsp
- EmployeeForm.jsp
- EmployeeListing.jsp
- Footer.jsp
- Header.jsp

The MVC version also includes the following Actions:

- DeleteEmployeeAction.java

- EditEmployeeAction.java
- ListDepartmentsAction.java
- ListEmployeesAction.java
- UpdateEmployeeAction.java

And the following ActionForm:

- EmployeeForm.java

At first, this seems like a lot of extra components--and it is--but it isn't as bad as you think. The Actions separate the Java code from the JSPs and are relatively small.

List Department Action

ListDepartmentsAction.java works in conjunction with DeptListing.jsp to list all the departments in the company. We moved all the necessary Java code out of DeptListing.jsp and put it into ListDepartmentsAction.java.

An action mapping binds ListDepartmentsAction.java and DeptListing.jsp to a path /listDepartments.do, as follows:

```
<action path="/listDepartments"
        type="action.ListDepartmentsAction"
        scope="request"
        validate="true">
  <forward name="listing"
           path="/DeptListing.jsp">
  </forward>
</action>
```

Actions interact with the Model classes and then delegate the display to JSP. In this case, action.ListDepartmentsAction will get the departments out of the Model and pass them to the DeptListing.jsp to display.

You may wonder why the link ends in *.do*. This identifies the action so that the Action Servlet (org.apache.struts.action.ActionServlet) can handle these requests. The configuration is stored in /WEB-INF/struts-config.xml as specified by the config parameter in the Action Servlet declaration in the deployment descriptor (/WEB-INF/web.xml), as shown here:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-
class>
  <init-param>
    <param-name>application</param-name>
    <param-value>ApplicationResources</param-value>
  </init-param>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  ...
```

```
        <load-on-startup>2</load-on-startup>
    </servlet>
```

There is an servlet mapping entry in the deployment descriptor so that the Action Servlet handles all requests that end in *.do, as follows:

```
        <!-- Standard Action Servlet Mapping -->
    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
```

The ListDepartmentsActionclass is particularly short. It gets all of the departments in the system, maps the departments into request scope, and then forwards *listing*. From the action mapping above, you can see that *listing* is mapped to DeptListing.jsp.

The implementation of this Action has only two lines of code in the execute() method:

```
package action;
...

import bean.Dept;
...
public class ListDepartmentsAction extends Action {
...
    public ActionForward execute(
        ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        request.setAttribute("departments", Dept.getDepartments());
        return mapping.findForward("listing");

    }
}
```

Action Details

Notice that the ListDepartmentAction subclasses the Action class. The Action class defines an execute() method, as follows:

```
    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
        throws Exception;
```

Action classes process requests via their execute() method and return an ActionForward object. The ActionForward object identifies where control should be forwarded (in our case, DeptListing). Most of your interaction with the Model classes will occur in the Actions. The Model will in turn do things such as add rows to database tables.

When the Action has finished interacting with the Model, it returns an appropriate ActionForward object, which identifies the JSP page to be used to generate the View. The Action and the View (forward) are defined in the configuration file so that they can be changed without modifying any code. The findForward() method of the action mapping is used to look up the View. For example, if you wanted to add steps in a user registration system, you could do so without affecting your current code base. Just add the new Actions and JSP, and wire them into the application using the Struts configuration file.

The Struts DeptListing View

Notice that the Struts DeptListing in Listing 13.8 is considerably different than the DeptListing in Listing 13.4. First, the Struts DeptListing has no JSP scriptlet and no import page directives; instead, it uses the custom tags that come with Struts.

```
<%@ taglib uri="strutslogic" prefix="logic" %>
<%@ taglib uri="strutsbean" prefix="bean" %>
<%@ taglib uri="strutshtml" prefix="html" %>

<jsp:include page="Header.jsp">
  <jsp:param name="header" value="Dept Listing"/>
</jsp:include>

<table>

  <logic:iterate id="dept" name="departments">

    <tr>
      <td>
        <bean:write name="dept" property="name" />
      </td>

      <td>
        <html:link page="/listEmployees.do"
          paramId="deptid" paramName="dept"
          paramProperty="id">
          show
        </html:link>
      </td>
    </tr>
  </logic:iterate>

</table>

<%@include file="/Footer.jsp" %>
```

Listing 13.8

Struts DeptListing.jsp.

The structure of the JSP is similar to our previous example. For instance, we still use the include action to dynamically include Header.jsp:

```
<jsp:include page="Header.jsp">
  <jsp:param name="header" value="Dept Listing"/>
</jsp:include>
```

The Struts Custom Tags

One way the Struts DeptListing gets rid of the JSP scriptlets is by using these Struts tag libraries: bean, logic, and html. We use these taglibs throughout this chapter to remove Java code from our JSPs.

The Struts DeptListing uses the taglib directive to import three taglibs, as follows:

```
<%@ taglib uri="strutslogic" prefix="logic" %>
<%@ taglib uri="strutsbean" prefix="bean" %>
```

```
<%@ taglib uri="strutshtml" prefix="html" %>
```

The tag library descriptor files are stored in /WEB-INF/tlds. Then, a reference to the tag libraries is defined in the Web application deployment descriptor (/WEB-INF/web.xml) as follows:

```
<!-- Struts Tag Library Descriptor -->
<taglib>
  <taglib-uri>
    strutsbean
  </taglib-uri>
  <taglib-location>
    /WEB-INF/tlds/struts-bean.tld
  </taglib-location>
</taglib>
<taglib>
  <taglib-uri>
    strutshtml
  </taglib-uri>
  <taglib-location>
    /WEB-INF/tlds/struts-html.tld
  </taglib-location>
</taglib>
<taglib>
  <taglib-uri>
    strutslogic
  </taglib-uri>
  <taglib-location>
    /WEB-INF/tlds/struts-logic.tld
  </taglib-location>
</taglib>
...

```

Logic Iterate Struts Custom Tag

Remember that the ListDepartments action mapped the departments to our request object. Instead of using a JSP scriptlet to iterate over each entry in the department list in order to build an HTML table, this version of DeptListing.jsp uses the Struts iterate tag:

```
<logic:iterate id="dept" name="departments">
  <tr>
    ...
  </tr>
</logic:iterate>
```

Compare this to the first version of DeptListing, which used a scriptlet:

```
<%
  for (int index=0; index < depts.length; index++){
%>
```

If you are a Web designer, the Struts version looks less intimidating.

Note that the iterate tag specifies that it is going to iterate over the departments array with the name attribute--that is, *name="departments"*. Each iteration extracts a department and maps to a dept mapped to the page scope of our JSP, as specified by the id attribute--that is, *id="dept"*.

Bean Write Struts Custom Tag

The dept that is mapped to our page scope is an instance of bean.Dept. You may recall that bean.Dept has a property called name. Each iteration defines another row in the HTML table with the dept name and a link to the listing of employees in the dept. The dept name is extracted with the bean:write custom tag:

```
<td>
  <bean:write name="dept" property="name" />
</td>
```

Note that the name attribute specifies the bean we wish to write out, and the property specifies which field of the beans we wish to write out. Compare this to the last version of DeptListing:

```
<td>
  <%=depts[index].getName()%>
</td>
```

This Struts version seems to be a little more verbose, but the former version requires knowledge of Java arrays and indexing.

HTML Link Struts Custom Tag

Next, we need to add a link to the employee listing using the html:link tag:

```
<td>
  <html:link page="/listEmployees.do"
            paramId="deptid"
            paramName="dept" paramProperty="id">
    show
  </html:link>
</td>
```

Notice that we do not link directly to EmployeeListing.jsp. This is because we've divided what was in EmployeeListing into two components: the ListEmployeesAction class and EmployeeListing.jsp (more on this later in this section). The html:link tag specifies the page to link with the page attribute--/listEmployees.do in this case. The link tag also lets you specify the name of the parameter that you wish to pass using the paramId attribute--*paramId="deptid"*. The link tag will automatically build the correct query string and encode it if necessary. The html:link tag uses paramName and paramProperty to specify the name of the bean and the name of the property it will use to set the deptid parameter of the query string. The above html:link is equivalent to this code in the first non-Struts DeptListing.jsp as follows:

```
<td>
  <a href="EmployeeListing.jsp?deptid=<%=depts[index].
  getId()%>">
    show
  </a>
</td>
```

The problem with the first sample solution for adding the EmployeeListing link is that it will not automatically encode the jsessionid; if users have cookies turned off, the application will lose their session information when they click on the EmployeeListing link. The html:link takes care of encoding the URL for us so the link will not lose the session information if the user has cookies turned off.

The html:link for the Struts employee listing points to /listEmployees.do. The listEmployees action is defined with the following action mapping in the configuration file:

```
<action path="/listEmployees"
        type="action.ListEmployeesAction"
        scope="request"
        validate="true">
  <forward name="listing"
          path="/EmployeeListing.jsp">
```

```
</forward>
</action>
```

EmployeeListing: Action and View

Thus the Action Servlet directs the request specified by /listEmployees.do to the action.ListEmployeeAction class, shown in Listing 13.9.

