```
    <title>Edit Employee Page</title>
    <link rel="stylesheet" type="text/css" href="styles.css">
  </head>
  <body>
    <%
    String employeeId = request.getParameter("employeeId");
    Employee emp = null;
    emp= EmployeeData.getEmployee(employeeId);
    %>
    <form action="/HibernateApplication/EditEmployeeServlet">
    <table width="500" border="0">
    <tr>
          <td>Employee Id</td>
          <td> <%=emp.getEmployeeId() %> <input type = "hidden" name ="employeeId"
          value=<%=emp.getEmployeeId() %>></td>
    </tr>
    <tr>
          <td>Employee Name</td>
          <td><input type="text" name="name" value=<%=emp.getName()%>></td>
    </tr>
    <tr>
          <td>Employee Age</td>
          <td><input type="text" name="age" value=<%=emp.getAge() %>></td>
    </tr>
    <tr>
          <td>Employee Salary</td>
  <td><input type="text" name="salary" value=<%=emp.getSalary() %>></td>
    </tr>
    <tr>
          <td> <input type="submit" name ="submit" value="submit"> </td>
    </tr>
    </table>
  </form>
  </body>
  </html>
```

Creating the EmployeeList.jsp File

The EmployeeList JSP page displays the Employee List Page screen that is used to display a list of the employees working in a company. This JSP page contains three hyperlinks that are used to add, delete, and update the details of an employee. The code for the EmployeeList JSP page is shown in Listing 15.11 (you can find the EmployeeList.jsp file on CD in the `code\JavaEE\Chapter15\HibernateApplication` folder):

Listing 15.11: Showing the **Code for the** EmployeeList.jsp File

```
<%@ page import="java.util.ArrayList, com.kogent.hibernate.Employee" %>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
<head>
  <title>Employee List Page</title>
  <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <table width="700"
  border="0" cellspacing="0" cellpadding="0">
  <tr align="left">
        <th>Employee Id</th>
        <th>Name</th>
        <th>Age</th>
        <th>Salary</th>
  </tr>
  <!-- iterate over the results of the query -->
<% ArrayList employees=((ArrayList) session.getAttribute("employees"));
if(employees!= null && employees.size()>0)
{
  for(int i=0;i<employees.size();i++)
  {
```

```
            Employee emp= (Employee) employees.get(i);
            %>
            <tr>
                    <td> <%=emp.getEmployeeId() %> </td>
                    <td> <%=emp.getName() %> </td>
                    <td> <%=emp.getAge() %> </td>
                    <td> <%=emp.getSalary() %> </td>
            <td><A href
            ="/HibernateApplication/DeleteEmployeeServlet?employeeId=<%=emp.getEmployeeId()
            %>"> Delete</A>
            <td><A href
            ="/HibernateApplication/EditEmployee.jsp?employeeId=<%=emp.getEmployeeId() %>">
            Edit</A>
            </tr>
            <%
        }
    } %>
    </table>
    <A href ="/HibernateApplication/AddEmployee.jsp">Add New Employee </A>
    </body>
    </html>
```

Creating the web.xml File

The web.xml file contains the configuration details of the HibernateApplication application and the mapping of various servlets that are used in the application. The code for the web.xml file is shown in Listing 15.12 (you can find the web.xml file on CD in the `code\JavaEE\Chapter15\HibernateApplication\WEB-INF` folder):

Listing 15.12: Showing the Configuration File web.xml of the HibernateApplication

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

<web-app version="3.0"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

<servlet>
  <servlet-name>EmployeeList</servlet-name>
  <servlet-class>com.kogent.hibernate.EmployeeList</servlet-class>

</servlet>

<servlet>
  <servlet-name>EmployeeListServlet</servlet-name>
  <servlet-class>com.kogent.hibernate.EmployeeListServlet</servlet-class>

</servlet>

<servlet>
  <servlet-name>AddEmployeeServlet</servlet-name>
  <servlet-class>com.kogent.hibernate.AddEmployeeServlet</servlet-class>

</servlet>

<servlet>
  <servlet-name>DeleteEmployeeServlet</servlet-name>
  <servlet-class>com.kogent.hibernate.DeleteEmployeeServlet</servlet-class>

</servlet>

<servlet>
  <servlet-name>EditEmployeeServlet</servlet-name>
  <servlet-class>com.kogent.hibernate.EditEmployeeServlet</servlet-class>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>EmployeeList</servlet-name>
  <url-pattern>/EmployeeList</url-pattern>

</servlet-mapping>

<servlet-mapping>
  <servlet-name>EmployeeListServlet</servlet-name>
  <url-pattern>/EmployeeListServlet</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>AddEmployeeServlet</servlet-name>
  <url-pattern>/AddEmployeeServlet</url-pattern>

</servlet-mapping>

<servlet-mapping>
  <servlet-name>DeleteEmployeeServlet</servlet-name>
  <url-pattern>/DeleteEmployeeServlet</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>EditEmployeeServlet</servlet-name>
  <url-pattern>/EditEmployeeServlet</url-pattern>

</servlet-mapping>

</web-app>
```

After developing the files related to view, logic, Hibernate configuration, and Web configuration for the HibernateApplication, let's now describe the directory structure of the application to understand where these files are located.

Exploring Directory Structure

The HibernateApplication is the root directory in which all the JSPs, servlets, and configuration files are stored. To execute the HibernateApplication application properly,,you must store the following JAR files in the \WEB-INF\lib directory:

- antlr 2.7.6.jar
- asm-attrs.jar
- asm.jar
- cglib-2.1.3.jar
- classes12.jar
- commons-collections-2.1.1.jar
- common-collections-3.1.jar
- commons-loggin-1.1.jar
- dom4j-1.6.1.jar
- ehcahe-1.2.3.jar
- hibernate3.jar
- jaas.jar
- jaxen-1.1-beta-7.jar
- jdbc2_0-stdex.jar
- jta-1.1.jar
- log4j-1.2.11.jar
- slf4j-api-1.5.8.jar
- xerces-2.6.2.jar

- xml-apis.jar

The directory structure of the HibernateApplication application is shown in Figure 15.3:

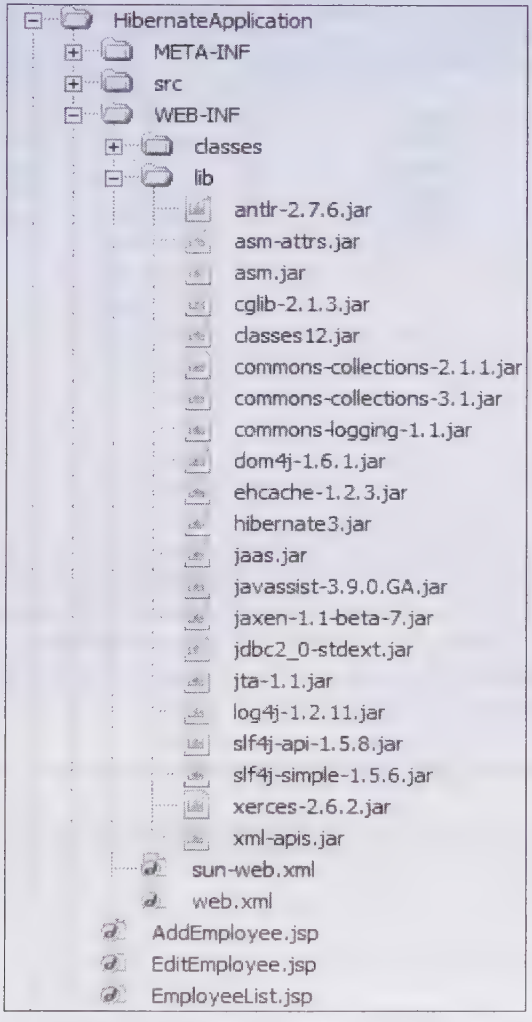


Figure 15.3: Showing the Directory Structure of HibernateApplication

Now, let's learn to run the HibernateApplication application on Glassfish application server

Running the Application

After developing the servlets, JSP pages, and configuration files, you should store them at the appropriate location in the HibernateApplication directory structure, as described in Figure 15.3. Create a Web ARchive (WAR) file as discussed in Chapter 13 and name it as HibernateApplication.war. Deploy the HibernateApplication.war file on the Glassfish application server. After successful deployment of this Hibernate Web application, perform the following steps to run the application:

1. Open the Internet Explorer Web browser and type the following URL:

   ```
   http://localhost:8080/HibernateApplication/EmployeeListServlet
   ```

   The details of employees appear, as shown in Figure 15.4:

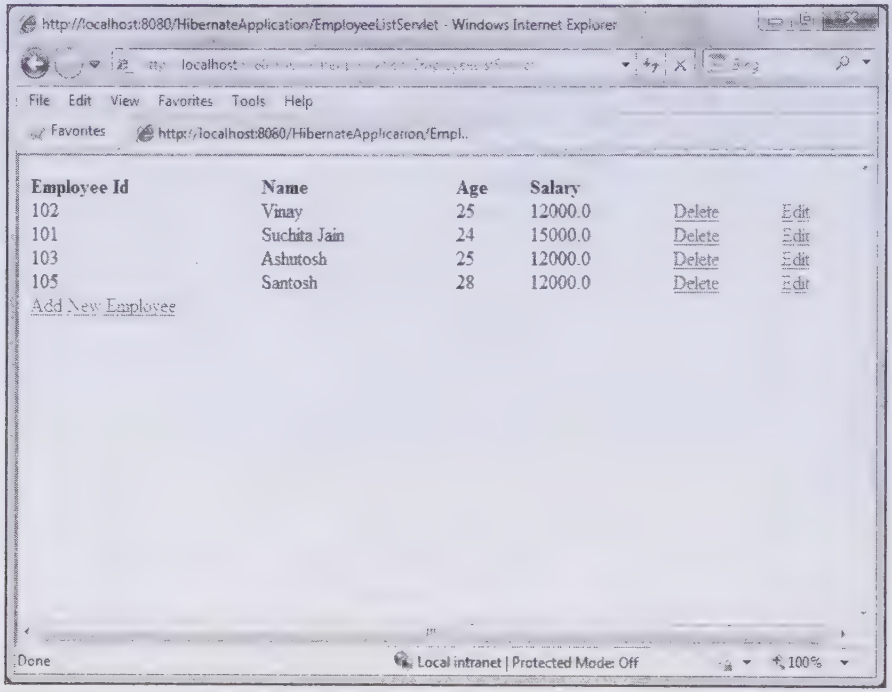


Figure 15.4: Showing the Employee List Page

Figure 15.4 shows the Add New Employee hyperlink used to add a new employee record in the database.

2. Click the Add New Employee hyperlink (Figure 15.4). The Add Employee page appears, as shown in Figure 15.5.
3. Enter Employee Id, Employee Name, Employee Age, and Employee Salary in the corresponding text fields, as shown in Figure 15.5:

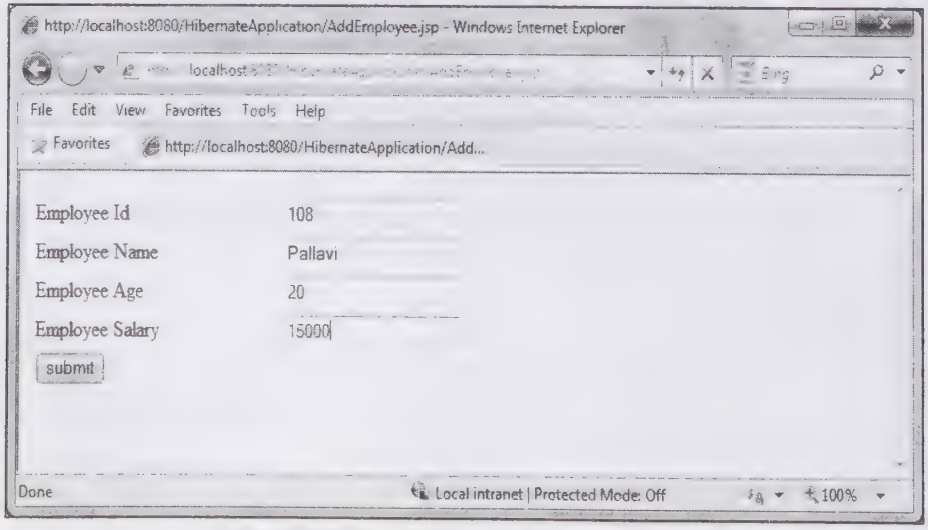


Figure 15.5: Showing the Add Employee Page

4. Click the **submit** button displayed on the Add Employee page to add a row in the EMPLOYEE table. The updated Employee List page is as shown in Figure 15.6:

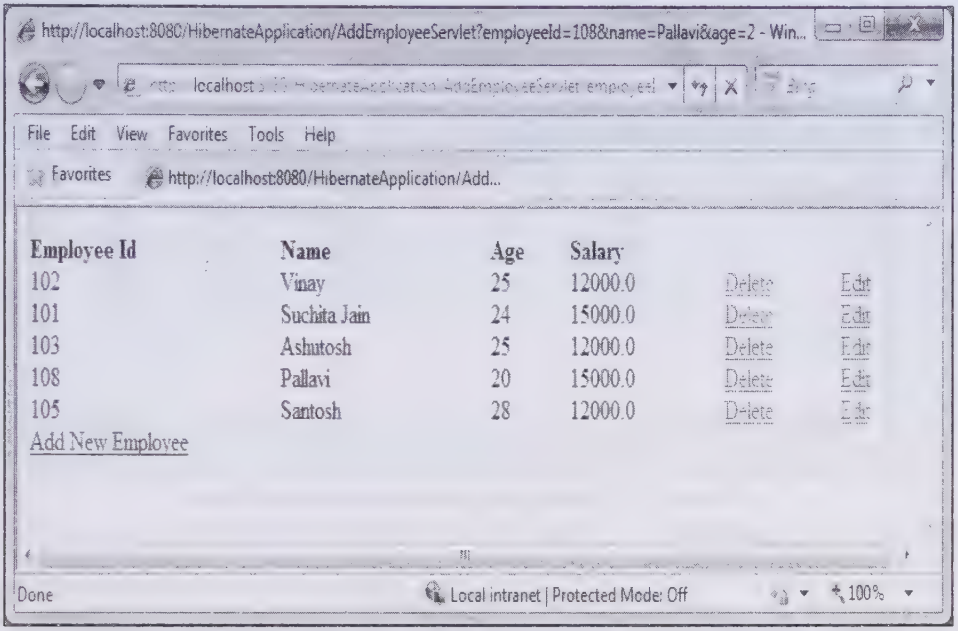


Figure 15.6: Showing the Added Employee List Page

5. Click the Edit hyperlink to edit the corresponding row. When you click the Edit hyperlink of employee whose employee id is 102, you can see the Edit Employee page, as shown in Figure 15.7:

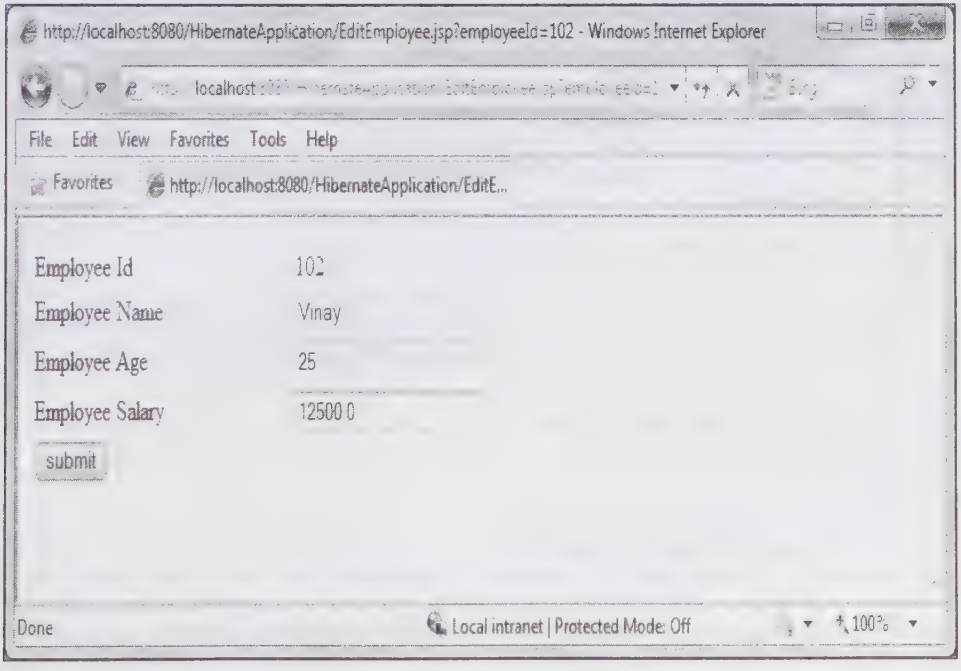


Figure 15.7: Showing the Edit Employee Page

6. Edit the employee name, age, and salary and click the submit button to update the row whose Employee Id is 102. The updated details of the Employee Id 102 are shown in Figure 15.8:

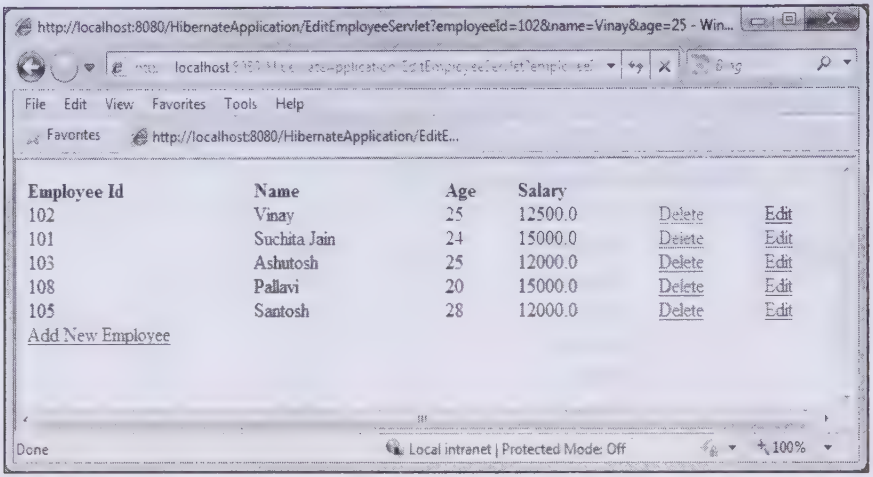


Figure 15.8: Showing the Updated Employee List Page

7. Click the Delete hyperlink shown in Figure 15.8 to delete a row whose Employee Id is 103. The record of the Employee ID 103 is deleted from the database and does not appear in the Employee List page, as shown in Figure 15.9:

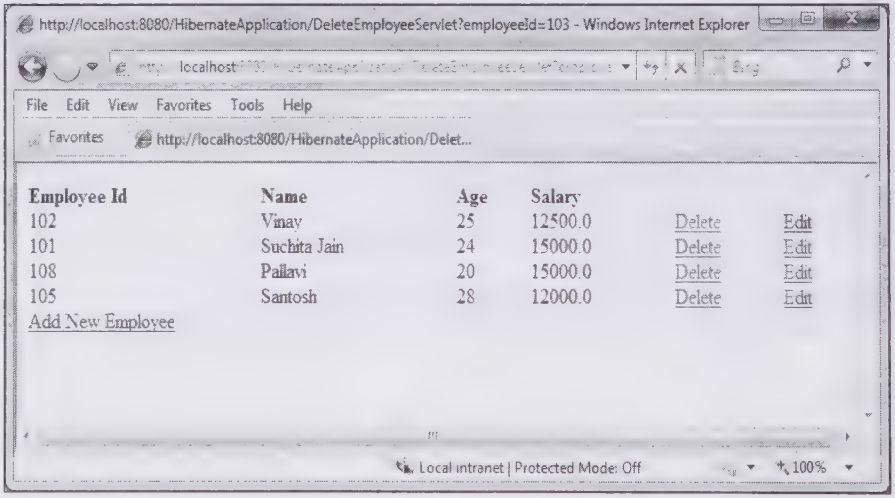


Figure 15.9: Showing the Employee List Page (After Deleting a Row)

Now, you are familiar with the Hibernate framework and can develop simple Web applications by using Java Persistence, ORM, and HQL. Let's now recapitulate the main topics discussed in this chapter.

Summary

The chapter has discussed the architecture and features of the Hibernate framework. You have also learned to retrieve data from the database table by using the O/R mapping and HQL. The chapter has also explored various clauses, such as select, from, where, and group by that can be used in HQL to query the database. You have also learned how to configure Hibernate for developing Hibernate Web applications. Towards the end, you have learned how to implement Hibernate in an application by developing a simple Hibernate Web application performing the CRUD operations.

In the next chapter, you learn the JBoss Seam framework.

Quick Revise

Q1. What is Hibernate?

Ans. Hibernate is a persistence model that provides powerful, high performance object/relational persistence, and query services. It also helps you in developing persistent classes with object-oriented features, such as association, inheritance, polymorphism, and composition.

Q2. What are the main features of Hibernate?

Ans. The main features of Hibernate are:

- Provides HQL to query database to retrieve the desired results
- Provides a way to store the Java object directly in the database table
- Provides O/R mapping in such a manner that reduces the difference between object-oriented and relational database systems

Q3. What is O/R mapping?

Ans. O/R mapping is the technique of mapping the data representation from an object-oriented system to a relational database system. The ORM feature allows you to perform various operations, such as insert, delete, update, and select, on the data stored in a database.

Q4. What is HQL?

Ans. HQL stands for Hibernate Query Language. This language is based on the relational object models and serves as an object-oriented extension to SQL.

Q5. What is the main advantage of using Hibernate than using the SQL?

Ans. The main advantage of using the Hibenate is that it allows you to map Java objects to tables in a database for persistent storage of data.

Q6. What is the difference between sorted and ordered collection in Hibernate?

Ans. A sorted collection in Hibernate is a collection that are sorted in memory using Java Comparator interface, whereas the ordered collections are the collections that are ordered in the database using the order by clause.

Q7. List the benefits of using the Hibernate framework.

Ans. The following are the benefits of using the Hibernate framework:

- Supports object-oriented programming models, such as inheritance, polymorphism, composition, abstraction, and the Java collections framework.
- Provides developers with persistence feature and a code generation library (CGLIB) that helps in extending Java classes and implementing Java interfaces at runtime environment. The changes that are made to objects associated with a transaction are automatically addressed in the database. This saves the time spent in extra coding for bytecode processing.
- Provides object-oriented query language called HQL, which is similar to SQL. HQL is an ORM query language defined in EJB 3.0. It helps in writing multi-criteria search and dynamic queries.
- Provides Object/Relational mapping for bridging the gap between object-oriented systems and relational databases.
- Enables the developer to build a Hibernate Web application very efficiently in MyEclipse by using Hibernate eclipse plug-ins that provide mapping editor, interactive query prototyping, and schema reverse engineering tool.
- Reduces the development time, as it supports object-oriented programming, such as inheritance, polymorphism, composition, and Java Collection framework.

Q8. What are the benefits of HQL?

Ans. The following are the benefits of HQL:

- Provides full support for relational operations
- Returns results as objects
- Supports polymorphic queries

- Easy to learn
- Supports advanced features
- Provides database independence

Q9. Define Session interface.

Ans. All applications need to handle the session details of a user to carry out further transactions. Sessions are created and destroyed several times in an application. Sessions are lightweight and inexpensive. Hibernate maintains a session between a connection and a transaction. It caches all the loaded objects in a session and can keep track of any changes made to these objects.

Q10. What is Transaction interface?

Ans. The Transaction interface is an optional API that resides in the `net.sf.hibernate` package. The use of this API makes Hibernate applications portable on different platforms as well as on different containers.

Subject to Pune Jurisdiction

CASH MEMO

Mob. : 9028486497
9823289154

ADVAIT BOOK SERVICES

Old & New Books Sale / Purchase

27 C, Budhawar Peth, Near Jogeshwari Temple, Pune-411002.

Name : _____

No. : 710

Date : 27/2/014

Sr. No.	Name of the Book	Qty.	Amount Rs.	Ps.
1	JSP 1:6. Black Book	1	300	
		TOTAL		

Book only

Note : Books once sold will not be taken back or exchanged

For **ADVAIT BOOK SERVICES**

16

nting

Seam

In Java EE, Enterprise JavaBeans 3.0 (EJB 3.0) is used to implement the business logic in an enterprise application. JavaServer Faces (JSF) helps in creating the user interfaces (UI) of these enterprise applications. However, developers need to provide a lot of code in an application to create UI components by using JSF. JBoss Seam (Seam) a light weight framework that helps to integrate the EJB and JSF technologies and facilitates efficient development of UI components. Consequently, while using Seam, the developers can focus more on the business logic of the enterprise applications. Seam is developed and designed by JBoss, a division of RedHat.

The Seam framework is based on a three-tier architecture that supports the Model, View, Controller (MVC) architecture. The three-tier architecture of the Seam framework consists of the presentation tier, business logic tier, and persistent tier. The presentation tier and the business logic tier integrate with JSF and EJB, respectively. The persistent tier represents the backend database support for the database access.

In this chapter, you learn about the Seam framework and its features. In addition, you learn about the Seam context, Seam components, and built-in Seam components, such as context injection, internationalization, and Mail and Java Message Service (JMS). You also learn about the configuration of the Seam components and how Business Process Management (BPM) and page flow are implemented in the Seam application. Finally, you learn how to configure the Seam framework to develop a Seam application.

Listing the Features of the Seam Framework

The Seam framework provides a simplified programming model, which is a combination of various frameworks, such as EJB and JSF. The features of the Seam framework are as follows:

- **Support for using JSF with EJB 3.0**—Provides support for integrating JSF and EJB 3.0. Integration of JSF and EJB in the Seam framework speeds up the process of creating enterprise applications, as the developers only need to focus on the business logic that is to be implemented in the application.
- **Support for AJAX**—Provides support for AJAX-based applications. Seam uses the JBoss, RichFaces, and ICEFaces frameworks, also known as JSF with AJAX frameworks, to create AJAX-based applications. Using these frameworks, developers do not need to write JavaScript code to implement the AJAX functionality. Seam makes the use of the JavaScript remoting layer to asynchronously call a component from the client side.
- **Support for jBPM**—Provides support for Java Business Process Management (jBPM), which is a workflow management system provided by JBoss. It reduces the time spent in implementing the complex workflows, collaboration, and task management features of enterprise applications.
- **Support for declarative application state management**—Provides support for declarative application state management, which overcomes the problems, such as navigation with back button, refresh button, and duplicate form submission, during the application sessions. Seam provides the conversational and business process contexts, which automate the manual technique of state management.
- **Support for Bijection**—Provides support for Bijection technique, which helps in aliasing the variables used in a context and assigning the aliased variable to attributes of the components. This automatically assembles the component tree. The Bijection technique also defines the scope of stateful components with sufficient flexibility, which was not possible with the Inversion of Control (IoC) or Dependency Injection (DI) features of JSF and EJB 3.0.
- **Support for annotations**—Provides support for the annotations feature of EJB 3.0. Earlier, while creating Seam applications, you need to provide Extensible Markup Language (XML) based configuration details in Deployment Descriptors. Seam extends the annotations feature of EJB 3.0 and uses the same with declarative state management and context demarcation annotations, which help in reducing the time required for writing JSF-based managed bean declarations. In addition, the use of annotations provides a flexible approach of configuring the components of an application without using Deployment Descriptors; thereby, saving the time that was spent earlier in creating Deployment Descriptors.
- **Support for workspace management**—Provides support for workspace management, which allows a user to switch between multiple workspaces in a single tab window. This helps in implementing transaction management at isolation level and multi-window browsing.

- **Other supports**—Provides support for other Java Persistence API (JPA) and Hibernate 3 integration, which deals with persistence objects.

After having a brief overview of the features of the Seam framework, let's now learn to use the Seam framework in an application.

Working with the Seam Framework

The Seam framework is based on three fundamental concepts: contexts, components, and annotations. Contexts and components of the Seam framework are implemented in an application as stateful EJB objects. Generally, the enterprise beans, along with instances of the Seam components, are associated with the Seam context. This provides a naming convention context to develop stateful Web applications. The @In annotation injects Seam components in a Seam application and the @Out annotation outjects the components from a Seam application.

Let's now explore these three fundamental concepts of the Seam framework in detail.

Understanding Contexts

A context is a set of namespaces and data items that are associated with the Seam components. Seam contexts are used to maintain the state of sessions in a Web application. These contexts are generated and destroyed by the Seam framework. The basic Seam contexts are as follows:

- **Stateless context**—Deals with stateless session beans. In this context, the state of a bean is active during the invocation of the bean. When a client invokes a stateless context bean, the bean instance variables remain in a session only till the time the bean is being invoked.
- **Event or request context**—Provides functionalities to manage short time events, such as remotely calling a resource and request invocations. The event or request context is the simple and most used context throughout the Seam context. This context is activated at the time of its generation and gets destroyed at the end of the event life-cycle.
- **Page context**—Allows you to maintain the state of a particular task, such as retrieving information using the dynamic list, on the Web page. You can initialize the state of a page context in the event listener, which is defined in the web.xml Deployment Descriptor, and then access it from linked events. This context is specifically used in case of dynamic Web pages.
- **Conversation context**—Holds the state of a client that supports multiple conversations in multiple windows. The conversation context is one of the most important contexts in the Seam framework. A conversation may consist of several user interactions, requests, responses, and database transactions in a Web application. One of the most important tasks of conversation context is to ensure that the states of the different conversations do not collide and cause errors. For example, using the Airlines website involves multiple conversations, such as selecting the flight, hotel, and car. If a user performs these tasks simultaneously, the respective conversations might collide. To overcome this problem, the conversation context holds the state for the current task, while the other tasks are running in the background.

NOTE

Conversation may also be nested, which means that one large conversation can contain multiple smaller conversations.

- **Session context**—Holds the state of the HyperText Transfer Protocol (HTTP) session throughout the scope of a Seam application. The session context is used as a Web request, which is used to share the global application information across multiple conversations. In the session context, the stateful session bean is used to maintain the state of several conversations throughout the Seam application.
- **Business process context**—Holds the state of the business processes involved in a Seam application. The state is managed, created, and destroyed by the Seam's jBPM. The business process holds multiple interactions with multiple users, so the state involved in a business process is shared among multiple users. The state of the current task defined by the business process instance and the life cycle of the business process is managed by the jBPM Process Definition Language (jPDL), so there are no special annotations for business process demarcation.

- **Application context**—Holds static information, such as configuration of data and data models of an application. The application context is similar to the servlet context.

The variables defined in a Seam context are known as context variables. In Seam, a developer can bind any value to a context variable, similar to defining the session and request attributes in servlets. Within a Seam framework, the Seam component variables are accessed by using the context variable names and the context class. This context class provides the component instance that can be retrieved from the Context interface, as shown in the following code snippet:

```
User user = (User) Contexts.getSessionContext().get("user"); //Method for accessing the
  value of the context variable.

Contexts.getSessionContext().set("user", user); // Method for setting the value of context
  variable.
```

The context variable name can be changed or set according to the user's choice, as shown in the preceding code snippet. The components from the application context can be accessed by injecting a component using the `@In` annotation, and can be outjected to the context by using the `@Out` annotation. In the preceding code snippet, `user` is the context variable used in the context interface. The value for the `user` context variable can be retrieved by using the `Contexts.getSessionContext().get("user")` method; whereas, the value for the variable can be set by using the `Contexts.getSessionContext().set("user")` method.

However, when you need to obtain the components, injection is used to get components from a context, and outjection is used to outject the component instances into a context.

NOTE

Seam components instances are obtained from a defined scope. All stateful scopes are searched in the priority order, which is as follows:

1. Event context

2. Page context

3, Conversation context

4. Session context

5. Business process context

6. Application context

You can perform priority search by calling the `Contexts.lookupInStatefulContexts()` method.

Working with Seam Components

The Seam components are similar to Plain Old Java Object (POJO) components that integrate JavaBeans or EJBs and JSF to implement the business and presentation logic. The Seam framework supports the following types of beans:

- JavaBeans
- EJB 3 session beans
- EJB 3 stateless session beans
- EJB 3 stateful session beans
- EJB 3 entity beans
- EJB 3 message driven beans

NOTE

You can learn more about EJB 3 in Chapter 13, Working with EJB3.1 and Chapter 14, Implementing Entities and Java Persistence API.

The Seam components must be intercepted before they are used in the Seam framework. For example, to access the session beans, you must register an EJB interceptor in a class through the `@Interceptors` annotation. The interceptors for the session bean component can be defined, as shown in the following code snippet:

```
@Stateless
@Interceptors (SeamInterceptor.class)
public class LoginAction implements Login {
  // codes for login module
}
```

You can also define the interceptors in the ejb-jar.xml file. The representation of the interceptors within the ejb-jar.xml file is shown in the following code snippet:

```
<interceptors>
<interceptor>
  <interceptor-class> org.jboss.Seam.ejb.SeamInterceptor </interceptor-class>
</interceptor>
</interceptors>
<assembly-descriptor>
  <interceptor-binding>
        <ejb-name>*</ejb-name>
<interceptor-class> org.jboss.Seam.ejb.SeamInterceptor </interceptor-class>
</interceptor-binding>
</assembly-descriptor>
```

In the Seam framework, all Seam components must be assigned a name. The @Name annotation is used to assign a name to a component. The following code snippet shows how to assign a name to a component:

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
...
}
```

In the preceding code snippet, we have created the `loginAction` Seam component for the `LoginAction` class that implements the `Login` interface.

Apart from creating a Seam component, you can also use the built-in Seam components in a Seam application. Let's discuss about the built-in Seam components and how to configure a Seam component in the following subsections.

Exploring Built-in Seam Components

The Seam framework provides some built-in components, such as context injection and internationalization, which help to access entity beans and stateful session beans in a Seam application. These built-in components are used to customize the basic functionalities of Seam applications and can be defined at runtime. You can define built-in components in the components.xml file by overriding the default properties, such as `localeSelector.localeString`, `localeSelector.language`, and `localeSelector.cookieEnabled`.

To use the built-in components, you first need to replace the existing Seam components with the built-in components. The built-in components can be replaced by specifying the name of the corresponding Seam component, such as stateless, stateful, and entity, in a class by using the `@Name` annotation. A built-in Seam component must be qualified with a name. However, you cannot replace some existing Seam components as they are aliased to unqualified names, by default. This implies that you cannot assign a name to these Seam components. In such cases, you can use the `auto-create` property to create an instance of Seam built-in component corresponding to a Seam component that cannot be replaced. You can set the `auto-create` property as true in the components.xml file.

The built-in components are defined in the Seam API of the `org.jboss.seam.core` package. Some important built-in Seam components provided by the Seam framework are listed as follows:

- Context injection component
- Internationalization component
- Mail and Java Messaging Service (JMS) related components
- Infrastructural component
- Special components

These Seam built-in components can be used in Seam applications to extend the security, internationalization, and message services functionalities.

The Context Injection Component

The context injection Seam built-in component is used to inject various contextual objects, such as `sessionContext` and `applicationContext`, in a class. The following syntax is used to inject and instantiate a Seam session context object in a Seam application:

```
@In private Context <Name of the context>;
```

The preceding syntax allows you to specify the `org.jboss.seam.core.contexts` class to access a Seam context object. For example, the `org.jboss.seam.core.contexts.sessionContext[loginUser]` object is used to access a user object in a session context.

The Internationalization Components

The internationalization built-in component provides an easy way to build internationalized Seam applications with support for multiple languages.

The following components are used to implement internationalization in Seam applications:

- `org.jboss.seam.international.locale`—Serves as a built-in Seam component used to handle Locales, such as geographical, political, or cultural regions. You can use the `org.jboss.seam.international.localeSelector` object to implement Locale-specific functionalities at either configuration level or runtime. The select() method is used to set a specific Locale. You can use the following properties with the `localeSelector object`:
  - `localeSelector.localeString`—Provides string representation of a Locale
  - `localeSelector.language`—Represents the language of the specified Locale
  - `localeSelector.country`—Denotes a particular country for the specified Locale
  - `localeSelector.cookieEnabled`—Helps to select a Locale that should be persisted through a cookie
- `org.jboss.seam.international.timezone`—Serves as a built-in Seam component used to set time zone for the current session. You can use the `org.jboss.seam.international.timezoneSelector` component to set the time zone at either configuration level or runtime. The following method and properties are used with the `timezoneSelector` component:
  - `select()`—Helps to select a specified Locale
  - `timezoneSelector.timeZoneId`—Specifies the time zone, represented as a String value
  - `timezoneSelector.cookieEnabled`: Helps to select the time zone that should persist throughout a cookie
- `org.jboss.seam.core.resourceBundle`—Serves as a built-in Seam component that is used to search the resource bundles associated with the current session. The Seam resource bundle performs a detailed search to locate the required information for internationalization in a list of Java resource bundles.

NOTE

You can refer to Appendix H, Implementing Internationalization for more information on internationalization.

The Mail and JMS Related Components

The Seam framework provides mail related built-in component, `MailSession`, which is used to send and receive email messages. The `org.jboss.seam.mail.MailSession` component is a manager component with session scope that can be looked up either by Java Naming and Directory Interface (JNDI) or can be created by configuring a host. A manager component is used to manage the life cycle of a component. Any component used with the `@Unwrap` method can be a manager component.

The properties of the `org.jboss.seam.mail.mailSession` component are as follows:

- **org.jboss.seam.mail.mailSession.host**—Specifies the hostname of the Simple Mail Transfer Protocol (SMTP) server.

- **org.jboss.seam.mail.mailSession.port**—Denotes the port number of the SMTP server.
- **org.jboss.seam.mail.mailSession.username**—Denotes the username used to connect to the SMTP server.
- **org.jboss.seam.mail.mailSession.password**—Specifies the password used to connect to the SMTP server.
- **org.jboss.seam.mail.mailSession.debug**—Helps in debugging JavaMail.
- **org.jboss.seam.mail.mailSession.ssl**—Helps to enable Secure Socket Layer (SSL) connection to SMTP. The default port number for this property is 465.
- **org.jboss.seam.mail.mailSession.sessionJndiName**—Helps to specify the name of the current session, represented by the javax.mail.Session object, which is bound to JNDI. If this property is passed to the MailSession component, all the properties are ignored.

The JMS built-in components are used with the `TopicPublishers` and `QueueSenders` managed properties. The `org.jboss.seam.jms.queueSession` property is a manager component of the `JMS QueueSession` component and `org.jboss.seam.jms.topicSession` is a manager component of the `JMS TopicSession` component.

The Infrastructural Component

The infrastructural built-in component provides an environment to integrate Seam annotation, EJB 3, JSF, and Hibernate to develop JBoss Seam applications. However, this component is not installed by default; you need to set the `install` property to `true` in the components.xml file to install this component.

The properties of the infrastructural component are as follows:

- **org.jboss.seam.core.init.jndiPattern**—Helps to look up session beans by using the JNDI pattern.
- **org.jboss.seam.core.init.debug**: Helps to enable the Seam debug mode.
- **org.jboss.seam.core.init.clientSideConversations**—Helps to save the conversation context variables. When this property is set to true, the conversation context variables are saved at the client side instead of an HttpSession.
- **org.jboss.seam.core.init.userTransactionName**—Helps to look up Java Transaction API (JTA) UserTransaction objects with the help of JNDI name.
- **org.jboss.seam.core.manager.conversationTimeout**—Specifies the conversation context timeout, in milliseconds.
- **org.jboss.seam.core.manager.concurrentRequestTimeout**—Specifies the maximum wait time of a thread. This property also attempts to lock a long-running conversation context after the specified maximum wait time.
- **org.jboss.seam.core.manager.conversationIsLongRunningParameter**—Specifies whether or not a conversation is long-running.

The Special Components

Some special Seam components are manager components that manage the entity manager with extended persistence context. For example, the following code snippet installs and configures the two Seam components in the components.xml file, which serve as the entity managers:

```
<component name="loginDatabase"
   class="org.jboss.Seam.persistence.ManagedPersistenceContext">
<property name="persistenceUnitJndiName">java:/comp/emf/loginPersistence</property>
</component>

   <component name="employeeDatabase"
   class="org.jboss.Seam.persistence.ManagedPersistenceContext">
<property name="persistenceUnitJndiName">java:/comp/emf/employeePersistence</property>
</component>
```

In the preceding code snippet, `loginDatabase` and `employeeDatabase` are special Seam components.

Now, let's discuss how to configure Seam components.

Configuring Seam Components

The Seam components are configured by using XML files, such as web.xml and components.xml, available in the Seam framework. You also need to set the configuration details in the property file. Configuring Seam components helps a developer to isolate deployment-specific information from Java code, and provides reusability. The web.xml and component.xml files are also used to configure the built-in Seam components.

You can configure the Seam components by using:

- Property settings
- The components.xml file
- Sliced configuration files
- XML namespaces

Let's discuss these in detail.

Using Property Settings

The properties of the Seam components can be configured by using the servlet context parameters or the Seam properties file, named `seam.properties`. The `seam.properties` file is available in the root of the `classpath`. To understand how to configure Seam components, let's look at an example. Suppose we have a property, `com.jboss.myapp.setting` and a setter method, `setLocale()` in the Seam framework. The property name for the `setLocale()` method can be specified as `com.jboss.myapp.setting.locale` in the `seam.properties` file or in the servlet context parameter. The Seam framework sets the value of the locale attribute while instantiating the Seam component.

To configure Seam in an application, you need to provide the property settings in the web.xml or seam.properties file. For example, to store the state of conversation timeout, you must provide a value for the `org.jboss.seam.core.manager.conversationTimeout` property in the `web.xml` or `seam.properties` file.

Using the components.xml File

Using the components.xml file to configure Seam components is a more powerful approach than setting a property in the web.xml or seam.properties file. The components.xml file has the following advantages over the seam.properties file:

- Configures the installed component automatically. This includes both the built-in Seam components as well as other Seam components, such as stateless session bean and entity bean, which have been annotated with the `@Name` annotation.
- Helps to configure Seam components that have the `@Name` annotation but have not been installed automatically. This is because the `@Install` annotation indicates whether or not the component is already installed.

The components.xml file can be stored either in the WEB-INF directory of a WAR, in the META-INF directory of a jar, or within any directory of a jar that contains the classes with the `@Name` annotation. In general, Seam components are installed when the deployment scanner finds a class annotated with the `@Name` annotation stored in an archive with the `seam.properties` file or the `META-INF/components.xml` file.

The annotations used in the classes may be overridden. The `components.xml` files handle these special cases of overriding annotations within the `seam.properties` file.

The following code snippet shows the code to be provided in the `components.xml` file to install jBPM in an application:

```
<components xmlns=http://jboss.com/products/Seam/components"
   xmlns:xsi=http://www.w3.org/2001/XMLschema-instance
   xmlns:bpm=http://jboss.com/products/Seam/bpm >
   <bpm:jBPM/>
</components>
```

jBPM can also be installed by making some changes in the `components.xml` file, as shown in the following code snippet:

```
<components>
  <component class="org.jboss.Seam.bpm.jbpm">
</components>
```

The Seam managed persistence context can be installed by using EJB 3 as well. In this case, you need a database and the persistence context unit used for that database. You can include the scope of the persistence context in an application while configuring the persistence context, as shown in the following code snippet:

```
<components xmlns=http://jboss.com/products/Seam/components
  Xmlns:persistence=http://jboss.com/products/Seam/persistence >
  <persistence:managed-persistence-context name="StudentEntityManagerFactory"
        scope="session"
        auto-create="true"
  persistent -unit-jndi-name="java:/studentEntityManagerFactory"/>
  <persistent:managed-persistence-context name="employeeDatabase"
        scope="session"
        auto-create="true"
  persistence-unit-jndi-name="java:/employeeEntityManagerFactory"/>
</components>
```

While configuring infrastructure components, you need to explicitly specify the `create` property as true in the `components.xml` file. To avoid this, you can create an instance of the component by using the `auto-create` method. Generally, `auto-create` is used along with the `@In` annotation in the `components.xml` file.

Alternatively, you can also install the Seam persistence context by making changes in the `components.xml` file, as shown in the following code snippet:

```
<components>
<component name="studentDatabase">
  Class="org.jboss.Seam.persistenceManagedPersistenceContext"
  Scope="session"
  auto-create="true">
<property name="PersistentUnitJndiName">java:/customerEntityManageFactory
</property>
</component>
<component name="employeeDatabase"
  Class="org.jboss.Seam.persistence.ManagedPersistenceContext"
  Scope="session"
  auto-create="true">
<property name="PersistentUnitJndiName">java:/employeeEntityManagerFactory
</property>
</component>
</components>
```

Using the Sliced Configuration Files

Configuring multiple components using XML in the components.xml file is a cumbersome task. To avoid this, you can divide the information specified in the components.xml file into small files. Let's consider that we have a class named `com.JavaEE.Java` in an application. Seam specifies the configuration information for this class into a resource file named com/JavaEE/Java.component.xml. The root of the resource file can be either the <component> element or the <components> element. The following code snippet shows the definition of components in the `Java.component.xml` file:

```
<components>
  <component class="com.JavaEE.Java" name="Java">
        <property name="name">#{user.name}</property>
  </component>
  <factory name="message" value="#{Java.message}"/>
</components>
```

In the preceding code snippet, the Java component is configured by providing the `com.JavaEE.Java` class name. However, in the following code snippet, the class name has not been specified to configure the Java component in the `Java.component.xml` file:

```
<component name="Java">
  <property name="name">#{user.name}</property>
```

```
</component>
```

In the preceding code snippet, the class name is implied by the file in which the component definition appears.