```
package action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import bean.Dept;

/**
 * @author Rick Hightower
 * @struts:action
 *          path="/listEmployees"
 *
 * @struts:action-forward name="listing" path="/EmployeeListing.jsp"
 */
public class ListEmployeesAction extends Action {

    /**
     * @see org.apache.struts.action.Action#execute(ActionMapping,
     * ActionForm,
     *          HttpServletRequest, HttpServletResponse)
     */
    public ActionForward execute(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {

        Dept dept = new Dept();

        int deptid = Integer.parseInt(
            request.getParameter("deptid"));

        dept.setId(deptid);

        request.setAttribute("dept", dept);
        return mapping.findForward("listing");
    }
}
```

Listing 13.9

ListEmployeesAction.java.

The ListEmployeesAction creates a new Dept domain object called dept:

```
Dept dept = new Dept();
```

Then, it gets the deptid from the request--that is, from the query string parameter deptid, as shown here:

```
int deptid = Integer.parseInt(request.getParameter("deptid"));
```

It passes the deptid to the Dept domain object, and then maps the dept to the request so that the Dept domain object can use the ID to load the employees for the department and the JSP can use the Dept instance to display department information of the end user:

```
dept.setId(deptid);
request.setAttribute("dept",dept);
```

Finally, it forwards this request to the View--that is, the listing, which is mapped to EmployeeListing.jsp in the Struts configuration file:

```
return mapping.findForward("listing");
```

The View, EmployeeListing.jsp, uses the dept object to iterate over a list of employees:

```
<logic:iterate id="employee" name="dept" property="employees">
  <tr>
...
  </tr>
</logic:iterate>
```

Notice that this time we use the property parameter to specify that we want the employees property from the dept object. The above code is functionally equivalent to the following non Struts code in the original EmployeeListing.jsp as follows:

```
<% Employee[] emps = dept.getEmployees(); %>
...
<%
  for (int index=0; index < emps.length; index++){
%>
  <tr>
...
  </tr>
<%}%>
```

Here we can see that using custom tags and Struts really pays off--our code is much simpler and relies a lot less on the Web designer's knowledge of Java. The complete listing for the Struts version of EmployeeListing appears in Listing 13.10. Please take some time and compare this to Listing 13.5 (the original Model 1 EmployeeListing.jsp).

```
<%@ taglib uri="strutslogic" prefix="logic" %>
<%@ taglib uri="strutsbean" prefix="bean" %>
<%@ taglib uri="strutshtml" prefix="html" %>

<jsp:include page="Header.jsp">
  <jsp:param name="header" value="Employee Listing"/>
</jsp:include>

<html:link      page="/EmployeeForm.jsp"
                paramId="deptid"
                paramName="dept"      paramProperty="id">
  add
</html:link>
```

```
<table>

<logic:iterate id="employee" name="dept" property="employees">
  <tr>
    <td>
      <bean:write name="employee" property="firstName" />
    </td>

    <td>
      <bean:write name="employee" property="lastName" />
    </td>

    <td>
      <bean:write name="employee" property="phone" />
    </td>

    <td>
      <html:link      page="/editEmployee.do"
                    paramId="id"
                    paramName="employee" paramProperty="id">
        edit
      </html:link>
    </td>

    <td>
      <html:link      page="/deleteEmployee.do"
                    paramId="id"
                    paramName="employee" paramProperty="id">
        delete
      </html:link>
    </td>

  </tr>
</logic:iterate>

</table>

<%@include file="/Footer.jsp" %>
```

Listing 13.10

Struts EmployeeListing.jsp.

The Struts EmployeeListing has an add link just as before, and for each employee there is also an edit link and a delete link. Notice that the edit link points to /editEmployee.do while the add link point to /EmployeeForm.jsp. The add link passes the dept ID on the query string, while the edit link passes the employee ID just as before.

Managing Form Data with Struts

In our previous non-Struts example, EmployeeForm.jsp had to keep track of whether it was in add or edit mode, and it populated the HTML form fields if it was in edit mode. Look back to Listing 13.5 to see the amount of Java code EmployeeForm.jsp needed to perform its tasks; as we mentioned then, it was scriptlet overkill. This time, EditEmployeeAction will (with some help from the Struts framework) handle the population of the form fields. Listing 13.11 shows EditEmployeeAction.java.

```
package action;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import bean.Employee;
import form.EmployeeForm;

/**
 * @author Rick Hightower
 * @struts:action path="/editEmployee"
 *                 attribute="employeeForm"
 *
 * @struts:action-forward name="form" path="/EmployeeForm.jsp"
 */
public class EditEmployeeAction extends Action {

    /**
     * @see org.apache.struts.action.Action#execute(ActionMapping,
     */
    public ActionForward execute(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {

        int empid = Integer.parseInt(request.getParameter("id"));

        /* Populate the employee form */
        if (form == null) {
            form = new EmployeeForm();
            if ("request".equals(mapping.getScope())) {
                request.setAttribute(mapping.getAttribute(),
                    form);
            } else {
                request.getSession()
                    .setAttribute(mapping.getAttribute(), form);
            }
        }