Using XML Namespaces

You can also declare the Seam components with or without the XML namespaces. The following code snippet shows the declaration of a component without using namespaces:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns=http://jboss.com/products/Seam/components
xsi:schemaLocation=http://jboss.com/products/Seam/components
http://jboss.com/products/Seam/components-2.0.xsd">
  <component class="org.jboss.Seam.core.init">
        <property name="debug">true</property>
        <property name="jndiPattern">@jndiPattern@</property>
</component>
</components>
```

The following code snippet shows how to configure components by using namespaces:

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/Seam/components"
  xmlns:core="http://jboss.com/products/Seam/core"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://jboss.com/products/Seam/core
http://jboss.com/products/Seam/core-2.0.xsd
http://jboss.com/products/Seam/components http://jboss.com/products/Seam/components-
2.0.xsd">
  <core:init debug="true" jndi-pattern="@jndiPattern@"/>
</components>
```

XML namespaces provide detailed information about each component and attribute in the file you want to configure. The use of the namespaced elements makes the components.xml file much simpler. The Seam framework supports the mixing of both the models (with or without the usage of the namespaces). It also allows the use of the <component> generic component declaration or quick declarations of the namespaces for the components. You can also associate a Java package with an XML namespace by using the `@Namespace` annotation.

Table 16.1 shows some namespaces provided by the Seam framework:

Table 16.1: Namespaces of the Seam framework

Namespaces	Namespace representation
components	Helps to handle Seam components. Its namespace representation is `http://jboss.com/products/ Seam/components`.
core	Helps to handle core classes of the Seam framework, such as `BusinessProcess`, `Ejb`, and `Pageflow`. Its namespace representation is `http://jboss.com/ products/Seam/core`.
drools	Helps to handle Seam drools classes, such as `DrollsActionHandler`, `DrollsDecisionHandler`, and `RuleBase`. You can define its namespace as `http://jboss.com/products/Seam/drools`.
security	Helps to handle security features of Seam applications. It can be defined as `@Namespace(value="http://jboss.com/products/seam/security", prefix="org.jboss.seam.security")` in your Seam application.
mail	Helps to handle email functionalities, such as receiving and sending emails by using Seam framework. Its namespace representation is `http://jboss.com/ products/Seam/mail`.
web	Helps to handle Web classes, such as `CharacterEncodingFilter`, `ContextFilter`, and `MultipartRequest`, to handle client requests and responses. It can be defined as

Table 16.1: Namespaces of the Seam framework	
Namespaces	**Namespace representation**
	`@Namespace(value="http://jboss.com/products/seam/web", prefix="org.jboss.seam.web")` in Seam applications.
pdf	Helps to handle the Portable Document Format (PDF) files. Its namespace representation is `http://jboss.com/products/Seam/pdf`.
Spring	Helps to add Spring framework functionality in an application. Its namespace representation is `http://jboss.com/products/Seam/spring`.

Let's now move ahead and explore the annotations of the Seam framework.

Using Annotations

The Seam framework supports annotations, allowing you to directly configure a Seam component instead of creating an instance of the component. This simplifies the code used for a Seam component in a class. Seam introduces the bijection mechanism for injecting and outjecting Seam components. As already discussed, Seam is an integration of the JSF and EJB frameworks; therefore, it supports the annotations defined in the EJB 3.0 specification. The annotations for Seam component life cycle methods are also discussed in this subsection as these annotations allow a Seam component to interact with its life cycle events. The `@Logger` annotation has also simplified the code for providing a simple log message. The Seam framework provides the annotation support for:

- Bijection mechanism
- Life cycle callback methods
- Conditional installation
- Logging

Now let's discuss these in detail.

Understanding the Bijection Mechanism

DI allows you to separate the construction and implementation of an object. It allows a component to obtain a reference of another component by injecting a setter method or an instance variable in a class. The injection mechanism occurs only when the component is created and the reference of the component is changed during the lifetime of the component instance.

All the instances of a stateless component are interchangeable in a Seam application. Therefore, DI is not very useful in this scenario. As a result, the bijection mechanism has been introduced in the Seam framework. The usage of DI in the Seam framework is often referred as bijection, because the injection is performed in a two-way format, injection and outjection. For example, a component named A creates another component named B, which can outject to another component, C, for use at a later point of time. This process in the Seam framework is known as bijection.

The bijection mechanism is used to assemble the Seam components from different contexts, such as event, page session, and business process. The values of context variables are injected to the Seam components and can again be outjected from the Seam components. Consequently, bijection occurs each time a Seam component is invoked. In other words, the bijection mechanism is contextual, bidirectional, and dynamic in the Seam framework.

The Seam components can be injected and outjected by using the `@In` and the `@Out` annotations. The `@In` annotation specifies the Seam component that needs to be injected. The following code snippet shows the use of the `@In` annotation with a Seam component:

```
@Name ("HelloUser")
@Stateless
public class HelloUser implements User
{
  @In Username name;
```

```
    …
}
```

The `@In` annotation can also be used with setter methods, such as the `setUserName()` method of a Seam component. The following code snippet shows the use of `@In` annotation with a setter method:

```
@Name ("HelloUser")
@Stateless
public class HelloUser implements User
{
  Username name;
  @In
  public void setUsername (Username name)
  {
        this.name=name;
  }
  …
}
```

Similarly, you can use the `@Out` annotation for outjection in a Seam component. The `@Out` annotation can also be used with the getter method. The following code snippet shows the use of the `@Out` annotation:

```
@Name ("HelloUser")
@Stateless
public class HelloUser implements User
{
  @In Username name;
  …
}

// Using @Out annotation in the getter method

@Name ("HelloUser")
@Stateless
public class HelloUser implements User
{
  Username name;
  @In
  public void setUsername (Username name)
  {
        this.name=name;
  }
  @Out
  public Username getUserName() {
        return name;
  }
  …
}
```

Understanding Life Cycle Callback Methods

The life cycle of the Seam components is managed with the help of various annotated methods, such as @Create, @PreDestroy, @PostConstruct, and @Destroy. These methods are known as life cycle callback methods. When the first object of a class is created, the life cycle callback methods get attached to the object and are available throughout the session of the object. A Seam component, such as session bean or entity bean, supports all common EJB 3 life cycle callback methods, such as `@PostConstruct` and `@PreDestroy`. In addition, the Seam life cycle callback methods are also supported by JavaBeans.

The Seam framework also supports two additional life cycle callback methods, `@Create` and `@Destroy`, which are equivalent to `@PostConstruct` and `@PreDestroy`. The `@Create` callback method is defined whenever the Seam framework instantiates a Seam component and the @Destroy callback method is called whenever the Seam

framework destroys a Seam component. In the Seam framework, the `@Startup` callback method is used to instantiate a Seam component when the life cycle of a Seam application starts.

Understanding Conditional Installation

When you install Seam, some of its components are installed by default. However, you can install other components by using the `@Install` annotation in the Seam framework. In other words, the `@Install` annotation allows you to control the conditional installation of Seam components required to deploy Web applications. When you use the `@install` annotation, you need to specify the precedence value of the corresponding Seam component. The precedence of a component is a numeric number, which specifies the component that needs to be installed if multiple classes with the same component name exist in the classpath of the Seam application.

The components in the Seam framework are selected according to the pre-defined precedence values, as shown in Table 16.2:

Table 16.2: Pre-defined Precedence Value of Seam Components

Precedence Value	Description
BUILT_IN	Specifies the lowest precedence value among all the precedence values available in the Seam framework.
FRAMEWORK	Specifies the component defined by a third party framework that overrides the built-in components. This precedence value can be overridden by the application components.
APPLICATION	Specifies the default component in the Seam framework. This precedence value is applicable for most application frameworks, such as JSF.
DEPLOYMENT	Specifies the precedence value for application components used for deployment purpose.
MOCK	Specifies the precedence value used for testing Seam components.

In case you unknowingly begin to install an already installed component, the `@Install` annotation automatically prevents the installation.

Implementing Logging

Earlier, you had to write multiple lines of code for implementing even simple functionalities in an application. For example, more lines of code need to be provided for implementing logging than for implementing the business logic. The following code snippet shows the code that need to be written to implement logging in an application without using Seam:

```
private static final Log log = LogFactory.getLog(CreateOrderAction.class);
public Order createOrder(User user, Product product, int quantity) {
  if (log.isDebugEnabled()) {
        log.debug("Creating new order for user: " + user.username() +
        " product: " + product.name()
        + " quantity: " + quantity);
  }
  return new Order(user, product, quantity);
}
```

In the preceding code snippet, the `log.isdebugEnabled()` method is verified, because string concatenation is preformed inside the `debug()` method. To overcome this problem, Seam provides a logging API that simplifies the code for implementing logging. The following code snippet shows the use of the logging API:

```
@Logger private Log log;
public Order createOrder(User user, Product product, int quantity) {
log.debug("Creating new order for user: #0 product: #1 quantity: #2", user.username(),
product.name(), return new Order(user, product, quantity);
}
```

In the preceding code snippet, the `log` variable is not static, but it should be declared static for entity bean components. Moreover, in the preceding code snippet, we do not need to provide the `log.isDebugEnabled()` method as we do not need to explicitly specify the log category when the Seam component is injected.

NOTE

If log4.jar is set in the classpath environment variable, Seam uses it for logging; else it uses JDK logging.

Now, let's discuss business process management (BPM) and page flow of Seam framework, which allow you to create business processes.

Implementing BPM and Page Flow in Seam

A business process is a well defined group of tasks that need to be performed by the users or applications. jBPM is integrated with the Seam application to create and implement business processes. jBPM is a business process management engine for the Java environments. BPM is the process to manage the set of tasks in a business process. jBPM also supports the Java Standard Edition (SE) and the Enterprise Edition (EE) environments to create and implement business processes.

jBPM provides well-defined rules specifying who can perform a task, and when it should be performed. It represents the tasks of a business process as nodes of a Seam application. The nodes represent the wait states, decisions, tasks, and Web pages. jBPM is introduced in the Seam framework for the following two reasons:

- Defining page flow by using jPDL, which allows you to define a single user interaction and a single jPDL process for single conversation. Therefore, a Seam conversation is referred to be a short running interaction for a single user.
- Providing facilities to manage multiple user tasks and conversations.

NOTE

jPDL is an extendable language that is used to design long range Seam applications and define the page flow of the Seam applications.

The Seam framework stores the associated business process tasks in the BUSINESS_PROCESS context. The states of the tasks are persisted by the jBPM variables. The business process definition of the Seam framework looks similar to the page flow definition, except the <page> nodes. In case of business process definition, we use the <task-node> as the node of a task, as shown in the following code snippet:

```
<process-definition name="todo">

  <start-state name="start">
        <transition to="todo"/>
  </start-state>

  <task-node name="todo">
        <task name="todo" description="#{todoList.description}">
              <assignment view-id="#{view.id}"/>
        </task>
        <transition to="done"/>
  </task-node>

  <end-state name="done"/>

</process-definition>
```

You can use the jPDL business process definition along with the jPDL pageflow definition in the same component. The similarity between these two definitions is that a single <task> element in a business process corresponds to the page flow in the <pageflow-definition> tag.

Depending on the work flow and task management of the Seam application, the page flow can be categorized into the following three different groups:

- Stateless navigation model
- Stateful navigation model
- jPDL Pageflows

Let's discuss these in detail.

Stateless Navigation Model

The stateless navigation model defines a mapping of a set of logical outcome of events with the resources of a Seam application. The logical outcomes of the events are represented in a view page. The navigation rules for a Web page depend on the source of the event, i.e. from where the event has been generated. The stateless navigation model can be represented by using the JSF navigation rules or Seam navigation rules. For example, suppose we have a page named guess.jsp in a Seam application. In case of a successful event, the view page is forwarded to the success.jsp page. In case of a failure event, the failure.jsp file is displayed. The JSF navigation rules for this example are shown in the following code snippet:

```
<navigation-rule>
  < from-view-id>/Guess.jsp</from-view-id >
  <navigation-case>
        <from-outcome>guess</from-outcome>
        <to-view-id>/Guess.jsp</to-view-id>
        <redirect/>
  </navigation-case>
  <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/success.jsp</to-view-id>
        <redirect/>
  </navigation-case>
  <navigation-case>
        <from-outcome>failure</from-outcome>
        <to-view-id>/ failure.jsp</to-view-id>
        <redirect/>
  </navigation-case>
</navigation-rule>
```

The stateless navigation model can be represented by using the Seam navigational rules as well, as shown in the following code snippet:

```
<page view-id="/Guess.jsp">
  <navigation>
        <rule if-outcome="guess">
              <redirect view-id="/Guess.jsp"/>
        </rule>
        <rule if-outcome="success">
              <redirect view-id="/success.jsp"/>
        </rule>
        <rule if-outcome="failure">
              <redirect view-id="/failure.jsp"/>
        </rule>
  </navigation>
</page>
```

Stateful Navigation Model

The stateful navigation model works on the principle of jPDL. This model defines the flow of the pages of a Seam application by using a set of defined elements, such as <start-page> and <transition>.

To implement stateful navigation, the action listener methods for a particular event are used to define the flow of multiple pages in a Seam application, as shown in the following code snippet:

```
<pageflow-definition name="numberGuess">
  <start-page name="viewGuess" view-id="/Guess.jsp">
    <redirect/>
```

```
            <transition name="guess" to="evaluateGuess">
                  <action expression="#{Guess.guess}" />
            </transition>
      </start-page>
      <decision name="evaluateGuess" expression="#{Guess.correctGuess}">
            <transition name="true" to="success"/>
            <transition name="false" to="evaluateRemainingGuesses"/>
      </decision>
      <decision name="evaluateRemainingGuesses" expression="#{Guess.lastGuess}">
            <transition name="true" to="failure"/>
            <transition name="false" to="viewGuess"/>
      </decision>
      <page name="success" view-id="/success.jsp">
            <redirect/>
            <end-conversation />
      </page>
      <page name="failure" view-id="/failure.jsp">
            <redirect/>
            <end-conversation />
      </page>
</pageflow-definition>
```

Using jPDL Pageflows

The installation of the jBPM components specifies the position of the jPDL pageflows in an application. The Seam configuration for the pageflows can be stored within the components.xml file of a Seam application. The specification of the pageflows within the components.xml file is shown in the following code snippet:

```
<core:jbpm>
  <core:pageflow-definitions>
        <value>pageflow.jpdl.xml</value> .
  </core:pageflow-definitions>
</core:jbpm>
```

In the preceding code snippet, the <core> element installs the jBPM component; whereas, the <value> element, which contains the pageflow.jpdl.xml file, specifies the jPDL based pageflow definition.

After installing the required components, you need to start the jPDL pageflows. The jPDL pageflows are initiated by providing the name of the process definitions along with the @Begin, @BeginTask, or @StartTask annotations. The following code snippet shows how to start a jPDL pageflow by using the @Begin annotation:

```
@Begin(pageflow="guess")

public void begin() { ... }
```

The pageflows can be started by using the pages.xml file as well, as shown in the following code snippet:

```
<page>
  <begin-conversation pageflow="guess"/>
</page>
```

If the pageflow is started by using the @Create annotation or during the RENDER-RESPONSE phase, you need to use the <start-page> element in the pages.xml file as the starting node of the pageflow. If the pageflow is started by the invocation of the action listener, the <start-state> element is specified as the starting node of the particular pageflow. The following code snippet demonstrates the use of the starting nodes within a pageflow:

```
<pageflow-definition name="viewEditRecord">
  <start-state name="start">
        <transition name="RecordFound" to="displayRecord"/>
        <transition name="RecordNotFound" to="notFound"/>
  </start-state>

  <page name="displayRecord" view-id="/Record.jsp">
        <transition name="edit" to="editRecord"/>
        <transition name="done" to="main"/>
```

```
    </page>
    ...

    <page name="notFound" view-id="/404.jsp">
          <end-conversation/>
    </page>

</pageflow-definition>
```

Each page node defined in the pages.xml file is used to define a state of the pageflow. The state is meant to accept input from the user. The view-id element is used in this file to represent the JSF view-id. The transition name specifies the JSF outcome, which is triggered by the action event listener.

The pageflow can be controlled by using the <decision> elements. The name of the control structure is given in the <decision> element, and the value for the <decision> element is given in the <expression> element. The decision is made by evaluating JSF Expression Language (EL) expressions within the Seam contexts.

After the successful completion of the pageflow, the associated conversation needs to be terminated by the Seam framework. The conversation can be terminated by using the `<end-conversation>` element or by specifying the `@End` annotation in the pages.xml file, as shown in the following code snippet:

```
<page name="success" view-id="/success.jsp">
  <redirect/>
  <end-conversation/>
</page>
```

You can also end a conversation by specifying the jBPM transition name in the `pages.xml` file, as shown in the following code snippet:

```
<page name="success" view-id="/success.jsp">
  <redirect/>
  <end-task transition="success"/>
</page>
```

Let's now learn how to configure the Seam framework to develop Seam applications.

Configuring JBoss Seam

Prior to the configuration of JBoss Seam, you need to download the compressed version of JBoss Seam framework and decompress its Seam API. It can be downloaded as a gun-zipped Tape ARchive (TAR) file or as a ZIP file from the http://labs.jboss.com/portal/jbossSeam/download/index.html Uniform Resource Locator (URL).

NOTE

The JBoss Seam (jboss-Seam-2.1.0.A1.zip) is used in this chapter to develop Seam applications.

A large number of configurations are possible with the Seam framework, which contains many external library files associated with it. You only need the jboss-seam-ui.jar and jboss-seam.jar files to compile the source code of a Seam application.

A Seam application generally uses JSF to create a view. You need to define the JSF support to create the view within the <servlet element> element, as shown in the following code snippet:

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.Webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>


<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.Seam</url-pattern>
</servlet-mapping>
```

In addition, you also need to specify the following code snippet in the web.xml file to support event handling:

```
<listener>
  <listener-class>org.jboss.Seam.servlet.SeamListener</listener-class>
</listener>
```

In the preceding code snippet, the listener is responsible for destroying the session and application contexts, and loading the Seam application.

NOTE

If you face a problem with conversation propagation during form submission, then the following mapping in web.xml, switches you to the client side state:

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>
```

Configuring JSF in Seam

The Seam framework allows you to configure JSF to develop view components of an application. To use JSF with Seam, you need to configure the following resources:

- Facelets
- Seam Resource Servlet
- Seam Servlet Filters

Let's discuss these configurations in detail.

Configuring Facelets

Facelets are viewhandlers that are used with JSF pages having no JSP code. The facelets do not need any Tag Library Descriptor (TLD) file or tag classes to define a UI component and is faster than using JSF. When you want to use facelets instead of JSF, the following code snippet needs to be added in the faces-config.xml file:

```
<application>
  <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
</application>
```

You need to provide the <context-param> element to call the .xhtml page in the web.xml file at the time of running the default.xhtml page, as shown in the following code snippet:

```
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.xhtml</param-value>
</context-param>
```

Configuring Seam Resource Servlet

The Seam Resource Servlet represents the servlet provided with Seam to access various runtime resources, such as a view required by Seam components. This servlet provides resources, such as a view that is required for security and JSF UI controls. You need to add the following code snippet in the web.xml file to configure the Seam Resource Servlet:

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>org.jboss.Seam.servlet.SeamResourceServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/Seam/resource/*</url-pattern>
</servlet-mapping>
```

Configuring Seam Servlet Filters

Seam provides a facility to add and configure servlet filters in Seam components. The SeamFilter class is used to support servlet filter to perform the filter operation. The following code snippet shows how to use Seam servlet filters to filter and compress data:

```
<filter>
  <filter-name>Seam Filter</filter-name>
  <filter-class>org.jboss.Seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Seam Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Let's now learn how to configure Seam with EJB 3.

Configuring EJB components in Seam

To integrate EJB with the Seam framework, you need the ejb-jar.xml file. The SeamInterceptor is configured in the ejb-jar.xml file and is used to perform all session beans of the Seam application. The following code snippet shows how to integrate the Seam framework with EJB:

```
<interceptors>
  <interceptor>
  <interceptor-class>org.jboss.Seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor>
  </interceptors>

  <assembly-descriptor>
  <interceptor-binding>
  <ejb-name>*</ejb-name>
  <interceptor-class>org.jboss.Seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
</assembly-descriptor>
```

The Seam components also need to provide the JNDI within the EJB specification so that they can be looked up with their JNDI names. The `@JndiName` annotation is used to work with session beans. This is due to the fact that no standard mapping is specified in the EJB 3 specification. Therefore, you must include the JNDI specification in the components.xml file. The following code snippet shows the use of JNDI with the EJB specification:

```
<core:init jndi-name="myEarName/#{ejbName}/local" /> // inside the context of an EAR

<core:init jndi-name="#{ejbName}/local" />    // outside the context of an EAR
```

The seam.properties file must be placed within the META-INF/seam.properties or META-INF/components.xml file. If you include the Seam components within the WAR files, you must add the seam.properties file in the WEB-INF/classes directory.

Let's now move ahead and learn to create a Seam application.

Creating a Jboss Seam Application

You can also integrate some other technologies, such as Hibernate, Spring, and AJAX, with Seam to provide persistence, Object Relational Mapping (ORM), POJO, and other services to a Seam application. We explore the integration of these technologies, as well as EJB and JSF, with Seam by developing a Seam application, UserLogin. You can find this application on the CD in the code\JavaEE\Chapter16\UserLogin folder. To develop the UserLogin Seam application, you need to perform the following tasks:

- Create an EJB component
- Create Views
- Create Resources

- Package and deploy the Seam application
- Run the application

Let's discuss these tasks, one by one.

Creating an EJB Component

The EJB component contains the business logic, datasource, and persistence services of the UserLogin application. All EJB components must be stored under the `src` directory of the `UserLogin` Seam application. The following files are required to create the EJB component of the UserLogin Seam application:

- User.java
- AuthenticatorAction.java
- RegisterAction.java

Let's discuss these files one by one.