        /* Load the employee information */
        EmployeeForm eform = (EmployeeForm) form;
        eform.getEmployee().load(empid);

        /* Set the mode attributes */
        eform.setAction(EmployeeForm.EDIT_MODE);

        return mapping.findForward("form");
    }
}
```

Listing 13.11

Struts EditEmployeeAction.java.

The action mapping for EditEmployeeAction is defined in the Struts configuration file as follows:

```
<action path="/editEmployee"
        type="action.EditEmployeeAction"
        attribute="employeeForm"
        scope="request"
        validate="true">
  <forward name="form"
          path="/EmployeeForm.jsp">
</forward>
</action>
```

Notice that the attribute refers to the name that the EmployeeForm bean will be mapped to in the request (more on this later in this section).

EditEmployeeAction looks up the employee ID passed on the query string:

```
int empid = Integer.parseInt(request.getParameter("id"));
```

EditEmployeeAction then creates a new EmployeeForm (which is defined in Listing 13.12) and maps the employee form to the session object or the request object, depending on how the scope was defined in the ActionMapping:

```
/* Populate the employee form */
if (form == null) {
  form = new EmployeeForm();
  if ("request".equals(mapping.getScope())) {
    request.setAttribute(mapping.getAttribute(), form);
  } else {
    request.getSession()
      .setAttribute(mapping.getAttribute(), form);
  }
}
```

EmployeeForm has a property that refers to a bean.Employee--in our case, the Employee domain object. EditEmployeeAction passes the empid to the nested bean.Employee instance via the load() method. You may recall that the load() method loads the Employee information that is stored in the database. The code that loads the nested employee property in EmployeeForm is defined as follows:

```
/* Load the employee information */
EmployeeForm eform = (EmployeeForm) form;
eform.getEmployee().load(empid);
```

EmployeeForm has a property called action, which specifies whether EmployeeForm should be in add, edit, or delete mode. EditEmployeeAction sets the Action to edit mode, as shown here:

```
/* Set the mode attributes */
eform.setAction(EmployeeForm.EDIT_MODE);
```

Finally, EditEmployeeAction forwards this request to the View:

```
return mapping.findForward("form");
```

The form forward is mapped to EmployeeForm.jsp in the Struts configuration file--<forward name="form" path="/EmployeeForm.jsp">. The EmployeeForm.java extends ActionForm and is used to populate the HTML form in EmployeeForm.jsp, and validate form data from Employee.jsp (see Listings 13.12 and 13.13 for the complete code for EmployeeForm.java and EmployeeForm.jsp). Before we delve into the details of how EmployeeForm.jsp is implemented, let's cover some background on form beans.

```
package form;
```

```
import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

import bean.Employee;

/**
 * @author Rick Hightower
 *
 * @struts:form name="employeeForm"
 */
public class EmployeeForm extends ActionForm {

    public static final String EDIT_MODE = "edit";
    public static final String DELETE_MODE = "delete";
    public static final String ADD_MODE = "add";

    String action;

    Employee employee;

    public EmployeeForm() {
        employee = new Employee();
        action = EmployeeForm.ADD_MODE;
    }

    public Employee getEmployee() {
        return employee;
    }

    public void setEmployee(Employee employee) {
        this.employee = employee;
    }

    /**
     * Returns the action.
     * @return String
     */
    public String getAction() {
        return action;
    }

    /**
     * Sets the action.
     * @param action The action to set
     */
    public void setAction(String action) {
        this.action = action;
    }

    /**
     * @see org.apache.struts.action.ActionForm#reset(ActionMapping,
     *                                             HttpServletRequest)
     */
    public void reset(ActionMapping mapping,
        HttpServletRequest request) {
```

```

        this.employee = new Employee();
        this.action = ADD_MODE;
    }

/**
 * @see org.apache.struts.action.ActionForm#validate(ActionMapping,
 *                                                     HttpServletRequest)
 **/
    public ActionErrors validate(ActionMapping arg0,
                                HttpServletRequest arg1) {
        ActionErrors errors = new ActionErrors();
        if ((employee.getFirstName() == null)
            || (employee.getFirstName().length() < 3)) {

            errors.add("FirstName", new ActionError("error.employee.
                firstname"));

        }
        return errors;
    }
}

```

Listing 13.12

Struts form bean EmployeeForm.java.