The User.java File

The User.java file is an entity bean that contains the data model of the UserLogin Seam application. It also contains username, password, and the name property used to handle user requests. The @Entity annotation is used to create an entity bean. Listing 16.1 shows the code for the User.java file (you can find this file on CD in the code\JavaEE\Chapter16\UserLogin\src\com\kogent\seam folder):

Listing 16.1: Displaying the Code for the User.java File

```
package com.kogent.seam;

import static org.jboss.seam.ScopeType.SESSION;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

import org.hibernate.validator.Length;
import org.hibernate.validator.NotNull;
import org.hibernate.validator.Pattern;
import org.jboss.seam.annotations.Name;
import org.jboss.seam.annotations.Scope;

@Entity
@Name("user")
@Scope(SESSION)
@Table(name="Customer")
public class User implements Serializable
{
   private String username;
   private String password;
   private String name;

   public User(String name, String password, String username)
   {
         this.name = name;
         this.password = password;
         this.username = username;
   }

   public User() {}

   @NotNull
```

```
    @Length(max=100)
    public String getName()
    {
          return name;
    }
    public void setName(String name)
    {
          this.name = name;
    }

    @NotNull
    @Length(min=5, max=15)
    public String getPassword()
    {
          return password;
    }
    public void setPassword(String password)
    {
          this.password = password;
    }

    @Id
    @Length(min=5, max=15)
    @Pattern(regex="^\\w*$", message="not a valid username")
    public String getUsername()
    {
          return username;
    }
    public void setUsername(String username)
    {
          this.username = username;
    }
    @Override
    public String toString()
    {
          return "User(" + username + ")";
    }
}
```

The AuthenticatorAction.java File

The AuthenticatorAction.java file is used to verify whether or not the user is an authenticated user. The `@Name("authenticator")` annotation is used to identify the method, authenticate(), which must be used to authenticate a user. The `authenticate()` method returns a boolean value, which indicates whether or not the authentication is successful. The username and password can be obtained from the `identity.username` and `identity.password` fields.

Listing 16.2 shows the code for the Authenticator.java file (you can find this file on CD in the code\JavaEE\Chapter16\UserLogin\src\com\kogent\seam folder):

Listing 16.2: Displaying the Code for the AuthenticatorAction.java File

```
package com.kogent.seam;

import static org.jboss.seam.ScopeType.SESSION;

import java.util.List;

import javax.persistence.EntityManager;

import org.jboss.seam.annotations.In;
```

```
import org.jboss.seam.annotations.Name;
import org.jboss.seam.annotations.Out;

@Name("authenticator")
public class AuthenticatorAction
{
  @In EntityManager em;

  @Out(required=false, scope = SESSION)
  private User user;

  public boolean authenticate()
  {
List results = em.createQuery("select u from User u where u.username=#{identity.username}
and u.password=#{identity.password}").getResultList();

        if ( results.size()==0 )
        {
                return false;
        }
        else
        {
                user = (User) results.get(0);
                return true;
        }
  }
}
```

The RegisterAction.java File

The RegisterAction.java file is used to verify the username and password of a user. If the username and password match the session's username and password, the main.xhtml page is displayed. The @In annotation is used to inject the context variable or attribute of the bean of the `UserLogin` Seam application. The EntityManager API uses the register() action listener method and interacts with a database to return a valid JSF outcome.

Listing 16.3 shows the code for the RegisterAction.java file (you can find this file on CD in the code\JavaEE\Chapter16\UserLogin\src\com\kogent\seam folder):

Listing 16.3: Displaying the Code for the RegisterAction.java File

```
package com.kogent.seam;

import static org.jboss.seam.ScopeType.EVENT;
import java.util.List;
import javax.persistence.EntityManager;
import org.jboss.seam.annotations.In;
import org.jboss.seam.annotations.Name;
import org.jboss.seam.annotations.Scope;
import org.jboss.seam.faces.FacesMessages;

@Scope(EVENT)
@Name("register")
public class RegisterAction
{

  @In
  private User user;

  @In
  private EntityManager em;

  @In
```

```
    private FacesMessages facesMessages;

    private String verify;

    private boolean registered;

    public void register()
    {
        if ( user.getPassword().equals(verify) )
        {
  List existing = em.createQuery("select u.username from User u where
u.username=#{user.username}").getResultList();
            if (existing.size()==0)
            {
                em.persist(user);
  facesMessages.add("Successfully registered as #{user.username}");
                registered = true;
            }
            else
            {
  facesMessages.addToControl("username", "Username #{user.username} already exists");
            }
        }
        else
        {
            facesMessages.add("verify", "Re-enter your password");
            verify=null;
        }
    }
    public void invalid()
    {
        facesMessages.add("Please try again");
    }

    public boolean isRegistered()
    {
        return registered;
    }

    public String getVerify()
    {
        return verify;
    }

    public void setVerify(String verify)
    {
        this.verify = verify;
    }
}
```

After developing the EJB components, you need to create the view components to interact with EJB components.

Creating Views

In this application, let's use the JSF framework to develop the view pages. All views must be under the view directory of the UserLogin Seam application. The following views are required for developing the `UserLogin` Seam application:

- index.xhtml
- home.xhtml
- main.xhtml
- register.xhtml
- conversations.xhtml

- template.xhtml

Let's create these views one by one.

The index.xhtml View

The index.xhtml view is used to forward a request to the home.xhtml view. The forwarding of request is done through the `<url-pattern>` mapping specified in the web.xml file.

Listing 16.4 shows the code for the index.xhtml view (you can find the index.xhtml file on CD in the code\JavaEE\Chapter16\UserLogin\view folder):

Listing 16.4: Displaying the Code for the index.xhtml File

```
<html>
  <head>
        <meta http-equiv="Refresh" content="0; URL=home.seam">
  </head>
</html>
```

The home.xhtml View

The home.xhtml view displays the home page of the `UserLogin` Seam application. This view contains two textboxes, username and password, and one submit button. Listing 16.5 shows the code for the home.xhtml view (you can find the home.xhtml file on CD in the code\JavaEE\Chapter16\UserLogin\view folder):

Listing 16.5: Displaying the Code for the home.xhtml File

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:s="http://jboss.com/products/seam/taglib">
  <head>
        <title>home</title>
  </head>
  <h1>User Login </h1>
  <body id="pgHome">
        <f:view>
              <div id="document">
              <div id="container">
              <div id="sidebar">
              <h:form id="login">
              <fieldset>
              <div>
              <h:outputLabel for="username">Login Name:      </h:outputLabel>
              <h:inputText id="username"
              value="#{identity.username}" style="width: 175px;"/>
              <div class="errors"><h:message for="username"/></div>
              </div>
              <div>
              <h:outputLabel for="password">Password :</h:outputLabel>
              <h:inputSecret id="password"
              value="#{identity.password}" style="width: 175px;"/>
              </div>
              <div class="errors"><h:messages globalOnly="true"/></div>
              <div class="buttonBox"><h:commandButton id="login"
              action="#{identity.login}" value="User Login"/></div>
              <div class="notes"><s:link id="register"
              view="/register.xhtml" value="Register New User"/></div>
              </fieldset>
              </h:form>
              </div>
              </div>
              </div>
        </f:view>
```

```
    </body>
</html>
```

The main.xhtml View

The main.xhtml view is used to display the successful login information when a user has successfully logged on to the application. The main.xhtml view uses the template.xhtml file to add images and template text. Listing 16.6 shows the main.xhtml view (you can find the main.xhtml file on CD in the code\JavaEE\Chapter16\UserLogin\view folder):

Listing 16.6: Displaying the Code for the main.xhtml File

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:s="http://jboss.com/products/seam/taglib"
    xmlns:a="https://ajax4jsf.dev.java.net/ajax"
    template="template.xhtml">

    <!-- content -->
        <ui:define name="content">

        <div class="section">
            <h:form id="main">

                    <span class="errors">
                        <h:messages globalOnly="true"/>
                    </span>
    </h:form>
        </div>

    </ui:define>
</ui:composition>
```

The register.xhtml View

The register.xhtml view is used to register a new user. This view contains various textboxes, such as username, name, password, and verify. If any text box is left blank, the view displays a validation error. Listing 16.7 shows the code for the register.xhtml view (you can find the register.xhtml file on CD in the code\JavaEE\Chapter16\UserLogin\view folder):

Listing 16.7: Displaying the Code for the register.xhtml File

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:s="http://jboss.com/products/seam/taglib"
    xmlns:a="https://ajax4jsf.dev.java.net/ajax">
    <head>
        <title>Registration Page</title>
    </head>
    <body id="pgHome">
        <div id="document">

            <div id="container">

            <div id="content">
            <div class="section">
            <h1>
                Register
            </h1>
            </div>
```

```
<div class="section">

<h:form id="register">
<fieldset>

<s:validateAll>

<f:facet name="aroundInvalidField">
<s:span styleClass="errors" />
</f:facet>
<f:facet name="afterInvalidField">
<s:div styleClass="errors">
<s:message />
</s:div>
</f:facet>

<div class="entry">
<div class="label">
<h:outputLabel for="username">Username:</h:outputLabel>
</div>
<div class="input">
<s:decorate id="usernameDecorate">
<h:inputText id="username" value="#{user.username}"
required="true">
<a:support event="onblur" reRender="usernameDecorate" />
</h:inputText>
</s:decorate>
</div>
</div>

<div class="entry">
<div class="label">
<h:outputLabel for="name">Real Name:</h:outputLabel>
</div>
<div class="input">
<s:decorate id="nameDecorate">
<h:inputText id="name" value="#{user.name}" required="true">
<a:support event="onblur" reRender="nameDecorate" />
</h:inputText>
</s:decorate>
</div>
</div>

<div class="entry">
<div class="label">
<h:outputLabel for="password">Password:</h:outputLabel>
</div>
<div class="input">
<s:decorate>
<h:inputSecret id="password" value="#{user.password}"
required="true" />
</s:decorate>
</div>
</div>

<div class="entry">
<div class="label">
<h:outputLabel for="verify">Verify Password:</h:outputLabel>
</div>
```

```
                    <div class="input">
                    <s:decorate>
                    <h:inputSecret id="verify" value="#{register.verify}"
                    required="true" />
                    </s:decorate>
                    </div>
                    </div>

                    </s:validateAll>

                    <div class="entry errors">
                    <h:messages globalOnly="true" />
                    </div>

                    <div class="entry">
                    <div class="label">
                     
                    </div>
                    <div class="input">
                    <h:commandButton id="register" value="Register"
                    action="#{register.register}" />
                     
                    <s:button id="cancel" value="Cancel" view="/home.xhtml" />
                    </div>
                    </div>

                    </fieldset>
                    </h:form>

                    </div>
                    </div>
                    </div>

            </div>
    </body>
</html>
```

The conversations.xhtml View

The conversations.xhtml view is used to maintain a list of concurrent conversations in the current user session. The concurrent conversations for the current user session is maintained in a component named `#{conversationList}`. You can iterate a list of current information of a user, such as description of the conversations, their starting time, and the time the conversation was last used.

Listing 16.8 shows the code for the conversations.xhtml view (you can find the conversations.xhtml file on CD in the code\JavaEE\Chapter16\UserLogin\view folder):

Listing 16.8: Displaying the Code for the conversations.xhtml File

```
<div xmlns="http://www.w3.org/1999/xhtml"
    xmlns:c="http://java.sun.com/jstl/core"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:s="http://jboss.com/products/seam/taglib">

    <div class="section">
<h1><h:outputText rendered="#{not empty conversationList}" value="Workspaces"/></h1>
    </div>

    <div class="section">
          <h:form>
```

```
                    <h:dataTable value="#{conversationList}" var="entry">
                        <h:column>
    <h:commandLink action="#{entry.select}" value="#{entry.description}"/>
                                 
    <h:outputText value="[current]" rendered="#{entry.current}"/>
                        </h:column>
                        <h:column>
                                <h:outputText value="#{entry.startDatetime}">
    <s:convertDateTime type="time" pattern="hh:mm"/>
                                </h:outputText>
                                -
                                <h:outputText value="#{entry.lastDatetime}">
    <s:convertDateTime type="time" pattern="hh:mm"/>
                                </h:outputText>
                        </h:column>
                    </h:dataTable>
            </h:form>
      </div>

    </div>
```

The template.xhtml View

The template.xhtml view is used as a template page to create other pages, such as main.xhtml. The purpose of this page is to add extra information, such as an image and a cascade style sheet, on a specified page. Listing 16.9 shows the code for the template.xhtml view (you can find the template.xhtml file on CD in the code\JavaEE\Chapter16\UserLogin\view folder):

Listing 16.9: Displaying the Code for the template.xhtml File

```
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:s="http://jboss.com/products/seam/taglib">
      <head>
            <title>Successful login page</title>
      </head>
      <body>

            <div id="document">
            <div id="header">
            <div id="status">
            You have successfully login as #{user.name}
            | <s:link id="logout" action="#{identity.logout}" value="Logout"/>
            </div>
            </div>
            <div id="container">
                    <div id="sidebar">
                    <ui:insert name="sidebar"/>
            </div>
            <div id="content">
                    <ui:insert name="content"/>
                    <ui:include src="conversations.xhtml" />
            </div>
            </div>

            </div>
      </body>
    </html>
```

Creating Resources

In an application, various resources, such as name of the database and entity manager need to be configured. These resources are configured in various resources files, such as components.xml, pages.xml, and

persistence.xml. These files are stored in the resources directory of an application. In the UserLogin Seam application, the JSF and EJB 3 technologies are used with the Seam framework; therefore, you need to configure these technologies while deploying the UserLogin Seam application. For example, you need the pages.xml file to configure the JSF framework, and the persistence.xml file to configure Java persistency. All these files must exist in the resources directory of the UserLogin Seam application. To deploy the `UserLogin` Seam application, you need to create the following configuration files:

- components.xml
- web.xml
- pages.xml
- persistence.xml
- jboss-web.xml

Let's create these configuration files one by one.

The components.xml File

The components.xml file is stored in the resources directory of the WAR file. It is used to declare and configure the application component and various Seam runtime services, such as jBPM, pageflow, and security. Listing 16.10 shows the code for the components.xml file (you can find this file on CD in the code\JavaEE\Chapter16\UserLogin\resources folder):

Listing 16.10: Displaying the Code for the components.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
   xmlns:core="http://jboss.com/products/seam/core"
   xmlns:persistence="http://jboss.com/products/seam/persistence"
   xmlns:transaction="http://jboss.com/products/seam/transaction"
   xmlns:security="http://jboss.com/products/seam/security"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=
"http://jboss.com/products/seam/core http://jboss.com/products/seam/core-2.0.xsd
http://jboss.com/products/seam/persistence http://jboss.com/products/seam/persistence-
2.0.xsd
http://jboss.com/products/seam/security http://jboss.com/products/seam/security-2.0.xsd
http://jboss.com/products/seam/components http://jboss.com/products/seam/components-
2.0.xsd">
   <core:manager conversation-timeout="120000"
   concurrent-request-timeout="500"
   conversation-id-parameter="cid"/>

   <!-- <transaction:entity-transaction entity-manager="#{em}"/> -->

   <persistence:entity-manager-factory name="bookingDatabase"/>

   <persistence:managed-persistence-context name="em"
   auto-create="true"
   entity-manager-factory="#{bookingDatabase}"/>

   <security:identity authenticate-method="#{authenticator.authenticate}"/>

</components>
```

The web.xml File

The web.xml file is used to configure the UserLogin Seam application as well as the Seam and JSF frameworks. It is also used to configure a servlet and Seam listener. Listing 16.11 shows the code for the web.xml file (you can find this file on CD in the code\JavaEE\Chapter16\UserLogin\resource folder):

Listing 16.11: Displaying the Code for the web.xml File

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

```
    <context-param>
          <param-name>org.ajax4jsf.VIEW_HANDLERS</param-name>
          <param-value>com.sun.facelets.FaceletViewHandler</param-value>
    </context-param>

    <!-- Seam -->

  <listener>
          <listener-class>org.jboss.seam.servlet.SeamListener</listener-class>
    </listener>

    <filter>
          <filter-name>Seam Filter</filter-name>
          <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
    </filter>

    <filter-mapping>
          <filter-name>Seam Filter</filter-name>
          <url-pattern>/*</url-pattern>
    </filter-mapping>

    <servlet>
          <servlet-name>Seam Resource Servlet</servlet-name>
          <servlet-class>org.jboss.seam.servlet.ResourceServlet
          </servlet-class>
    </servlet>

    <servlet-mapping>
          <servlet-name>Seam Resource Servlet</servlet-name>
          <url-pattern>/seam/resource/*</url-pattern>
    </servlet-mapping>

    <!-- Faces Servlet -->

    <servlet>
          <servlet-name>Faces Servlet</servlet-name>
          <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
          <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
          <servlet-name>Faces Servlet</servlet-name>
          <url-pattern>*.seam</url-pattern>
    </servlet-mapping>

    <!-- JSF parameters -->

    <context-param>
          <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
          <param-value>.xhtml</param-value>
    </context-param>

    <context-param>
          <param-name>facelets.DEVELOPMENT</param-name>
          <param-value>true</param-value>
    </context-param>
  </web-app>
```

The pages.xml File

The pages.xml configuration file is used to configure various properties of the views of the UserLogin Seam application. This file is stored in the WEB-INF directory. It defines action, navigation, and error handling events

of the UserLogin application. The `action` attribute is used to invoke the action class before a page is invoked. Listing 16.12 shows the code for the pages.xml file (you can find this file on CD in the code\JavaEE\Chapter16\UserLogin\view folder):

Listing 16.12: Displaying the Code for the pages.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<pages xmlns="http://jboss.com/products/seam/pages"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://jboss.com/products/seam/pages
http://jboss.com/products/seam/pages-2.0.xsd"

    no-conversation-view-id="/main.xhtml"
    login-view-id="/home.xhtml">

    <page view-id="/register.xhtml">

        <action if="#{validation.failed}"
        execute="#{register.invalid}"/>

        <navigation>
            <rule if="#{register.registered}">
                <redirect view-id="/home.xhtml"/>
            </rule>
        </navigation>

    </page>

    <page view-id="/home.xhtml">

        <navigation>
            <rule if="#{identity.loggedIn}">
                <redirect view-id="/main.xhtml"/>
            </rule>
        </navigation>

    </page>

    <page view-id="/main.xhtml"
        login-required="true">

        <navigation from-action="#{hotelBooking.selectHotel(hot)}">
            <redirect view-id="/hotel.xhtml"/>
        </navigation>

    </page>

    <page view-id="*">

        <navigation from-action="#{identity.logout}">
            <redirect view-id="/home.xhtml"/>
        </navigation>

        <navigation from-action="#{hotelBooking.cancel}">
            <redirect view-id="/main.xhtml"/>
        </navigation>

    </page>

    <exception class="org.jboss.seam.security.NotLoggedInException">
        <redirect view-id="/home.xhtml">
<message severity="warn">You must be logged in to use this feature</message>
        </redirect>
    </exception>

</pages>
```

The persistence.xml File

The persistence.xml file is used to connect to the datasource provided by EJB persistence. This file contains some vendor-specific information, such as persistence-unit and datasource information. Listing 16.13 shows the code for the persistence.xml file (you can find this file on CD in the code\JavaEE\Chapter16\UserLogin\resources folder):

Listing 16.13: Displaying the Code for the persistence.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence

   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">
  <persistence-unit name="bookingDatabase" transaction-type="JTA">

  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>jdbc/__default</jta-data-source>

  <properties>
        <property name="hibernate.dialect"

        value="GlassfishDerbyDialect"/>
        <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
        <property name="hibernate.show_sql" value="true"/>
<property name="hibernate.cache.provider_class"
value="org.hibernate.cache.HashtableCacheProvider"/>

<property name="hibernate.transaction.manager_lookup_class"
  value="org.hibernate.transaction.SunONETransactionManagerLookup"/>
        </properties>

  </persistence-unit>
</persistence>
```

The jboss-web.xml File

The jboss-web.xml file is a Deployment Descriptor that specifies how to deploy a WAR application on the JBoss application server. Listing 16.14 shows the jboss-web.xml file (you can find this file on CD in the code\JavaEE\Chapter16\UserLogin\resources folder):

Listing 16.14: Displaying the Code for the jboss-web.xml File

```
<jboss-web>

  <class-loading java2ClassLoadingCompliance="false">
        <loader-repository>
              seam.jboss.org:loader=jboss-seam-jpa

<loader-repository-config>java2ParentDelegation=false</loader-repository-config>
        </loader-repository>

  </class-loading>
</jboss-web>
```

You need the `seam.properties` file, which is a blank file, to deploy the `UserLogin` Seam application. You also need to arrange the files created for the `UserLogin` Seam application in the directory structure, as shown in Figure 16.1:

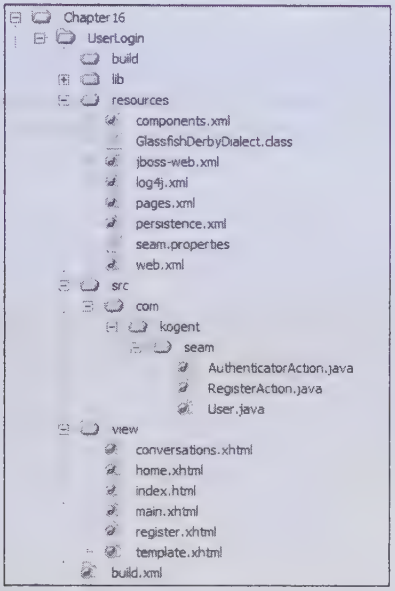


Figure 16.1: Showing the Directory Structure of the UserLogin Seam Application

You need to add various packages in the lib folder to compile the UserLogin Seam application. These packages are required to provide the functionalities of different frameworks used in the application, such as AJAX and Hibernate. These packages are as follows:

- ajax4jsf-1.1.1.jar
- commons-beanutils-1.7.0.jar
- commons-digester-1.6.jar
- ejb3-persistence.jar
- hibernate3.jar
- hibernate-annotations.jar
- hibernate-entitymanager.jar
- jboss-archive-browsing.jar
- jboss-common.jar
- jboss-seam.jar
- jboss-seam-debug.jar
- jboss-seam-ui.jar
- jsf-facelets.jar
- thirdparty-all

Packaging and Deploying the Seam Application

You need the Ant tool to package the `UserLogin` Seam application. Ant is a Java tool used to compile and package Web applications. Ant uses an XML file, build.xml, to compile and package Web applications. You can download the apache-ant-1.8.0-bin.zip file from the Apache website (`http://ant.apache.org/bindownload.cgi`). After downloading the apache-ant-1.8.0-bin.zip file, install the Ant tool in your system and set the path for the `ANT_HOME` environment variable in the class path.

You can deploy a Seam application by using the following servers:

- Glassfish
- JBoss

Let's explore these next.