```

<%@ taglib uri="strutsbean"   prefix="bean"   %>
<%@ taglib uri="strutshtml"   prefix="html"   %>
<%@ taglib uri="strutslogic"  prefix="logic" %>

<jsp:include page="Header.jsp">
    <jsp:param name="header" value="Employee Form"/>
</jsp:include>

<logic:equal name="employeeForm" property="action"
             scope="request" value="edit">
    <h3>Editing employee</h3>
</logic:equal>

<logic:equal name="employeeForm" property="action"
             scope="request" value="delete">
    <h3>Are you sure you want to delete the following employee?</h3>
</logic:equal>

<logic:equal name="employeeForm" property="action"
             scope="request" value="add">
    <h3>Adding a new employee</h3>
</logic:equal>

<html:errors/>

<html:form action="/updateEmployee.do" method="POST">

    First Name <html:text property="employee.firstName" />

    Last Name <html:text property="employee.lastName" />

```

```
Phone <html:text property="employee.phone" />
<html:hidden property="employee.id" />
<html:hidden property="employee.deptId" />

<html:hidden property="action"/>
<html:submit property="submit"/>
<input type="hidden" name="deptid" value="<%=request.
getParameter("deptid")%>" />

</html:form>

<%@include file="/Footer.jsp" %>
```

Listing 13.13

Struts EmployeeForm.jsp.

Background Information on ActionForms

ActionForm beans allow you to create a bridge between HTML forms in the View and domain objects in the Model. EmployeeForm.java defines an ActionForm bean. EmployeeForm, which extends the ActionForm class, represents the HTML input form parameters used in the Struts version of EmployeeForm.jsp (which we define in detail later in this section). We declare the EmployeeForm form bean in the Struts configuration file as follows:

```
<form-beans>
  <form-bean name="employeeForm"
            type="form.EmployeeForm" />
</form-beans>
```

The Struts Action Servlet, the Controller servlet, will use EmployeeForm to automatically map a new EmployeeForm bean into the request or session if it does not exist. Then, it will use EmployeeForm to map any HTML form request parameter whose name corresponds to the name of a property in the bean or in the nested bean properties. This is similar to the standard JSP action `<jsp:setProperty>` when you use the asterisk wildcard to select all properties; this is more powerful, however, since it can work with nested beans, and it features support for validation.

Form Validation and Application Resources

The EmployeeForm bean overrides the `validate()` method and provides error messages in the standard application resource. The Struts framework uses this `validate()` method to notify you when in the life cycle of the form bean to do the validation. EmployeeForm defines the `validate()` method as follows:

```
public ActionErrors validate(ActionMapping arg0,
    HttpServletRequest arg1) {
    ActionErrors errors = new ActionErrors();
    if ((employee.getFirstName() == null)
        || (employee.getFirstName().length() < 3)) {

        errors.add("FirstName",
            new ActionError("error.employee.firstname"));
    }
    return errors;
}
```

This is a simple example of some possible validation. It just checks if the employee's first name is present and, if it is, whether it is greater than three characters. If it is not valid, the `validate()` method creates a new `ActionError` and adds it to the list of errors. The `ActionError` refers to an error message (`error.employee.firstname`) defined in `ApplicationResources.properties`, which is located in the classes

directory of the Web application. The ApplicationResources defines String constants used by the application and can be used to help create an internationalized version of our application. ApplicationResources.properties defines the following properties for validating this form:

```
error.employee.firstname=<li>First name must be set and greater than
  three characters</li>

errors.header=<h3><font color="red">There were problems processing
  your form: <ul>

errors.footer=</ul></font><hr>
```

The errors.header and errors.footer are used to define the boundaries of the error block. The error.employee.firstname is the message that will be used for this applicaiton. If we want to provide a German-language version of ApplicationResources, we could also define an ApplicationResource_dn.properties file, for Japanese an ApplicationResource_jp.properties file, and on.

View: Forms and Displaying Validation Errors to Users

The EmployeeForm.jsp file uses the html:errors tag to inform the user of any errors that were created by our validate() method:

```
<html:errors/>
```

The EmployeeForm form bean has a nested property reference to bean.Employee, the employee domain object. This "employee" property will be used to map properties of the employee to form parameters in EmployeeForm.jsp:

```
<html:form action="/updateEmployee.do" method="POST">

  First Name <html:text property="employee.firstName" />

  Last Name <html:text property="employee.lastName" />
  Phone <html:text property="employee.phone" />
  <html:hidden property="employee.id" />
  <html:hidden property="employee.deptId" />

  <html:hidden property="action"/>
  <html:submit property="submit"/>
  <bean:parameter id="deptid" name="deptid"/>
  <input type="hidden" name="deptid" value="<bean:write
  name="deptid" />" /> />

</html:form>
```

Notice that all of the properties are passed back as either fields the user can modify or as hidden fields. The mode is sent via the action property.

EmployeeForm.jsp's HTML form posts its form data to the /updateEmployee.do action. The updateEmployee action mapping is defined as follows:

```
<action path="/updateEmployee"
  type="action.UpdateEmployeeAction"
  name="employeeForm"
  attribute="employeeForm"
  scope="request"
  input="/EmployeeForm.jsp"
  validate="true">
  <forward name="listing"
    path="/EmployeeListing.jsp">
  </forward>
</action>
```

Updating the Employee

Thus, the path `updateEmployee.do` is mapped to `UpdateEmployeeAction`, which is shown in Listing 13.14. Note that the name attribute specifies the form that will be used--in this case, it refers to `employeeForm`, which is mapped to the `EmployeeForm` form bean. `UpdateEmployeeAction` does one of the following--adds an employee, updates an employee, or deletes an employee--based on what the `EmployeeForms` action property is set to. Then, it shows the listing of employees by forwarding it back to `EmployeeListing.jsp`.

```
package action;

import org.apache.struts.action.*;
import javax.servlet.http.*;
import form.EmployeeForm;
import bean.Dept;
import bean.Employee;

/**
 * @author Rick Hightower
 * @struts:action name="employeeForm"
 *                 path="/updateEmployee"
 *                 input="/EmployeeForm.jsp"
 *                 attribute="employeeForm"
 *
 * @struts:action-forward name="listing" path="/EmployeeListing.jsp"
 */
public class UpdateEmployeeAction extends Action {

    public ActionForward execute(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {

        /* Cast the ActionForm into EmployeeForm */
        EmployeeForm eform = (EmployeeForm) form;

        /* Add, edit or delete this employee */

        /* ADD */
        if (EmployeeForm.ADD_MODE.equals(eform.getAction())) {

            Employee employee = eform.getEmployee();
            int deptid = Integer.parseInt(
                request.getParameter("deptid"));

            employee.setDeptId(deptid);
            eform.getEmployee().add();

            /* EDIT */
        } else if (EmployeeForm.EDIT_MODE.equals(eform.getAction())) {
            eform.getEmployee().edit();

            /* DELETE */
        } else if (EmployeeForm.DELETE_MODE.equals(eform.getAction())) {
            eform.getEmployee().remove();

            /* ILLEGAL STATE */
        } else {
```

```

        throw new java.lang.IllegalStateException("invalid " +
            "action");
    }

    Dept dept = new Dept();
    int deptid = eform.getEmployee().getDeptId();
    dept.setId(deptid);
    request.setAttribute("dept", dept);

    return mapping.findForward("listing");
}
}

```

Listing 13.14

Struts UpdateEmployeeAction.java.

Let's walk through UpdateEmployeeAction step by step since it is by far the most complicated action. Struts will instantiate an EmployeeForm and populate its properties and nested bean properties. Then it will allow EmployeeForm to validate itself. If the validation fails, the user will be shown the error message and given a chance to change things; if the validation passes, the execute() method of UpdateEmployeeAction will be called. Therefore, for all intents and purposes the form bean passed to the execute() method is a valid one. This means that the first thing the execute() method does is cast the ActionForm form to an EmployeeForm, as shown here:

```

    /* Cast the ActionForm into EmployeeForm */
    EmployeeForm eform = (EmployeeForm) form;

```

You may recall that the employee form has an action property. The action property specifies what mode the form was submitted in--add, edit, or delete. We check to see if the form is in add mode:

```

    /* ADD */
    if (EmployeeForm.ADD_MODE.equals(eform.getAction())) {

        Employee employee = eform.getEmployee();
        int deptid = Integer.parseInt(
            request.getParameter("deptid"));

        employee.setDeptId(deptid);
        employee().add();

    }

```

If the form was in add mode, then we get the nested employee from the form, set the employee dept id, and then call the employee.add() method to store the employee in the database. The EmployeeForm form bean default action mode is add. If the employee form's mode is set to edit, as shown here:

```

    /* EDIT */
    } else if (EmployeeForm.EDIT_MODE.equals(eform.getAction())) {
        eform.getEmployee().edit();
    }

```

then we get the nested employee domain object and call its edit() method to update the employee in the database. You may recall that EditEmployeeAction populated the form bean and set its action mode to edit. If the employee form's mode is set to delete, as shown here:

```

    /* DELETE */
    } else if (EmployeeForm.DELETE_MODE.equals(eform.getAction())) {
        eform.getEmployee().remove();
    }

```

```
}

```

then we get the nested employee domain object and call its `remove()` method to remove the employee from the database. `DeleteEmployeeAction` is similar to `EditEmployeeAction` in that it sets the mode to delete (they are so similar that we do not cover both actions, just `EditEmployeeAction`).

Now we need to prepare the employee listing view, as shown here:

```
Dept dept = new Dept();
int deptid = eform.getEmployee().getDeptId();
dept.setId(deptid);
request.setAttribute("dept", dept);

```

This code prepares the `dept` so that the view can use the `dept` to get the list of current employees. Finally, we forward this request to `EmployeeListing`:

```
return mapping.findForward("listing");

```

That's it. It is still a fairly simple application, and it is a lot more functional than before. You can see again the power of Struts. With just a little more code, we implemented add, edit, list, and delete functionality for Employees, with the added benefits of validation, encoded URLs, and more. Unlike with the Model 1 version of the application, we have done a good job of separating the presentation logic from the HTML. This code base will be easier to maintain and change. We have removed our Java code from our JSPs, and we can live in peace and harmony with our Web designer friends just as promised.

Templates

The Struts template custom tags are used to create templates for groups of pages that share a common format. The functionality of these templates is similar to the JSP include directive or the JSP include action, but more dynamic than the include directive and more flexible than the include action.

Previously, we used the JSP include action to dynamically include the header and footer files as runtime resources:

```
<jsp:include page="Header.jsp">
  <jsp:param name="header" value="Dept Listing"/>
</jsp:include>
...
<%@include file="/Footer.jsp" %>

```

This is okay, but what if we want the copyright information somewhere other than the footer? This approach is not very flexible. A more flexible approach would let us define a template, and the individual pages would insert their components into the template. You can think of templates as the reverse of dynamic includes.

The template taglib contains three custom tags: `put`, `get`, and `insert`. `Put` tags put content into request scope, and this content is used by the template JSP to fill itself in using the `get` tag. Templates are included with the `insert` tag.

The first step is to define a template JSP (`Template.jsp`) as follows:

```
<%@ taglib uri='strutstemplate' prefix='template' %>
<%@ taglib uri="strutshtml" prefix="html" %>

<html:html>
<head>
  <html:base/>
  <title>
    <template:get name='header' />

```

```

        </title>
    </head>

    <body>
    <h1><template:get name='header' /></h1>

    <table border='1'>
        <tr>
            <td>
                <template:get name='navigation' />
            </td>
            <td>
                <template:get name='content' />
            </td>
        </tr>
    </table>

    <template:get name='footer' />
    </body>

</html:html>

```

This code would specify the layout for our pages; it is saved in a file called Template.jsp. The `template:get` custom tag gets content from the pages that are using this template. The template expects the pages to fill out the navigation, header, content, and footer. Here is an example of DeptListing redone to use templates:

```

<%@ taglib uri="strutslogic" prefix="logic" %>
<%@ taglib uri="strutsbean" prefix="bean" %>
<%@ taglib uri="strutshtml" prefix="html" %>
<%@ taglib uri="strutstemplate" prefix="template" %>

<template:insert template="/Template.jsp">
    <template:put name='header' direct='true' content='Department
Listing' />
    <template:put name='footer' content='/CopyLeft.jsp' />
    <template:put name='navigation' content='/Navigation.jsp' />
    <template:put name='content' >
    <table>
        <logic:iterate id="dept" name="departments">
            <tr>
                <td>
                    <bean:write name="dept" property="name" />
                </td>
                <td>
                    <html:link page="/listEmployees.do"
                        paramId="deptid"
                        paramName="dept" paramProperty="id">
                        show
                    </html:link>
                </td>
            </tr>
        </logic:iterate>
    </table>
    </template:put >
</template:insert>

```

You'll notice how much shorter this listing is than the last version of DeptListing.jsp. Also notice that there is almost no code that specifies the layout of the page. Let's step through the template version of DeptLising.jsp.

The first step is to import the Struts template taglib:

```
<%@ taglib uri="strutstemplate" prefix="template" %>
```

Then, we use the `template:insert` custom tag to insert the template:

```
<template:insert template="/Template.jsp">
</template:insert>
```

The template parameter specifies the page we are going to use as a template, which was shown earlier. Although we insert the template, we still have to fill in the missing pieces using the `template:put` custom tag:

```
<template:put name='footer' content='/CopyLeft.jsp' />
```

The line of code `<template:put name='footer' content='/CopyLeft.jsp' />` inserts the full output of `CopyLeft.jsp` in the location defined in the template as `<template:get name='footer'/>`. The name attribute of the put in this page will be put where the corresponding name attribute is used with a `template:get` in template `jsp`.

In addition to inserting the full output of JSP pages, you can include just the contents of a String. This is really useful for elements such as page titles:

```
<template:put name='header' direct='true' content='Department
Listing'/>
```

This code snippet uses `template:put` to replace the header get in the template with the String "Department Listing". The `direct` parameter specifies that this will be a simple String in the content. If the `direct` parameter is false (the default), then the content parameter is assumed to be a resource, such as a JSP page.

You can also insert the body of the `template:put` custom tag, as shown here:

```
<template:put name='content' >
<table>
  <logic:iterate id="dept" name="departments">
    <tr>
      <td>
        <bean:write name="dept" property="name" />
      </td>
      <td>
        <html:link page="/listEmployees.do"
          paramId="deptid"
          paramName="dept" paramProperty="id">
          show
        </html:link>
      </td>
    </tr>
  </logic:iterate>
</table>
```

Notice here that we do not have a content attribute; instead, `template:put` has a body.

Struts and XDoclet

In Chapter 20 we use XDoclet for servlets to keep the mapping and parameters in sync with the deployment descriptor, and we use XDoclet to keep the variables and attributes in sync with our custom tags. Well, we can also use XDoclet to keep our Action and form beans in sync with our Struts configuration file. In this chapter, we've used XDoclets to generate the Struts configuration file.

Each action defines an action mapping with JavaDoc tags--for example, `UpdateEmployeeAction` uses the `@struts:action` tag to define an action mapping:

```
/**
 * @author Rick Hightower
 * @struts:action name="employeeForm"
 *                path="/updateEmployee"
 *                input="/EmployeeForm.jsp"
```

```

*           attribute="employeeForm"
*
* @struts:action-forward name="listing" path="/EmployeeListing.jsp"
*/
public class UpdateEmployeeAction extends Action {

```

In addition, you can create entries in the Struts configuration file for form beans. Here, EmployeeForm uses the @struts:form to define a form entry:

```

/**
* @author Rick Hightower
*
* @struts:form name="employeeForm"
*/
public class EmployeeForm extends ActionForm {

```

The Struts configuration file generated appears in Listing 13.15. We find it much easier to generate this file rather than to create it by hand.

```

<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD
  Struts Configuration 1.0//EN" http://jakarta.apache.org/struts/
  dtds/struts-config_1_0.dtd">

<struts-config>

  <!-- ===== Form Bean Definitions ===== -->
  <form-beans>
    <form-bean name="employeeForm"
      type="form.EmployeeForm" />

    <!-- If you have non XDoclet forms, define them in a file called
    struts-forms.xml and place it in your merge directory. -->
  </form-beans>

  <!-- ===== Global Forward Definitions ===== -->
  <!--
  Define your forwards in a file called global-forwards.xml and
  place it in your merge directory.
  -->

  <!-- ===== Action Mapping Definitions ===== -->
  <action-mappings>
    <action path="/deleteEmployee"
      type="action.DeleteEmployeeAction"
      attribute="employeeForm"
      scope="request"
      validate="true">
      <forward name="form"
        path="/EmployeeForm.jsp">
      </forward>
    </action>
    <action path="/editEmployee"
      type="action.EditEmployeeAction"
      attribute="employeeForm"
      scope="request"
      validate="true">
      <forward name="form"
        path="/EmployeeForm.jsp">
      </forward>
    </action>
  </action-mappings>

```

```

<action path="/listDepartments"
        type="action.ListDepartmentsAction"
        scope="request"
        validate="true">
  <forward name="listing"
          path="/DeptListing.jsp">
  </forward>
</action>
<action path="/listEmployees"
        type="action.ListEmployeesAction"
        scope="request"
        validate="true">
  <forward name="listing"
          path="/EmployeeListing.jsp">
  </forward>
</action>
<action path="/updateEmployee"
        type="action.UpdateEmployeeAction"
        name="employeeForm"
        attribute="employeeForm"
        scope="request"
        input="/EmployeeForm.jsp"
        validate="true">
  <forward name="listing"
          path="/EmployeeListing.jsp">
  </forward>
</action>

<!-- If you have non XDoclet actions, define them in a file
called struts-actions.xml and place it in your merge directory. -->
</action-mappings>

</struts-config>

```

Listing 13.15

Struts configuration file.

The build script for this example has the following target to generate the Struts configuration:

```

<target name="actions">
  <delete file="${WEBINF}/struts-config.xml" />

  <taskdef name="webdoclet"
          classname="xdoclet.web.WebDocletTask"
          classpath="${xdocpath}"
  />
  <echo message="${xdocpath}" />

  <webdoclet
    sourcepath="${src}"
    destdir="${dest}">

    <classpath refid="cpath"/>

    <fileset dir="${src}">
      <include name="**/*.java" />
    </fileset>

    <strutsconfigxml destdir="${WEBINF}" />

  </webdoclet>

```

</target>

This code is similar to the targets for generating Web application deployment descriptors, except it uses the `strutsconfigxml` subtask to generate the Struts configuration file.

Conclusion

Struts is an MVC framework built on top of JSP, servlets, and taglibs. Struts is a growing, evolving project. It would be impossible to cover all of Struts in one chapter, but we did examine the fundamentals. We did not cover the Struts template mechanism or its new validation framework. The book *Mastering Jakarta Struts*, by James Goodwill, covers those topics in detail.

This chapter started out with a brief description of Struts. Then, we presented a common Model 1-style JSP application, which we later compared to a Struts/MVC version. With Struts we were able to get our Java code out of our JSPs and create a flexible, easily maintained version of our application.