Using Glassfish

Packaging and deploying the Seam application on the Glassfish V3 Application server can be done by using the Ant tool. Run the `ant glassfish` command at command prompt to obtain the WAR file of the UserLogin Seam application, For example, type the following command at command prompt from the application directory:

```
ant glassfish
```

Figure 16.2 shows the output of the preceding command:

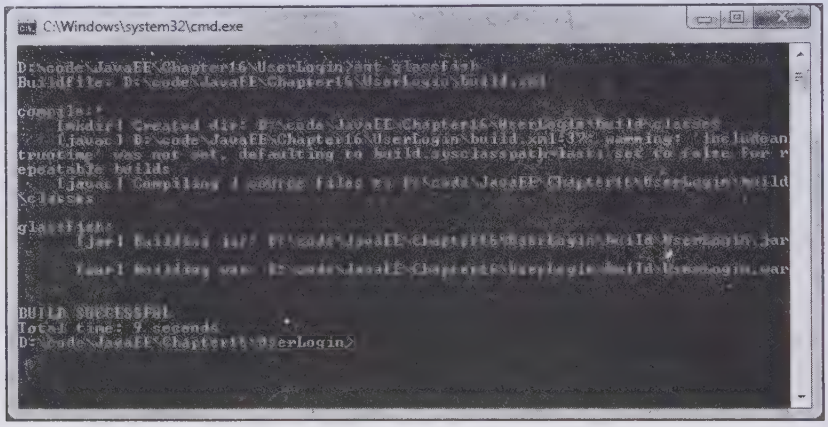


Figure 16.2: Packaging the UserLogin Application on Glassfish

The `ant glassfish` command generates the `UserLogin.war` file, which is stored in the `build` directory of the `UserLogin` Seam application. Now, open the admin console of the Glassfish V3 application server (`http://localhost:4848/`) and deploy the `UserLogin` Seam application.

Using JBoss

The JBoss application server is a free Java EE certified application server. It can be downloaded from the SourcdForge.net website by clicking the hyperlink of the required version, for example `http://www.jboss.org/jbossas/downloads/` URL for jboss-4.2.0.GA.zip. After downloading the JBoss application server (jboss-4.2.0.GA.zip), extract the zip file and install it on the working drive (C:\ drive).

NOTE

The JBoss Seam 2.0 framework supports only JBoss application server 4.2 or higher version.

After installing the JBoss application server on your system, you need to package and deploy the UserLogin Seam application on it. Prior to the process of packaging and deploying the UserLogin application, you need to ensure that the following code snippet is added to the build.xml file:

```
<!-- JPA and Seam POJO on JBoss AS 4.2.0-->
  <target name="jboss" depends="compile">

    <mkdir dir="${build.jars}"/>

    <jar destfile="${build.jars}/${projname}.jar">
      <fileset dir="${build.classes}">
        <include name="**/*.class"/>
      </fileset>
      <fileset dir="${resources}">
        <include name="seam.properties" />
        <include name="import.sql" />
      </fileset>
```

```
    <metainf dir="${resources}">
      <include name="persistence.xml" />
    </metainf>
  </jar>

  <war destfile="${build.jars}/${projname}.war"
       webxml="${resources}/web.xml">

    <webinf dir="${resources}">
      <include name="faces-config.xml" />
      <include name="pages.xml" />

      <include name="jboss-web.xml" />
      <include name="components.xml" />
    </webinf>
    <lib dir="${seamlib}">
      <include name="jboss-seam.jar" />

      <include name="jboss-seam-ui.jar" />
      <include name="jboss-seam-debug.jar" />
    </lib>
    <lib dir="${lib}">

      <include name="jboss-el.jar" />
      <include name="jsf-facelets.jar" />
      <include name="ajax4jsf-*.jar" />

      <!-- needed by ajax4jsf -->
      <include name="commons-digester*.jar" />
      <include name="commons-beanutils*.jar" />

      <include name="oscache*.jar" />
    </lib>
    <lib dir="${build.jars}">

      <include name="${projname}.jar" />
    </lib>
    <fileset dir="${view}"/>

  </war>
</target>
```

Run the Ant jboss command at command prompt; you will get a WAR file of the `UserLogin` Seam application. For example, type the following command at command prompt:

```
ant jboss
```

After the preceding command is executed, the generated WAR file (UserLogin.war) is saved in the build directory of the UserLogin application. Copy this WAR file from the build directory (`D:\code\JavaEE\Chapter16\UserLogin\build`) and paste it at the deployed location of the JBoss application server (`C:\jboss-4.2.0.GA\server\default\deploy`).

Next, you need to start the JBoss application server. To start the server, change the working directory to the bin subdirectory, where the JBoss application server is installed, at command prompt. Type the `run` command at command prompt to start the JBoss application server. The JBoss application server is started, as shown in Figure 16.3:

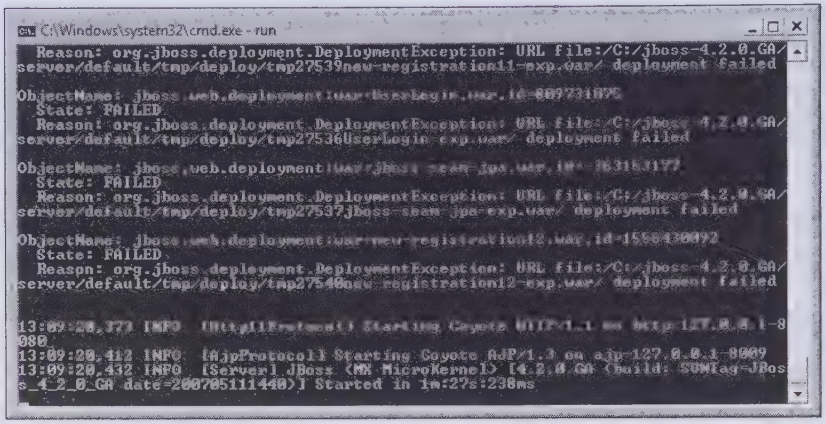


Figure 16.3: Showing the JBoss Application Server in the Running Mode

Now, deploy the `UserLogin` Seam application by copying it from the `D:\code\JavaEE\Chapter16\UserLogin\build` directory and pasting it in the `C:\jboss-4.2.0.GA\server\default\deploy` directory.

After successfully deploying the `UserLogin` Seam application, you need to run the application.

Running the Application

Perform the following steps to run the UserLogin application deployed on the Glassfish V3 application server:

- Open the Web browser and type the following URL:
  `http://localhost:8080/UserLogin/`

The home page of the application appears, as shown in Figure 16.4:

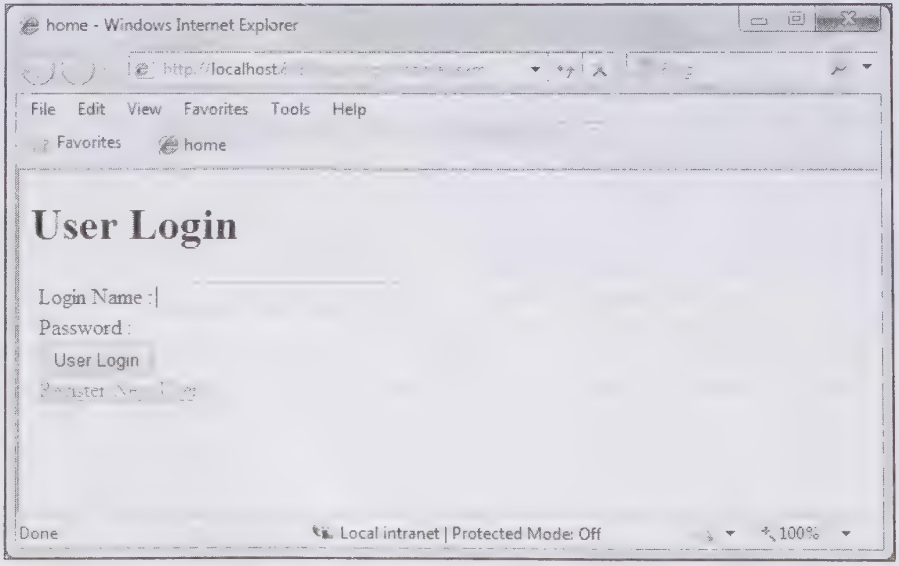


Figure 16.4: Displaying the Home Page Running on Glassfish V3

1. Click the `Register New User` hyperlink to open the `Registration` page.
2. Enter the values in the username, real name, password, and verified password text boxes and click the `Register` button to register a new user, as shown in Figure 16.5:

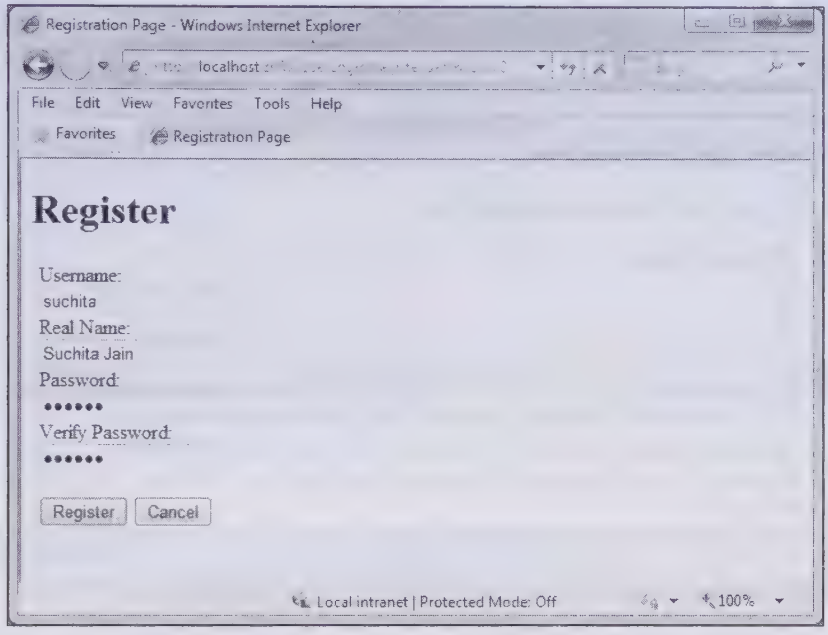


Figure 16.5: Displaying the Registration Page

Figure 16.6 shows the successful registration message:

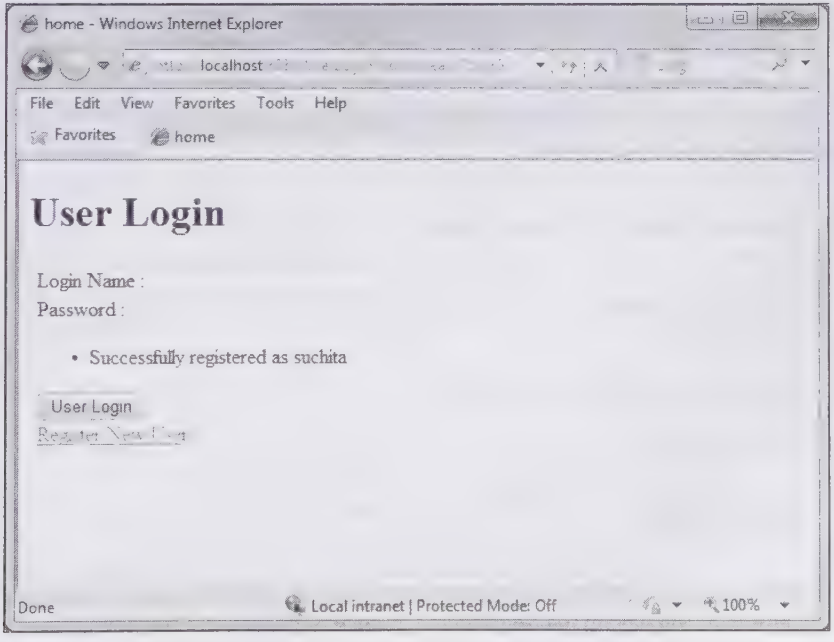


Figure 16.6: Showing the Home Page with a Message on Successful Registration

4. Enter the login name as `suchita`, password as `kogent`, and click the `User Login` button. The successful login page with the welcome message appears, as shown in Figure 16.7:

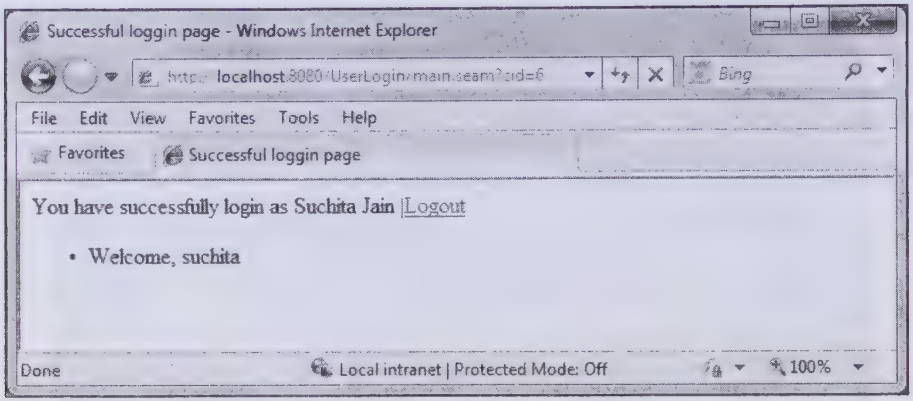


Figure 16.7: Showing the Home Page with Welcome Message

If any of the text boxes displayed in Figure 16.5 has been left blank, the application generates a validation error message, which is defined in the User.java file (entity bean). Figure 16.8 shows a registration page with the validation error message:

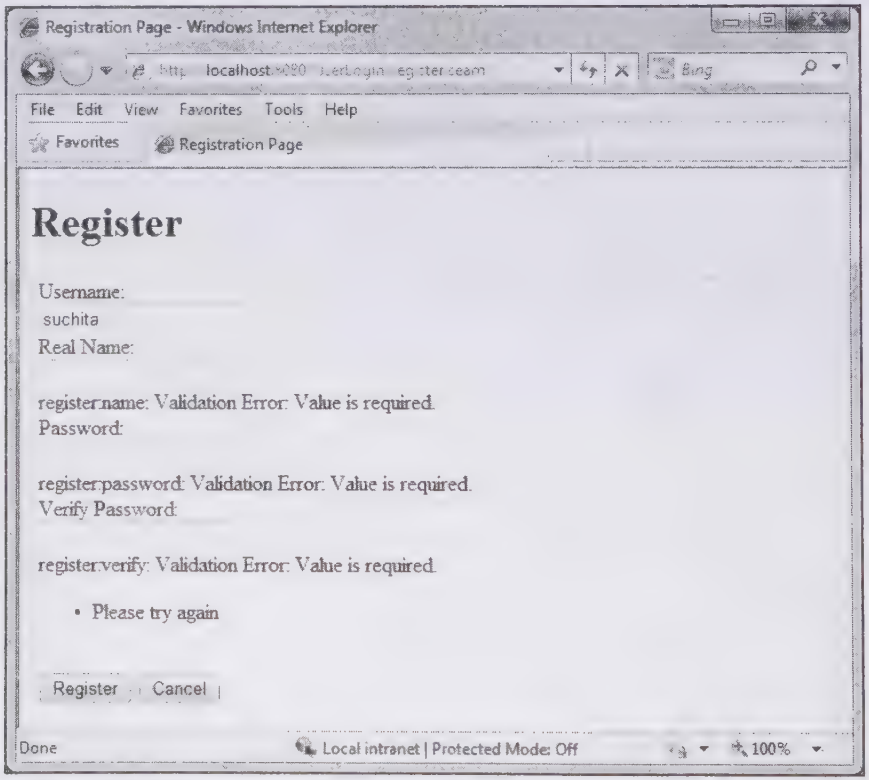


Figure 16.8: Displaying the Registration Page with Validation Error Messages

If you enter an invalid username or password, the application displays the `Login failed` message, as shown in Figure 16.9:

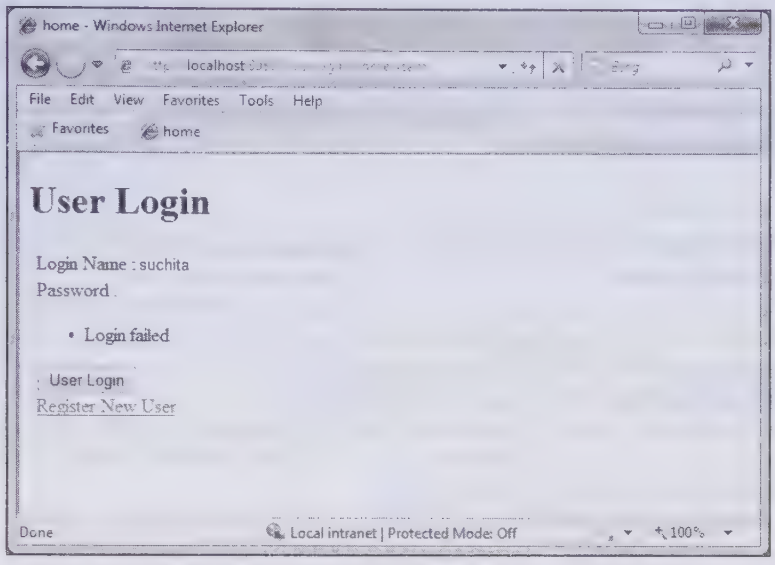


Figure 16.9: Displaying the Home Page with the Login Failed Message

If you have used JBoss to deploy the application, use the following URL to run the application:

```
http://localhost:8080/userLogin
```

The home page of the UserLogin application appears, as shown in Figure 16.10:

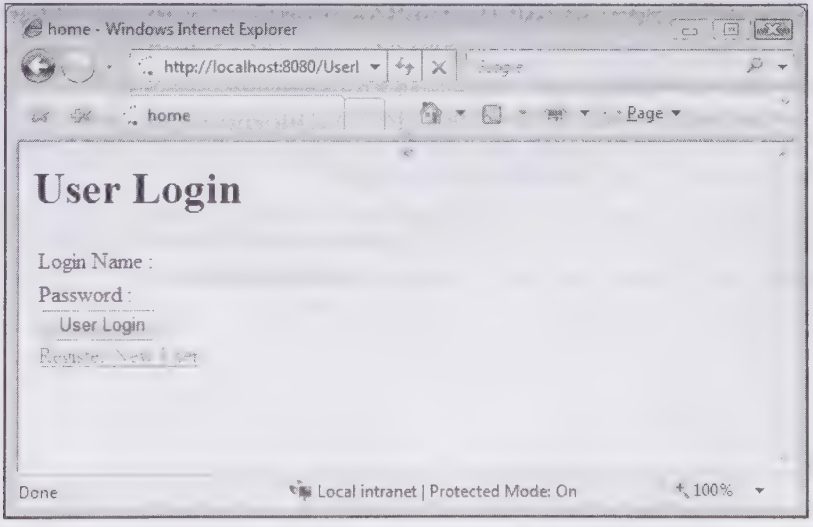


Figure 16.10: Displaying the Home Page Running on the JBoss Application Server

With this, we come to the end of the chapter. Let's now summarize the main points of the chapter.

Summary

In this chapter, you have learned about the JBoss Seam framework and its features. You have also learned about Seam context, Seam components, and built-in Seam components. In addition, you have learned how to configure Seam components that are used for developing Seam applications. Finally, you have learned to develop a Seam application, `UserLogin`.

In the next chapter, you learn about the Java EE connector architecture.

Quick Revise

Q1. The ..... package provides Seam's annotation.

A. org.jboss.seam.jms **B. org.jboss.seam.core**

C. org.jboss.seam.annotations **D. org.jboss.seam.servlet**

Ans. C

Q2. What is JBoss Seam?

Ans. JBoss Seam is a light weight framework formed by integration of EJB 3 and JSF.

Q3. What is a Seam context?

Ans. A Seam context is the context variable that is generated as well as destroyed by the Seam framework. Some examples of Seam context variables are page context, conversation context, session context, business process context, and application context.

Q4. What are Seam components?

Ans. Seam components are the POJO components, such as JavaBeans and EJB that are used for developing Seam applications. Seam framework also provides built-in Seam components, such as internationalization and mail.

Q5. What is jBPM?

Ans. jBPM stands for Java Business Process Management. It is used to develop Business Process Management (BPM) of an organization.

Q6. What is BPM?

Ans. BPM is **a well-defined group of tasks which is to be performed by the users or by the well defined software** systems.

Q7. Can a Seam application run on any other server than JBoss?

Ans. Yes, it can run Seam applications on Glassfish and Tomcat 5.5.

Q8. Can Seam run on JDK 1.4?

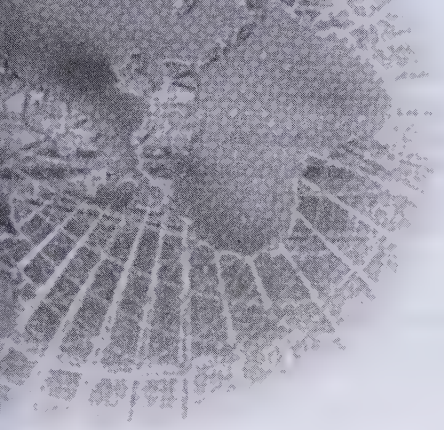
Ans. No, Seam only works on JDK 5.0 and higher versions because it uses annotations and other JDK 5.0 features, such as Java platform debugger architecture, internationalization, and ProcessBuilder.

Q9. How can we manage a state in the Seam framework?

Ans. A state in the Seam framework can be managed by several scopes, such as business process and conversation scope.

Q10. Name some Seam's annotations used to develop a Seam application.

Ans. @Name, @Scope, @Conversational, @Install, @In, and @Out.



17

Java EE Connector Architecture 1.6

Enterprise applications run on Java EE, which supports the application server. An enterprise application is a distributed application that can handle large amounts of data required in the day to day working of an organization. To manage such voluminous data efficiently, enterprise applications need Enterprise Information Systems (EIS). EIS is a computing system that facilitates an organization to integrate and coordinate their business transactions in an efficient manner.

Java EE Connector Architecture (JCA) is a Java-based technology that is used to connect a Java EE-compliant application server with EIS. JCA enables enterprise applications to deal with large volumes of data. Both the application server and EIS together constitute a part of Enterprise Application Integration (EAI) solutions. In the Java EE platform, separate containers for client applications and application components, such as servlets, JavaServer Pages (JSP), and Enterprise JavaBeans (EJB) provide deployment as well as runtime support for the components used in an enterprise application.

In Java, containers are able to run on existing systems, such as Web servers, Transaction Processing (TP) monitors, application servers, and database systems. This enables an organization to take advantage of the features of both the existing system as well as Java EE. Using the capabilities of Java EE, developers can write new applications as well as encapsulate parts of existing applications in EJBs, JSPs, or servlets. JCA also formalizes the interactions, relationships, and packaging of the integration layer that connects the application server with EIS; thereby, enabling the EIS system to handle large amounts of data. With the help of application servers, it is possible for enterprise applications to connect with heterogeneous EIS systems, and this connectivity is bi-directional. Note that bi-directional connectivity is provided by JCA with the help of standard contracts. Various tools and frameworks are provided by EAI vendors to simplify the integration of EIS with an enterprise application.

JCA is an EAI initiative that provides a standards-based mechanism to access legacy systems from Java or Java EE applications, which significantly reduces the difficulties faced in legacy system integration. JCA is able to solve problems relating to the integration of EIS with enterprise applications by using the Common Client Interface (CCI) Application Programming Interface (API). Earlier, developers used the Java Database Connectivity (JDBC) API to manage data. The JDBC API allows Java programs to access data in a standardized and easy way. However, the JDBC technology does not provide the advantage of platform independency, which is provided by JCA.

JCA is used for legacy data and application integration just as JDBC is used for database interaction. Compared to JDBC, JCA provides an easy-to-use, standardized, legacy system-independent way to interact with different backend systems. The use of JCA provides the benefits of Java technology, such as code reusability, platform independency, and code manageability.

This chapter begins by describing important concepts of JCA. You are then acquainted with the Lifecycle Management contract that controls the life cycle of a resource adapter that is used to connect EIS with an enterprise application. Moreover, the chapter explores the Workflow management process of a resource adapter and lists the differences between the JDBC and JCA technologies, both of which are used to access a data store. Apart from this, you learn about the Inbound Communication model, EJB invocation, and the CCI API. The chapter also explores JCA exceptions. Toward the end, you learn how to package and deploy a resource adapter.

Let's begin by describing the important concepts of JCA.

Describing the Key Concepts of the JCA

With the help of JCA, enterprise application components can communicate with EIS efficiently. JCA is a Java-based technology that connects an application server to various types of EIS. Examples of EIS are Enterprise Resource Planning (ERP), mainframe transaction processing, and non-relational databases. Knowledge of these and several other concepts is necessary to understand JCA. We discuss the concepts under the following broad-level headings:

- EIS
- Resource manager
- Resource adapter
- Managed environment

- Non-managed environment
- Connection of an application client with a resource manager
- System contracts
- CCI

Let's now start by discussing EIS.

Enterprise Information Systems

EIS is the backend computing system with which components of an enterprise application must integrate to retrieve the required data. EIS provides system-level services to the clients of an enterprise application. The main components of EIS are as follows:

- **The ERP system**—Handles the internal and external resources of an organization, including tangible assets, financial resources, materials, and human resources
- **The mainframe transaction processing system**—Contains information needed to process data transactions in a database system
- **The legacy database system**—Controls the storage, retrieval, security, and integrity of data in a database

To connect an enterprise application with EIS, a suitable EIS connector (a JCA-compliant resource adapter) needs to be installed on an application server. After connecting the enterprise application with EIS, you can develop the components of the enterprise application to communicate with EIS by using the CCI API.

Figure 17.1 shows the use of EIS along with a resource adapter and an application server:

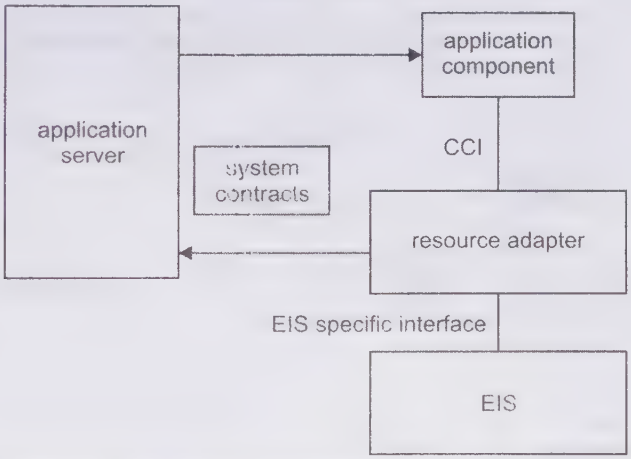


Figure 17.1: Displaying the Java EE Application Server with JCA Components and EIS

As shown in Figure 17.1, an application server communicates with the application components with the help of the CCI API. In addition, the system-level contracts define the rules on the basis of which a resource adapter can communicate with the application server. You must note that the resource adapter plays an important role in maintaining the communication between the application component and EIS.

JCA services are provided by an application server vendor and EIS is provided by an EIS vendor. By supporting JCA, all Java EE-compliant application servers can handle multiple and heterogeneous EIS resources. In this way, JCA helps to boost the productivity of a Java EE application developer while reducing development costs and protecting the existing investment in EIS systems by providing a scalable integration solution through Java EE. An EIS resource provides EIS-specific services to its clients.

Some examples of EIS resources are as follows:

- A record or set of records in a database system
- A business object in an ERP system
- A transaction processing application containing a transaction program

Next, let's discuss the resource manager.

Resource Manager

The role of a resource manager is to manage a group of sharable EIS resources. The resource manager can participate in data transactions, which are coordinated and controlled by a transaction manager. In JCA, a resource manager may have two types of clients, a client tier application or a middle-tier application server. The resource manager and the clients in JCA are located on different machines. A resource manager may be a mainframe transaction processing system, a database system, or an ERP system.

Now, let's discuss the resource adapter.

Resource Adapter

JCA, for a particular EIS type, is implemented by a Java EE component called resource adapter. A resource adapter improves the interaction between an enterprise application and EIS. The resource adapter is usually stored in a Resource Adapter ARchive (RAR) module (also known as .rar file) and is deployed on any Java EE application server. The `.rar` file can also be stored in an Enterprise ARchive (EAR) file.

Figure 17.2 shows the structure of a resource adapter:

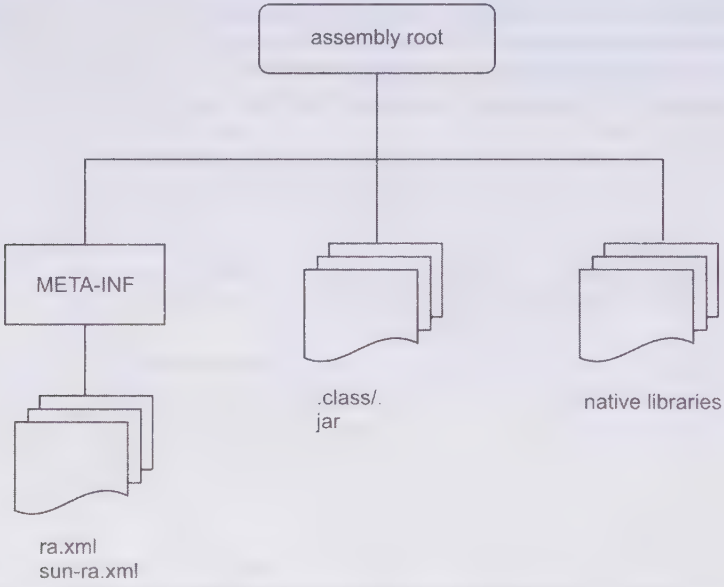


Figure 17.2: Showing the Structure of a Resource Adapter

As you can see in Figure 17.2, a resource adapter comprises the class file of the ResourceAdapter JavaBean, Deployment Descriptor of a resource adapter, and native libraries. All these are packaged under the assembly root of the resource adapter and are required to execute the resource adapter.

A resource adapter is similar to a JDBC driver used in relational database programming. The resource adapter and EIS both provide classes and interfaces that help an enterprise application to access resources not contained within an application server.

After learning about the resource adapter, let's discuss the concepts of managed and non-managed environments.

Managed Environment

In a managed environment, a resource adapter is deployed inside an application server. The methods of the resource adapter are called from inside an EJB container. An enterprise application may consist of application components, such as EJBs, JSPs, and servlets, which are deployed on their corresponding containers. These

application components possess the capability to access the EIS system. In the context of JCA, these enterprise applications are known as managed applications.

Non-Managed Environment

In a non-managed environment, a resource adapter is separated from an application server and is used outside the application server as a library. In a non-managed environment, with the help of JCA, clients such as applets or Java client applications can access an EIS system. In other words, in an application, a client can directly use a resource adapter library. This allows the resource adapter to offer low-level transactions and security to its clients. In the context of JCA, such applications are called non-managed applications.

Now, let's discuss the connection between an application client and a resource manager.

Connection of an Application Client with a Resource Manager

An application client and a resource manager are linked through a connection, so that the client application can access the services provided by the resource manager. These connections are classified into two categories, namely transactional connections and non-transactional connections. At a single point of time, only one type of connection is possible. A connection between a resource manager and an application client can be bi-directional; however, connection type is mainly based on the capabilities of a resource manager.

System Contracts

System contracts enable a resource adapter to link with the services of an application server to manage connections, transactions, and security. If an application server vendor wants the application server to support JCA, then the vendor must extend the application server only once so that connectivity between the application server and multiple EIS is assured. Similarly, an EIS vendor should provide a resource adapter that is able to integrate with any application server that supports JCA. You should note that only JCA can provide system-level pluggability.

Figure 17.3 shows how a standard EIS resource adapter can connect to multiple application servers and vice versa:

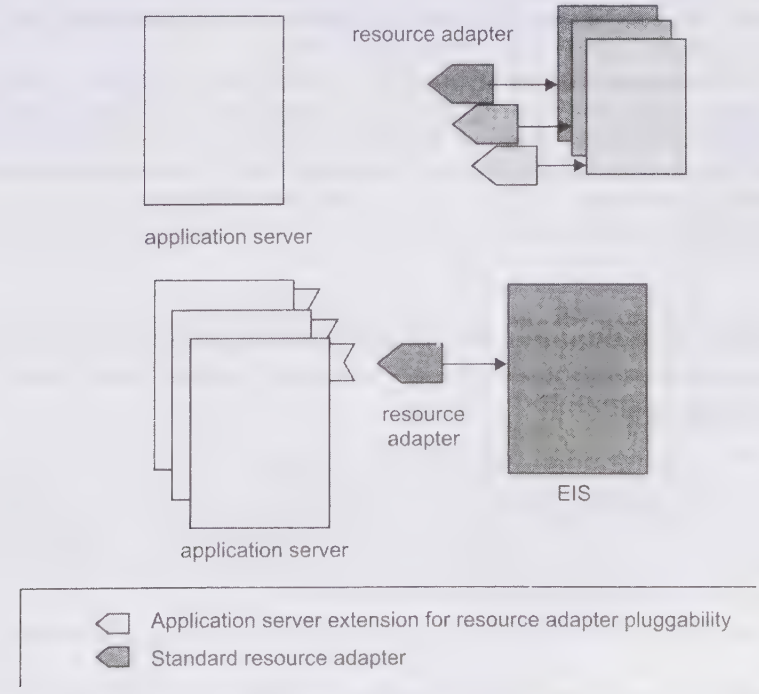


Figure 17.3: System-Level Pluggability between an Application Server and EIS

Figure 17.3 shows how an application server integrates with EIS. Prior to the introduction of JCA, application server and EIS were integrated manually by connecting them in a vendor-specific and non-standard way. As the number of EIS systems increased, it became difficult to manually manage integration between the application server and EIS. This problem was resolved with the introduction of JCA.

To understand how JCA helps in the problem of integration, let's consider an example in which there are `p` application servers and `q` EIS. JCA reduces the scope of the integration problem by changing it from a `p x q` problem to a `p + q` problem.

JCA defines the following standard system-level contracts between an application server and EIS, enabling outbound connectivity to the EIS system:

- **The Connection Management contract**—Allows an application server to offer its own services to create and manage connection pools with an underlying EIS resource. The creation and management of connection pools with EIS resources provide a scalable connection-management facility to support multiple clients.
- **The Transaction Management contract**—Extends the transactional capability of an application server to the underlying EIS resource management. This contract makes it possible for an application server to monitor a transaction by using a transaction manager across multiple resource managers.
- **The Security Management contract**—Allows an application server to access the information stored in EIS in a secured manner. This prevents security threats, such as loss of valuable information related to resources that are managed by EIS.

Apart from the preceding system-level contracts, JCA also defines the following system-level contracts between an application server and EIS, enabling inbound connectivity to an EIS system:

- **The Message Inflow contract**—Allows a resource adapter to deliver messages to their endpoints asynchronously. In addition, this contract allows various message providers, such as Java Message Service (JMS) to be plugged in an application server with the help of a resource adapter.
- **The Transaction Inflow contract**—Allows propagation of an imported transaction to an application server. In addition, this contract helps a resource adapter to manage the inflow of a transaction and prevent the atomicity, consistency, isolation, and durability (ACID) properties of the imported transactions.

JCA also defines the following system-level contracts between the application server and EIS to enable the resource adapter life cycle management and thread management:

- **The Lifecycle Management contract**—Provides a mechanism to an application server to manage the life cycle of a resource adapter. This contract allows an application server to bootstrap the instance of a resource adapter during either the deployment process or the starting up of the application server. In addition, the instance is notified about its undeployment or the shutting down of the application server.
- **The Work Management contract**—Allows a resource adapter to submit the instances of the Work interface to an application server. This allows the application server to execute Work instances by dispatching a thread.

NOTE

You learn more about the Lifecycle Management contract in the Describing the Life Cycle Management of a Resource Adapter section.

Common Client Interface

JCA provides a common interface in the form of CCI for clients to access EIS. CCI provides standard client APIs to solve the problems faced while integrating an application server with heterogeneous EIS systems. Integration across heterogeneous EIS systems is made possible with the help of CCI APIs. The CCI API provides a set of classes and interfaces that enable communication between the application server and EIS through a resource adapter. The CCI API allows a programmer of enterprise applications to execute operations with EIS and to manage data object/records as input, output, or return values.

Table 17.1 describes the interfaces in the CCI API, which are available in the javax.resource.cci package:

Table 17.1: Describing the Interfaces of the javax.resource.cci Package	
Interface	**Description**
`ConnectionFactory`	Provides an interface used by a resource adapter to establish a connection to an EIS instance. In other words, a client looks up the instance of the `ConnectionFactory` interface from Java Naming and Directory Interface (JNDI) to establish an EIS connection.
`Connection`	Represents a connection to a particular EIS.
`ConnectionSpec`	Passes the connection-specific properties to the `ConnectionFactory` interface when a connection request is made.
`Interaction`	Allows application components to execute EIS methods, such as procedures stored in a database.
`InteractionSpec`	Contains the properties for application components, which interact with EIS.
`Record`	Contains record instances. Classes of record instances include `MappedRecord`, `IndexedRecord`, and `ResultSet`.
`RecordFactory`	Creates the `MappedRecord` and `IndexedRecord` instances.
`Indexedrecord`	Represents an ordered collection of a record instance of the `java.util.List` type.

A client application component uses the CCI API to interact with EIS. Connection with the resource manager is established by using the `ConnectionFactory` instance. The instance of the `Connection` interface corresponds to an EIS connection and is used to perform EIS transactions. Application components interact with EIS to perform multiple tasks, such as accessing data from a specific table and using an iteration object. The `InteractionSpec` object provides the specifications for the interaction objects described in JCA. Data from EIS can be read and written to a table by using an instance of the class that implements the Record interface of the CCI API. The Record instance is classified into three main categories, namely the `MappedRecord` instance, the `IndexedRecord` instance, and the `ResultSet` instance.

After understanding the main concepts of JCA, let's discuss the life cycle management of a resource adapter.

Describing the Life Cycle Management of a Resource Adapter

The Lifecycle Management contract plays an important role in managing the life cycle of a resource adapter. It provides an environment to connect and integrate an application server with EIS and vice versa. Therefore, you can say that the resource adapter is the main means of communication among application servers, enterprise application components, and EIS. JCA provides well-defined contracts by which a resource adapter can communicate with the other components of an enterprise application.

The life cycle of a resource adapter starts when an application server bootstraps a resource adapter instance, after the deployment of the resource adapter or the starting of an application server. Next, some useful instances, such as WorkManager, XATerminator, and Timer are provided to the instance of the resource adapter. Finally, when the application server shuts down or the resource adapter is undeployed, a notification is sent from the application server to the resource adapter.

JCA introduces life cycle management interfaces that consist of methods. The implementation of these methods manages the life cycle of a resource adapter. A resource adapter must implement the `ResourceAdapter` interface to enable the Lifecycle Management contract to control the life cycle of the resource adapter.

Figure 17.4 shows the interfaces of the Lifecycle Management contract in JCA:

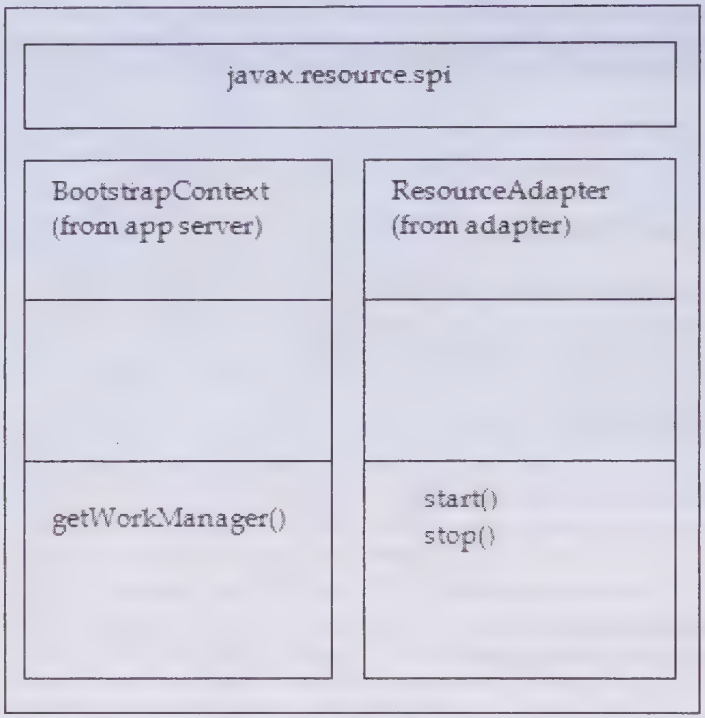


Figure 17.4: Showing the Interfaces in the Lifecycle Management Contract

Figure 17.4 shows the various interfaces needed to manage the life cycle of a resource adapter, which are available in the javax.resource.spi package. The start() and stop() methods of the ResourceAdapter interface allow an application server to start and stop, respectively, the execution of a resource adapter. You can control the process in which a resource adapter submits the instances of the Work interface to an application server for execution. This process is called Workflow management. The workflow of a resource adapter is managed by the instance of the `WorkManager` interface, which is returned by the getWorkManager() method of the BootstrapContext interface.

Figure 17.5 shows the object diagram of the Lifecycle Management contract:

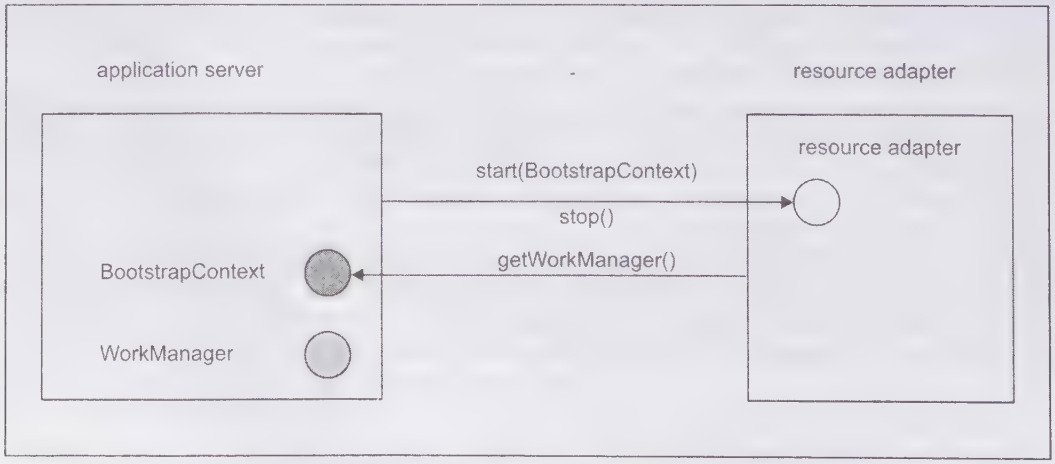


Figure 17.5: Showing the Lifecycle Management Contract by using an Object Diagram

In Figure 17.5, an application server calls the start() method to start the ResourceAdapter instance. The ResourceAdapter instance then uses the getWorkManager() method to access the application server's bootstrap context. The following code snippet shows the methods declared in the ResourceAdapter and BootstrapContext interfaces of JCA:

```
package javax.resource.spi;

import javax.resource.spi.work.WorkManager;

public interface ResourceAdapter {
  void start(BootstrapContext)
  throws ResourceAdapterInternalException;  // startup notification
  void stop(); // shutdown notification
  ... // other operations
}

public interface BootstrapContext{
  WorkManager getWorkManager();
  ... // other operations
}
```

The preceding code snippet shows the start() and stop() methods of the ResourceAdapter interface and the `getWorkManager()` method of the BootstrapContext interface. A resource adapter implements the `ResourceAdapter` interface and provides implementation for the start() method, which accepts the `BootstrapContext` instance as an argument. In addition, the class implementing the `BootstrapContext` interface should provide implementation for the `getWorkManager()` method.

Let's now discuss how an application server bootstraps a ResourceAdapter instance.

Bootstrapping a Resource Adapter Instance

The `ResourceAdapter` interface enables an application server to manage the deployment and undeployment of a resource adapter. A `JavaBean` implements the `ResourceAdapter` interface, which is configured in Deployment Descriptor of the resource adapter. The application server uses Deployment Descriptor to find the `JavaBean` class and calls its two important methods, `start ()` and `stop()` to start and stop, respectively, the execution of a resource adapter.

To deploy a resource adapter, an instance of the resource adapter is bootstrapped in its address space. The application server must use the configured `ResourceAdapter` JavaBean and call the `start()` method to bootstrap a resource adapter instance.

Let's discuss the start() and stop() methods of the ResourceAdapter JavaBean.

The start() Method of the ResourceAdapter JavaBean

An application server calls the `start()` method when you deploy a resource adapter or start an application server. This method initializes an instance of the resource adapter and accepts a parameter from the application server, i.e. an instance of the class that implements the `BootstrapContext` interface. This method also enables the resource adapter to use the services provided by the application server, such as transaction management. When the start() method throws any exception, it means that an error has occurred and the resource adapter is not initialized successfully.

At runtime, a resource adapter instance may contain several objects that may be created and discarded during its lifetime. These objects include the `ManagedConnectionFactory` and `ActivationSpec` JavaBeans, various connection objects, resource adapter private objects, and other resource adapter-specific objects, which can interact with enterprise applications.

Figure 17.6 shows the interaction of the resource adapter instance with the `ManagedConnectionFactory` and `ActivationSpec` JavaBeans:

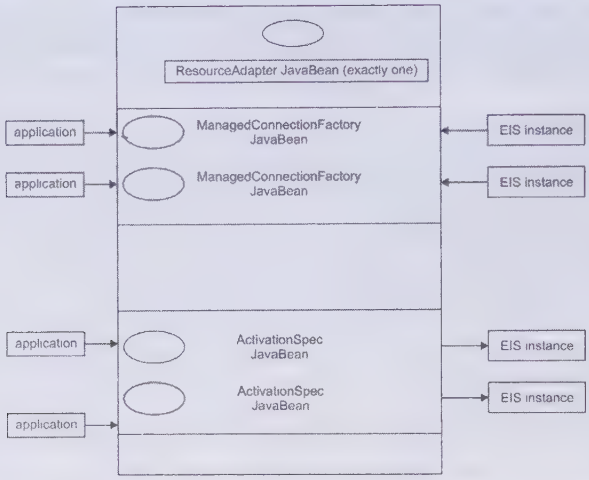


Figure 17.6: Showing Interaction among the Resource Adapter, ManagedConnectionFactory, and ActivationSpec Instances

In Figure 17.6, you can see that an application communicates with the ManagedConnectionFactory instance, which further communicates with the EIS instance to retrieve the required data. When the data is retrieved, the EIS instance communicates with the ActivationSpec instance, which in turn communicates with the application.

The stop() Method of the ResourceAdapter JavaBean

The stop() method of a JavaBean, which implements the ResourceAdapter interface, is invoked to stop the resource adapter functionality. This method is invoked during the shutdown process of the application server or when the resource adapter is undeployed.

Let's now learn about the ManagedConnectionFactory JavaBean that holds the configuration information related to outbound connectivity from an enterprise application component to an EIS instance.

Understanding a ManagedConnectionFactory JavaBean and Outbound Communication

Information about the outbound connectivity between an enterprise application and EIS is represented by the `ManagedConnectionFactory` JavaBean instance through a particular ResourceAdapter instance. The ManagedConnectionFactory JavaBean instance holds the configuration information related to outbound connectivity to an EIS instance. An enterprise application initiates outbound communication with EIS through a resource adapter. The configuration of outbound communication is a combination of the configurations of the `ResourceAdapter` and `ManagedConnectionFactory` JavaBeans. The following code snippet shows a JavaBean that implements the ManagedConnectionFactory and ResourceAdapterAssociation interfaces:

```
import javax.resource.spi.ResourceAdapterAssociation;
import javax.resource.spi.ManagedConnectionFactory;
import java.io.Serializable;
public class ManagedConnectionFactoryImpl implements ManagedConnectionFactory,
ResourceAdapterAssociation, Serializable
{
  ResourceAdapter getResourceAdapter();
  void setResourceAdapter(ResourceAdapter) throws ResourceException;
  ... // other methods
}
```

The association of the `ManagedConnectionFactory` JavaBean with a `ResourceAdapter` JavaBean is specified by the methods of the `ResourceAdapterAssociation` interface. The association between these two JavaBeans is created by calling the `setResourceAdapter()` method on the `ManagedConnectionFactory` JavaBean. You should note that the setResourceAdapter() method should be called only once. If the association is successfully created, the setResourceAdapter() method executes without raising an exception. The association

established between the ManagedConnectionFactory and ResourceAdapter JavaBeans does not change during the lifetime of the ManagedConnectionFactory JavaBean.

Let's now learn about the ActivationSpec JavaBean, which contains the configuration information related to the inbound connectivity from an EIS instance to an enterprise application component.

Understanding an ActivationSpec JavaBean and Inbound Communication

Inbound connectivity is represented by a JavaBean that implements the ActivationSpec interface, which originates from an EIS instance to an enterprise application through a specific resource adapter instance. The ActivationSpec JavaBean contains configuration information pertaining to inbound connectivity from an EIS instance. When an ActivationSpec JavaBean is created, it may inherit the ResourceAdapter JavaBean (which represents the resource adapter instance) configuration information, override specific global default values, if any, and add other configuration information specific to the inbound connectivity.

Inbound communication configuration is a combination of the configuration of the ResourceAdapter and ActivationSpec JavaBeans. This type of communication is initiated by an EIS instance and the communication occurs in the context of a resource adapter thread. You should note that application threads are not involved in such type of communication.

The following code snippet shows the code of the MyActivationSpecImpl class, which implements the ActivationSpec and Serializable interface:

```
import javax.resource.spi.ActivationSpec;
import java.io.Serializable;
// ActivationSpec interface extends ResourceAdapterAssociation interface.
public class MyActivationSpecImpl implements ActivationSpec, Serializable
{
   ResourceAdapter getResourceAdapter();
   void setResourceAdapter(ResourceAdapter) throws ResourceException;
   ... // other methods
}
```

The setResourceAdapter() method of the ActivationSpec JavaBean establishes an association between the ActivationSpec and ResourceAdapter JavaBeans. This method must be called by the application server before the invocation of the other methods of the ActivationSpec JavaBean. A successful association is established only when the setResourceAdapter() method of the ActivationSpec JavaBean is executed successfully without throwing an exception. The setResourceAdapter() method of the ActivationSpec JavaBean is invoked only once. The association between the ResourceAdapter and ActivationSpec instances must not change during the lifetime of the ActivationSpec JavaBean. The reason for this is that the ActivationSpec instance contains the configuration information pertaining to inbound connectivity.

Now, let's learn how to manage the life cycle of a resource adapter.

Managing the Life Cycle of a Resource Adapter

An application server manages the life cycle of a resource adapter. The following are some of the tasks performed by the application server to manage the life cycle of the resource adapter:

- Uses a new ResourceAdapter JavaBean to manage the life cycle of each resource adapter instance and discards the ResourceAdapter JavaBean after its stop () method has been called. In other words, the application server must not reuse the same ResourceAdapter JavaBean object to manage multiple instances of a resource adapter, as the ResourceAdapter JavaBean object may contain resource adapter instance-specific state information.
- Calls the start() method on the ResourceAdapter JavaBean to make a resource adapter instance functional, before accessing other methods on the ResourceAdapter JavaBean instance or before using other objects that belong to the same resource adapter instance.
- Invokes the start() and stop() methods of the ResourceAdapter JavaBean in an unspecified context. To invoke these methods, it is mandatory for the application server to possess the same security permission level as that of the resource adapter instance.

Irrespective of the cause for the shutdown of a resource adapter, an application server uses two phases to shut down the resource adapter. These phases are described in the following sections.

Phase One

An application server must ensure that all dependant applications using a specific resource adapter instance are stopped before calling the `stop` () method by using the instance of the `ResourceAdapter` JavaBean. This includes deactivating all the message endpoints that are receiving messages through a specific resource adapter. However, the dependant applications cannot be stopped until they are undeployed. Therefore, the application server may have to delay destruction of the resource adapter instance till all dependant applications are undeployed. The successful completion of phase one guarantees that the threads of an application will not use the resource adapter instance, even though the instance-specific objects of the resource adapter may still be in the memory heap. This ensures that all application activities, including transactional activities, are completed.

Phase Two

In phase two of the shutdown process of an application server, the `stop()` method is called by an application server thread on the `ResourceAdapter` JavaBean. The invocation of the stop() method notifies the instance of the `ResourceAdapter` JavaBean to stop its functioning and to unload the instance safely. The `ResourceAdapter` JavaBean is responsible for performing a proper shutdown of the resource adapter instance when the `stop()` method is called. Proper shutdown of a resource adapter may include closing network endpoints, relinquishing threads, and releasing all active Work instances. When all the Work instances are released during the shutdown, the resource adapter is allowed to complete the internal in-flight transactions, if the transactions are already in the process of doing a commit and flushing any cached data to EIS. If the `stop()` method throws any unchecked exception, the shutdown process of an application server and the undeployment process of a resource adapter instance are not rolled back. The application server may log or store the information about exceptions that may have occurred when various activities are performed on the server.

Figure 17.7 describes the life cycle of a resource adapter in JCA:

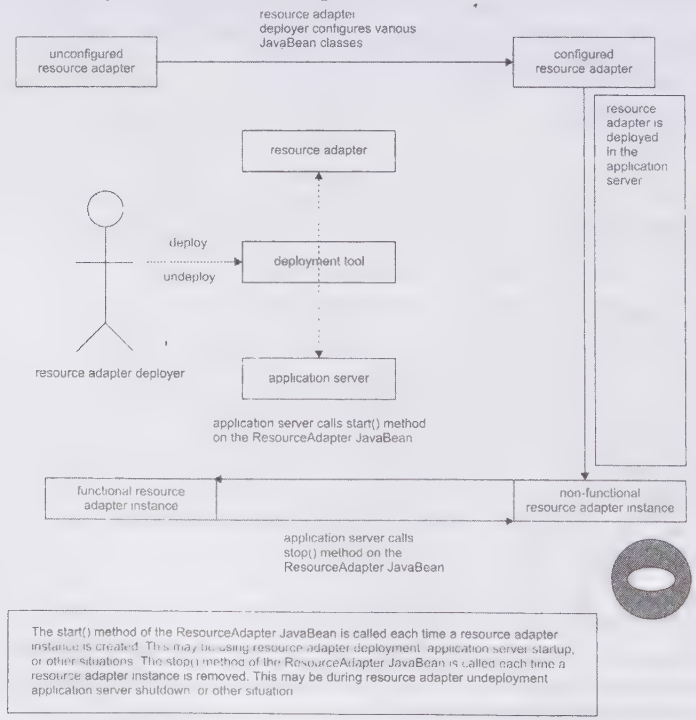


Figure 17.7: Showing the Resource Adapter Life Cycle (State Diagram)

Figure 17.7 shows the state diagram depicting the life cycle of a resource adapter in JCA. The resource adapter deployer maps various JavaBean classes to configure a resource adapter. The configured resource adapter is then deployed on the application server to create the instance of the resource adapter.

NOTE

The instance of the resource adapter is non-functional until the start() method is invoked.

Next, the `start()` method is invoked to make the instance functional or active. Later, when all the necessary tasks are performed by using the instance of the resource adapter, the `stop()` method is invoked on the ResourceAdapter JavaBean. The `stop()` method is generally invoked during the undeployment of the resource adapter instance or shutdown of the application server.

After understanding the life cycle of a resource adapter, let's explore Workflow management.

Exploring Workflow Management

Workflow management refers to the process of monitoring the Work instances submitted by a resource adapter to an application server for their execution. The instances of the `Work` interface can be managed by specifying the Work Management contract between the application server and the resource adapter. Threads are dispatched by the application server to run the submitted Work instances, which in turn prevent a request dispatcher to create or manage threads directly. The submitted Work instances contain the implementation of the tasks to be performed by the resource adapter. The process of Workflow management permits an application server to pool threads efficiently and possess more runtime environment control.

Let's now discuss the advantages of Workflow management.

Listing the Advantages of Workflow Management

There are several advantages of Workflow management, where threads are managed by an application server to execute Work instances. Some of these advantages are briefly described as follows:

- Allows an application server to optimally manage system resources, such as threads. The application server can pool threads and reuse them efficiently across different resource adapters deployed in its runtime environment.
- Allows an application server to control the runtime environment efficiently. Prior to the introduction of Workflow management, the application server did not create its own threads and the resource adapter created non-daemon threads that interfered with the proper shutdown of the application server. With the implementation of Workflow management, the application server owns the threads that are needed to run Work instances.
- Allows an application server to enforce control over the runtime behavior of its system components, including resource adapters. For example, an application server may choose to intercept operations on a thread object, perform checks, and enforce correct behavior.
- Enables an application server to disallow resource adapters from creating their own threads based on the security policy setting of the application server, enforced by a security manager.

The following are the goals of work management:

- Provide a work execution model that controls the thread needs of a resource adapter
- Provide a mechanism for an application server to pool and reuse threads
- Exercise control over thread behavior in a managed environment

Let's now explore the work management model that manages the execution of Work instances.

Describing the Work Management Model

This section helps you to learn how the Workflow management is implemented by using the work management model. In this model, a `WorkManager` instance from a `BootstrapContext` instance is retrieved by a resource adapter when the BootstrapContext instance is passed as an argument to the start () method of the resource adapter, during the deployment of the resource adapter. The resource adapter may create `Work` instances to do

its work and submit them to the `WorkManager` instance along with an optional execution context for the execution of the Work instances. A thread pool, where threads wait for the submission of Work instances, is maintained by an application server. When a Work instance is submitted, a free thread from the thread pool retrieves the Work instance, establishes a suitable executable context, and invokes the run() method. After the `run()` method is executed successfully, the application server can reuse the thread for execution of other Work instances.

To reclaim an active thread, an application server invokes the `release()` method on a particular `Work` instance in a separate thread. This method instructs a resource adapter to release the active thread executing the `Work` instance and performs internal cleanup. A mandatory condition for work management is that same priority level threads must be used by the application server to execute the run() method of the Work instances, which are submitted by a particular resource adapter.

Figure 17.8 shows the interfaces and classes used to manage the Work instances submitted by a resource adapter:

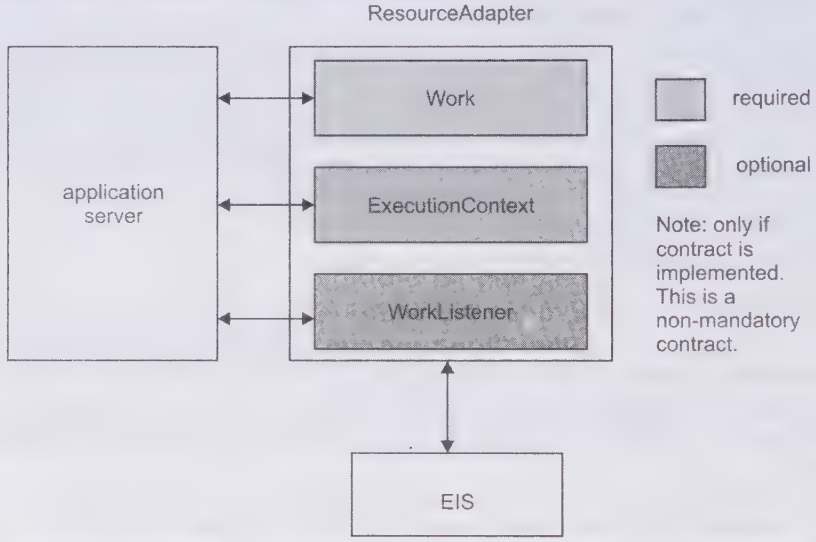


Figure 17.8: Displaying the Interfaces and Classes for Workflow Management

In Figure 17.8, the Work, ExecutionContext, and WorkListener interfaces are used by the application server to interact with EIS. In other words, Workflow management includes some predefined methods and interfaces that are used by the application server to manage Work instances. The BootstrapContext instance available in the Work Management contract is used to deploy the resource adapter in the application server.

Now, let's explore the interfaces used in Workflow management.

Describing the Work Interface

A resource adapter creates Work instances with the help of the `Work` interface. These Work instances are provided to an application server for their execution. Initially, the resource adapter creates the `Work` instances and then submits them to the `WorkManager` instance of the application server. The `WorkManager` instance is obtained from the `BootstrapContext` instance that is passed in the `start()` method at the time of the initialization of the resource adapter. The following code snippet shows the implementation of the methods of the Work interface:

```
public interface Work extends Runnable{
  void release();
    }
```

In the preceding code snippet, the `Work` interface extends the `Runnable` interface; therefore, apart from the `release()` method, the methods of the `Runnable` interface are also extended by the `Work` interface. The following is a brief description of the methods of the `Work` interface:

- The `run()` **method**—Initiates the execution of a `Work` instance. The `WorkManager` instance dispatches a thread that calls the `run()` method of the `Work` instance. The execution of the Work instance is completed when the `run()` method returns, with or without an exception. The `Work` instance treats the calling thread as a Java thread.
- The `release()` **method**—Allows a resource adapter to free a thread. This method is invoked by an application server. You cannot declare the `run ()` and `release()` methods as synchronized methods; however, these methods can have synchronized blocks.

The following code snippet shows the implementation of the MyEISWork class implementing the Work interface:

```
import javax.resource.spi.work;
public class MyEISWork implements Work
{
   void run () { }
   void release()
   {
         // Do clean up necessary on resource adapter so the
         // Application Server thread can be released.
   }
}
```

Next, let's explore the ExecutionContext class that defines the context in which Work instances are executed.

Describing the ExecutionContext Class

The `ExecutionContext` class allows a resource adapter to specify an execution context, such as a transaction context in which the `Work` instance must be executed. The resource adapter populates the `ExecutionContext` instance with an appropriate execution context. The default implementation of the `ExecutionContext` instance provides a null context.

The following code snippet shows the methods defined in the `ExecutionContext` class:

```
public class ExecutionContext
{
   public void setXid(xid) { ... }
   public Xid getXid() { ... }
   public long getTransactionTimeout() { ... }
   public void setTransactionTimeout(long seconds)throws NotSupportedException
   {}
}
```

After exploring the `ExecutionContext` class, let's explore the `WorkListener` interface, which listens to various events, such as accepting, rejecting, and starting the Work instances as well as completing the execution of Work instances.

Describing the WorkListener Interface

The `WorkListener` interface is optionally implemented by a resource adapter. When a resource adapter submits a `Work` instance, an instance of the class that implements the `WorkListener` interface is also passed. After the instance is passed, the resource adapter must provide updates corresponding to the Work instance of the class implementing the `WorkListener` interface, through application server threads. The `WorkListener` interface listens to the notifications for accepting, rejecting, and starting the Work instance as well as completing the execution of the Work instance. The class implementing the `WorkListener` interface provides thread safety and allows variation in the sequence of notifications based on your requirements.

The following code snippet shows the code of the WorkListener interface:

```
public interface WorkListener extends EventListener
{
   void workAccepted(workEvent);
   void workRejected(WorkEvent);
   void workStarted(WorkEvent);
   void workCompleted(WorkEvent);
}
```

The following code snippet shows the code of the class implementing the `WorkListener` interface:

```
import javax.resource.spi.work;
public class MyWorkListener implements WorkListener
{
   void workAccepted (WorkEvent e){…}
   void workRejected(WorkEvent e) {…}
   void workStarted(WorkEvent e) {…}
   void workCompleted(WorkEvent e) {…}
}
```

Next, let's explore the `WorkEvent` class, which encapsulates various events that occur during the execution of a Work instance.

Describing the WorkEvent Class

The `WorkEvent` class encapsulates the events that take place when a `Work` instance is executed. An instance of the `WorkEvent` class performs the following functions:

- Provides the type of the event that occurs during the execution of a Work instance
- Provides source objects, i.e., the Work instance on which an event initially occurs
- Accesses the Work instance associated with the event
- Provides an interval in milliseconds with which the execution of a Work instance is delayed
- Provides probable exceptions that are thrown during work processing, such as `WorkRejectedException` and `WorkCompletedException`

The following code snippet shows the implementation of the methods of the `WorkEvent` class:

```
public class WorkEvent extends EventObject
{
   public static final int WORK_ACCEPTED = 1;
   public static final int WORK_REJECTED = 2;
   public static final int WORK_STARTED = 3;
   public static final int WORK_COMPLETED = 4;
   public WorkEvent(Object source, int type, Work work,
   WorkException exc) {...}
   public WorkEvent(Object source, int type, Work work,
   WorkException exc, long startDuration) {...}
   public int getType() { ... }
   public Work getWork() { ... }
   public long getStartDuration() { ... }
   public WorkException getException() { ... }
}
```

Now, let's explore the WorkAdapter class, which is used to override the methods of the WorkListener interface.

Describing the WorkAdapter Class

A developer can create a class that extends the `WorkAdapter` class, which implements the `WorkListener` interface. This allows the developer to override the required methods of the `WorkListener` interface in the class extending the `WorkAdapter` class. The following code snippet shows the implementation of the methods of the `WorkListener` interface in the `WorkAdapter` class:

```
public class WorkAdapter implements WorkListener
{
   public void workAccepted(WorkEvent e) {}
   public void workRejected(WorkEvent e) {}
   public void workStarted(WorkEvent e) {}
   public void workCompleted(WorkEvent e) {}
}
```

In the preceding code snippet, the `WorkAdapter` class implements the `WorkListener` interface; thereby, providing body to all the methods of the interface.

You know that in Java, JDBC is used to establish connections to a data source. A similar functionality is also provided by JCA. In the following section, we differentiate the implementation and working of both these technologies in establishing connections.

Exploring the Differences between JDBC and JCA

JCA and JDBC have an identical mechanism for establishing a connection to a data source and managing transactions. JDBC provides drivers and a client API to connect and integrate with relational database applications; whereas, JCA provides resource adapters and the CCI API to enable seamless integration with EIS resources. Resource adapters for EIS systems correspond to JDBC drivers for relational databases.

Table 17.2 differentiates JDBC and JCA:

Table 17.2: Differences between JDBC and JCA

Points of Difference	JDBC	JCA
API	Defines a standard Java API to access relational databases	Uses CCI to provide an EIS-independent API
Connection	Uses JDBC drivers specific to a relational database management system (RDBMS).	Uses an EIS resource adapter to interact with EIS
Definition	Provides a generic interface to interact with relational databases	Provides a standard architecture for Java EE and Java applications to integrate and interact with EIS resources
Java Transaction API (JTA) support	Provides an interface to support JTA and Java Transaction Service (JTS)	Supports JTA by providing a contract between the Java EE transaction manager and the EIS resource manager
Server implementation	Allows Java EE servers to implement the JDBC connection pool mechanism to create a connection to a database	Allows Java EE servers to implement JCA service contracts to establish a connection factory and manage connections with EIS
Service Provider Interface (SPI) support	Provides support (since JDBC 3.0) for the JCA SPI, which is a contract between an application server and a resource adapter.	Defines SPI to integrate application server services, such as transactions, connections, and security
Support for non-managed applications	Supports non-managed applications that use JDBC	Supports non-managed applications that use CCI
Transaction support	Supports eXtended Architecture (XA) and non-XA transactions with underlying XA data sources	Supports XA and non-XA transactions with underlying EIS resources

After differentiating the JDBC and JCA technologies, let's explore the inbound communication model, which allows an enterprise application component to get information or messages from EIS, with the help of a resource adapter.

Exploring the Inbound Communication Model

You can learn about the inbound communication model only after understanding the communication types supported by a resource adapter. These two types of communication can be explained as follows:

- **Synchronous communication**—Refers to a direct communication process in which all parties engaged in the process are available at a single point of time. Examples of synchronous communication are telephonic communications and board meetings.
- **Asynchronous or event notification-based communication**—Refers to a process in which all parties may not be available at a single point of time. Examples of asynchronous communication are sending messages through mobile phones and blogging.

In JCA, inbound communication is maintained between EIS and an enterprise application component through a resource adapter. Both these types of communication processes are supported by the resource adapter. In case of synchronous communication, the message sent by EIS to the enterprise application component waits for the response from the application. However, in case of asynchronous communication, the message sent by EIS to the enterprise application component does not wait for the response from the application. In other words, it is not essential for the response to be sent to EIS in the same exchange when the message is sent to the application.

In the inbound communication model, all communication to an enterprise application is started by EIS. In this communication model, an enterprise application may be a combination of EJBs (session, entity, and singleton), which is contained in an EJB container. To enable inbound communication, you need a mechanism to invoke the EJBs from a resource adapter as well as propagate transaction information from EIS to the enterprise application residing in the EJB container.

In the following section, we discuss a situation that explains the flow of a message from EIS to an enterprise application component in the inbound communication model.

Discussing a Scenario using Inbound Communication

In the inbound communication model, EIS sends a message to a resource adapter that forwards the message to the enterprise application component by using the contracts defined for inbound communication.

Figure 17.9 shows a diagrammatic representation of the inbound communication model used to send a message:

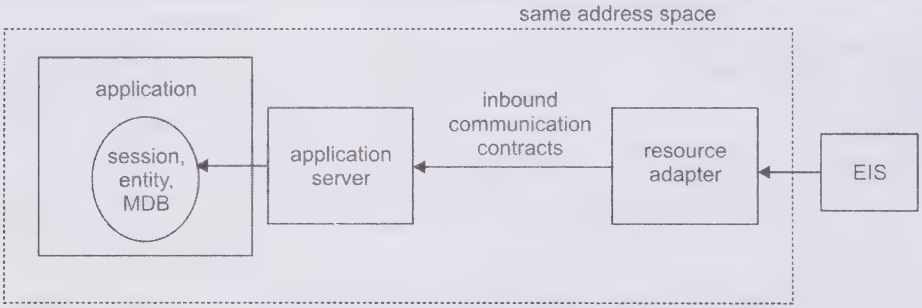


Figure 17.9: Showing the Structure of the Inbound Communication Model

In Figure 17.9, you can see that communication with an enterprise application starts from EIS.

The steps involved in the inbound communication are as follows:

1. EIS sends a message to a resource adapter.
2. The resource adapter receives the message and identifies its type.
3. The `Work` object is created by the resource adapter and submitted to the `WorkManager` object that executes the submitted `Work` object in a separate thread. After submitting the `Work` object, the resource adapter continues waiting for other incoming messages.
4. The resource adapter, based on the message type, looks up the name of the message endpoint in the message endpoint factory. A message endpoint is a proxy of an Message Driven Bean (MDB) instance available in the MDB pool.
5. The resource adapter creates a message endpoint by using the message endpoint factory.
6. The resource adapter calls the onMessage() method on the message endpoint and passes the content of the message received from EIS to the message endpoint.
7. On the invocation of the onMessage() method, an MDB (message endpoint) manages a message in the following ways:
   - Handles the message directly and returns a response to EIS through a resource adapter
   - Sends the message to some other application component
   - Stores the message in a queue, from where it is picked up by a client
   - Communicates directly with the client application

Let's now explore how message inflow works in inbound communication.

Exploring Message Inflow from EIS to Resource Adapter

The inflow of a message through a resource adapter is based on the concept of inbound communication, which provides a standard, generic contract between a resource adapter and an application server. This standard contract is known as the Message Inflow contract, which allows the resource adapter to transfer messages asynchronously or synchronously to message endpoints contained in the application server. In the message inflow process, a message is sent by EIS to a resource adapter, which further transfers the message to a message listener.

A resource adapter uses a communication protocol to establish a connection to EIS and a proprietary communication channel to communicate with EIS. The use of a communication protocol and a channel is an internal process and is not visible to the application server on which the resource adapter is deployed. In addition, the resource adapter also uses a dispatching mechanism to transfer a particular type of message to an application component in the application server.

NOTE

The messaging infrastructure and semantics used to deliver messages do not affect message inflow.

In addition, message inflow provides a standard, generic mechanism to plug in multiple message providers in an application server. An example of this integration is Java Message Service (JMS) that is plugged in the application server through a resource adapter. These message providers send messages to message endpoints that allow Java EE components to use the messages without being aware that a resource adapter is being used to deliver the messages.

Let's now learn about the different ways a resource adapter uses to handle inbound messages.

Explaining the Ways to Handle Inbound Messages

A resource adapter can use any of the following ways to handle inbound messages:

- Manage the messages locally, within the resource adapter bean, without involving other enterprise application components.
- Propagate the messages to another enterprise application component. For example, a resource adapter can look up an EJB component and call its method.
- Send the messages to a message endpoint, which is usually an MDB.

Now, let's discuss the proprietary communication protocol and channel, which are used by a resource adapter in the message inflow process.

Understanding the Proprietary Communication Protocol and Channel

A deployer provides configuration details, such as the communication channel or protocol to be used by a resource adapter to establish a connection between EIS and an enterprise application component. Apart from configuring the resource adapter, the deployer also deploys the resource adapter on an application server. One way to configure the resource adapter is to provide the configuration details in the properties represented by the `ResourceAdapter` bean or the `ActivationSpec` object. Another way of configuring the resource adapter is to provide the same communication channel for both inbound and outbound connectivity.

After learning about the protocol and channel used for message inflow, let's understand the message inflow model that will help to further clarify the concept of message inflow.

Understanding the Message Inflow Model

The message inflow model explains how a message is delivered from a resource adapter to a message listener or an MDB. In other words, this model explains the process of message inflow between an application server and a resource adapter in asynchronous or synchronous manner.

Figure 17.10 shows the message inflow model:

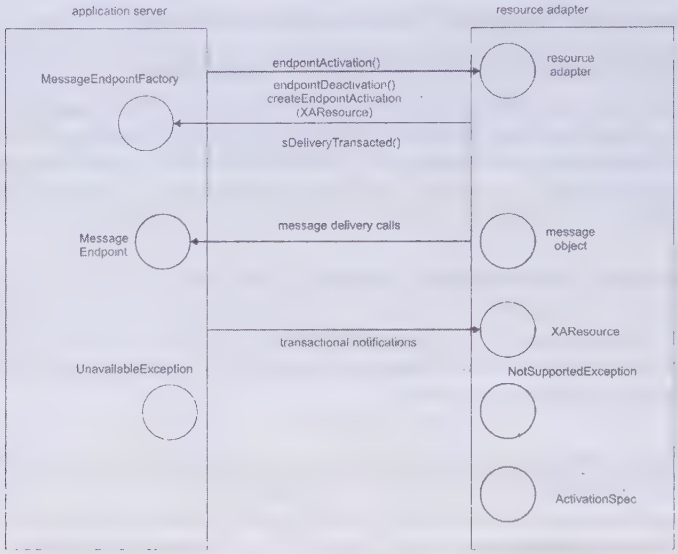


Figure 17.10: Showing the Message Inflow Model (Object Diagram)

As shown in Figure 17.10, the endpointActivation() or endpointDeactivation() method is invoked to activate or deactivate a message endpoint, respectively. During the activation or deactivation of the message endpoint, the MessageEndpointFactory instance is provided to the resource adapter and the ActivationSpec instance is configured. The MessageEndpointFactory instance is used by the resource adapter to obtain the message endpoint to deliver messages synchronously or asynchronously. You should note that the createEndpoint() method may throw the UnavailableException exception due to any of the following reasons:

- Activation of an endpoint is not completed by the application server
- Concurrent message deliveries may be limited by the application server
- Shut down of the application server is in progress
- Internal error condition may have occurred on the application server

As shown in Figure 17.10, the `isDeliveryTransacted()` method can also be invoked by using the MessageEndpointFactory instance to verify whether or not the message deliveries can be transacted.

The methods and instances discussed in the message inflow model are provided by the javax.resource.spi and javax.resource.spi.endpoint packages. Let's now explore the classes and interfaces of these packages.

Describing the javax.resource.spi Package

The `javax.resource.spi` package contains the classes and interfaces, such as ActivationSpec, ResourceAdapter, InvalidPropertyException, and UnavailableException that are used in the message inflow model. You have learned about the use of the endpointActivation() and endpointDeactivation() methods in the preceding section. These methods are also provided by the ResourceAdapter interface defined in the `javax.resource.spi` package.

The following code snippet shows the noteworthy classes and interfaces of the `javax.resource.spi` package:

```
package javax.resource.spi;
import java.beans.PropertyDescriptor;
import javax.resource.NotSupportedException;
import javax.resource.spi.endpoint.MessageEndpointFactory;
public interface ResourceAdapter
{
   ... // other methods
   public void endpointActivation(MessageEndpointFactory
   , ActivationSpec) throws ResourceException;
```

```
    public void endpointDeactivation(MessageEndpointFactory
    , ActivationSpec);
    public XAResource[] getXAResources(ActivationSpec[] specs) throws
    ResourceException;
}

public interface ActivationSpec
{
    // JavaBean
    public void validate() throws InvalidPropertyException;
}

public class InvalidPropertyException extends ResourceException
{
    public InvalidPropertyException() { ... }
    public InvalidPropertyException(String message) { ... }
    public InvalidPropertyException(String message,String errorCode) { ... }
    public void setInvalidPropertyDescriptors( PropertyDescriptor[]
    invalidProperties) { ... }
    public PropertyDescriptor[] getInvalidPropertyDescriptors() {... }
}

public class UnavailableException extends ResourceException
{
    public UnavailableException() { ... }
    public UnavailableException(String message) { ... }
    public UnavailableException(Throwable cause) { ... }
    public UnavailableException(String message, Throwable cause) {... }
}
```

Next, let's explore the javax.resource.spi.endpoint package, which provides classes and interfaces to create a message endpoint that acts as a proxy for an MDB.

Describing the javax.resource.spi.endpoint Package

A resource adapter needs to create a message endpoint before delivering a message to that endpoint. To do this, the resource adapter invokes the createEndpoint() method of the `MessageEndpointFactory` interface, present in the `javax.resource.spi.endpoint` package and then calls the `onMessage()` method of an MDB through the message endpoint to process the message. The javax.resource.spi.endpoint package is implemented by an application server.

The following code snippet shows the noteworthy interfaces defined in the javax.resource.spi.endpoint package:

```
package javax.resource.spi.endpoint;
import java.lang.Exception;
import java.lang.Throwable;
import java.lang.NoSuchMethodException;
import javax.transaction.xa.XAResource;
import javax.resource.ResourceException;
import javax.resource.spi.UnavailableException;
public interface MessageEndpointFactory
{
    MessageEndpoint createEndpoint(XAResource) throws UnavailableException;
    boolean isDeliveryTransacted(java.lang.reflect.Method) throws
    NoSuchMethodException;
}
public interface MessageEndpoint
{
    void beforeDelivery(java.lang.reflect.Method) throws NoSuchMethodException,
    ResourceException;
    void afterDelivery() throws ResourceException;
    void release();
}
```

After the message is sent by an application server to a resource adapter, the adapter delivers the message to a message endpoint. Let's now learn the message inflow process from the resource adapter to the message endpoint.

Exploring the Message Inflow to Message Endpoints

Various entities, such as the resource adapter, application server, and deployer play an important role to deliver messages to a message endpoint. A message provider or third party provides a resource adapter that is capable of sending or delivering messages. In addition, the application server provides a runtime environment for the deployment and execution of an application.

Let's first recall what you mean by the message endpoint. The message endpoint is a proxy to an MDB that is deployed on an application server, which asynchronously or synchronously uses the messages provided by message providers. You should note that the developer creating an MDB must provide various activation configuration details related to the style of the message and to the message provider. These configuration details are provided in Deployment Descriptor of the MDB and are used to activate the MDB (message endpoint).

A message delivery to an MDB should contain the following elements:

- Specific type of inbound message
- The ActivationSpec object implemented by a resource adapter
- Configuration details, such as those related to mapping of message types to message listener interfaces
- An MDB implementing a message listener interface
- The MDB that is bound to a resource adapter

Let's now learn how you can bind an MDB to a resource adapter.

Binding an MDB and a Resource Adapter

You can deploy a resource adapter either independently, i.e. as a standalone RAR file or as a part of an EAR file. Similarly, you can deploy an MDB independently (as a standalone JAR file). It can also be a part of an EAR file. In each case, an MDB must be bound to a resource adapter.

You should perform the following steps to bind an MDB and a resource adapter:

1. Configure the `jndi-name` element in the `sun-ra.xml` Deployment Descriptor for a resource adapter.
2. Configure the `adapter-jndi-name` element in the `sun-ejb-jar.xml` Deployment Descriptor with a value similar to the value set in the corresponding `jndi-name` element.
3. Deploy an MDB only after the resource adapter is deployed on an application server. The bean that represents the `ResourceAdapter` instance is bound to the adapter with a Java Naming and Directory Interface (JNDI) name that is indicated at the time of the deployment of the resource adapter.
4. Allow an MDB container to use an application server-specific API, which looks up the resource adapter with the help of its JNDI name, after the deployment of the MDB. Now, the application server-specific API calls the `endpointActivation(MessageEndpointFactory,ActivationSpec)` method on the resource adapter instance.
5. Provide permission to the resource adapter to use a configured `ActivationSpec` instance (containing configuration information) provided by the MDB container. The resource adapter also possesses a factory of message endpoints by which you can create instances of the message endpoint.

The configured information is saved by the resource adapter and used later to deliver a message to the appropriate message endpoint. The resource adapter contains information about the `MessageListener` interface that is implemented by an MDB. This information is used to determine the kind of messages required to be delivered to the MDB.

Let's learn how a resource adapter dispatches a message to an MDB.

Dispatching a Message

On receiving a message from EIS, a resource adapter determines where the message is to be dispatched. The sequence of events to dispatch a message is as follows:

1. A message comes from EIS to a resource adapter.

2. The resource adapter identifies the type of the message by looking up the message in an internal table located in an application server. The type of the messages is determined with the help of a particular pair of the `MessageEndpointFactory` and `ActivationSpec` instances.
3. The resource adapter, based on the type of the message, decides whether or not the message should be sent to an MDB.
4. The `MessageEndpointFactory` interface decides the message endpoint type. Then, the resource adapter creates an MDB instance by invoking the `createEndpoint()` method on the `MessageEndpointFactory` instance.
5. The resource adapter invokes the `onMessage()` method on the MDB instance and passes the required information, such as the body of an incoming message, to the MDB.
6. If a message listener does not return a value, it is assumed that the message dispatching process has been successfully completed.
7. If the message listener returns a value, this value is handled by the resource adapter.

This ends our discussion on the process of message inflow and how a message is transferred from EIS to a message endpoint. You have also learned about the classes and interfaces that provide the methods used to implement the message inflow model. Apart from this, you also need to understand the activation specifications for a resource adapter. In other words, the resource adapter has an instance of the ActivationSpec class that provides support for each type of message.

Exploring Activation Specifications (JavaBean)

A resource adapter gives a class name for each `ActivationSpec` JavaBean. The resource adapter also provides a class name for each supported message listener type. A JavaBean providing the activation specifications implements the ActivationSpec interface defined in the `javax.resource.spi` package. The configuration information required to activate an endpoint is used during deployment and is provided in Deployment Descriptor (ra.xml). During configuration, an `ActivationSpec` JavaBean instance may check the validity of the configuration settings provided by the deployer for a message endpoint serving as a proxy for an MDB. To check the overall activation configuration information given by the deployer, the `validate()` method of the `ActivationSpec` JavaBean is used while deploying an application on the endpoint.

Let's learn about administered objects that are needed to perform various tasks. For example, the Connection administered object helps to connect an application server to EIS.

Exploring Administered Objects

A resource adapter gives a Java class name and an interface type to an optional set of JavaBean classes that defines administered objects. These classes may represent various administered objects, which are specific to the style of a message or to a message provider. Enterprise applications may use special administered objects to synchronously send and receive messages through connection objects by using APIs, which are specific to the messaging style. Moreover, the administered objects are not used to set up asynchronous message deliveries to message endpoints. However, the `ActivationSpec` JavaBean is used to hold all the necessary activation information needed for an asynchronous message delivery setup. You can configure administered objects in the admin-objects elements of the `ra.xml` and `sun-ra.xml` Deployment Descriptor files.

You can define administered objects in three configuration scope levels similar to outbound connections and other GlassFish resource adapter configuration elements. The three configuration scopes are as follows:

- **Global**—Represents the scope that is acceptable for all administered objects in a resource adapter, which are defined in the ra.xml Deployment Descriptor by using the `default-properties` element.
- **Group**—Represents the scope that is acceptable for all administered objects in a resource adapter, which are defined in the ra.xml Deployment Descriptor using the `admin-object-group` element. The administered objects defined in the group level possess properties that override any property defined at the global level.
- **Instance**—Represents specific administered objects for a resource adapter. Instance-level properties can be defined at the instance level by using the `admin-object-properties` element in the ra.xml Deployment Descriptor. Instance-level properties override group and global level properties.

Now, let's learn how a resource adapter invokes a session or an entity bean.

Understanding EJB Invocation

This section discusses the process of invoking session and entity beans from a resource adapter. Session or entity beans are invoked by a resource adapter to perform the following two main tasks:

- Dispatch the calls from EIS to a bean in order to execute the business logic of an enterprise application
- Use the EJB Container-Managed Persistence (CMP) mechanism for persistence

Note that to dispatch calls to a session or entity bean, the resource adapter needs to know the target bean type and the name as well as parameters of the method to be invoked. When the resource adapter receives a request from EIS through a remote protocol, the adapter selects an appropriate bean and a target method name. In addition, the resource adapter deserializes the request parameters received from EIS and calls the target bean method.

To invoke a session or an entity bean, the resource adapter can use the bean client view model through an MDB. The EJB 3.0 specification defines the EJB client view and describes how the client view is used to access session or entity beans. The EJB client view is available to an MDB.

Bean dispatch logic is implemented in an MDB. The MDB selects an appropriate session or entity bean and a target method, deserializes the request parameters, and invokes the selected bean based on the request information received from the EIS system. The resource adapter uses the Message Inflow contract to invoke an MDB and then uses the invoked MDB through the EJB client view model to dispatch or send calls to the appropriate session and entity beans.

You can say that an MDB can be used as a replaceable unit of the resource adapter, which plays the role of a bean dispatcher. The Message Inflow contract allows the creation of multiple endpoint instances (MDBs) at runtime. This facilitates the concurrent dispatching of messages to an MDB.

Figure 17.11 shows the EJB invocation model with a resource adapter:

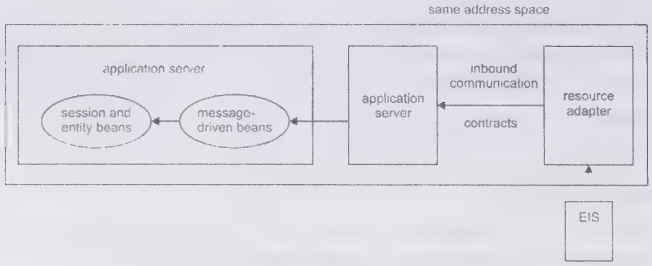


Figure 17.11: Showing the EJB Invocation Model

In Figure 17.11, an MDB is used along with the resource adapter. The resource adapter invokes the MDB to transfer a message to the application server.

Let's now discuss how an MDB is invoked by using the resource adapter in an enterprise application.

A resource adapter supports inbound communication from an EIS system to the application components residing in an application server container. The resource adapter uses the Message Inflow contract to call MDBs that serve as a dispatcher for session and entity bean invocations. The resource adapter and MDBs may be provided by separate or the same resource adapter vendor. An example of a resource adapter providing both the resource adapter and the MDBs is Wombat Inc.

The EIS system uses multiple concurrent conversations in its interactions with a resource adapter. Each conversation may involve multiple serial requests. The resource adapter has a set of `Work` objects (threads), each of which is used to carry on a specific conversation. The resource adapter manages the multiple concurrent conversations and calls an MDB instance whenever a request message arrives as part of a conversation.

The following code snippet shows the implementation of MDB with a resource adapter:

```
import javax.ejb.MessageDrivenBean;
import javax.naming.InitialContext;
```

```
public class WombatMDB
implements MessageDrivenBean, WombatMessageListener
{
   public static int myCONV_START = 0;
   public static int myCONV_CONTINUE = 1;
   public static int myCONV_END = 2;
   private Context myjndiContext = null;
   private ConvBeanHome mychome = null;
   public void ejbCreate()
   {
         myjndiContext = new InitialContext();
         mychome = (ConvBeanHome)
         jndi.lookup("java:comp/env/ConvBeanHome");
   }

   ConvResponse onMessage(ConvRequest requestMsg) {
         // get conversation id and request type from the request
         // message
         int convId = ...;
         int type = ...;

         if (type == myCONV_START)
         {
                // create Entity EJB for holding the specific
                // conversation state
                ConvBean cbean = mychome.create(convId);
         }
         else if (type == myCONV_CONTINUE)
         {
                ConvBean cbean = mychome.findByPrimaryKey(convId);
                // Unmarshal EJB method parameters
                ...;
                // invoke EJB and return response
                Object resp = cbean.myBusinessMethod(params);
                ConvResponse cresp = Utility.convert(resp);
                return cresponse;
         }
         else if (type == myCONV_END)
         {
                cbean.remove();
         }
         return null;
   }
   public void ejbRemove()
   {
         myjndiContext = null;
         mychome = null;
   }
}
```

The resource adapter uses an MDB as a generic dispatcher for session and entity bean invocations and relies on the application server to efficiently pool MDB instances. When a thread from a resource adapter accesses an MDB method, the JNDI context of the MDB is available to the thread. The resource adapter may take advantage of this and use the MDB as a dispatcher. The bean method may be used in a while loop by a resource adapter. The thread in the bean's while loop is used to process specific data structures, such as Queue and List of the resource adapter. The instances of these data structures are passed to the bean method as parameters. In such a case, the bean becomes a special Java object that has access to the JNDI context, which the resource adapter may use. This usage pattern illustrates a tight coupling between the resource adapter and the MDB.

After understanding the concept of EJB invocation, let's discuss the CCI API.

Understanding the CCI API

The classes and interfaces included in the CCI API provide the methods that allow enterprise application components to access data across heterogeneous EIS systems, such as database systems, legacy applications

coded in other programming languages, and ERP. By using the CCI API, EAI frameworks and application components can communicate across heterogeneous EIS systems. Some advantages of the CCI API are as follows:

- Enhances application development tools and EAI frameworks.
- Provides a remote function-call interface that can be used to execute EIS application functions and retrieve their results. The CCI API provides the basic classes and interfaces to connect to EIS.
- Serves as a simple functionality that component developers may use to develop complex enterprise applications. It also provides an extensible application programming model.
- Provides classes and interfaces that can be used in the Java EE and J2SE platforms.
- Provides EIS independence that can be derived by EIS-specific metadata present in the repository.

Let's now briefly discuss the interfaces and classes of CCI.

The CCI API contains the following interfaces:

- `Connection-related` **interfaces**—Represent a connection factory and an application-level connection. The following code snippet lists the connection-related interfaces along with their packages:

```
javax.resource.cci.ConnectionFactory
javax.resource.cci.Connection
javax.resource.cci.ConnectionSpec
javax.resource.cci.LocalTransaction
```

- `Interaction-related` **interfaces**—Allow an enterprise application component to maintain an interaction that is determined by the `InteractionSpec` interface by using an EIS instance. The following code snippet lists the interaction interfaces available in the CCI API:

```
javax.resource.cci.Interaction
javax.resource.cci.InteractionSpec
```

- `Data representation-related` **interfaces**—Represent the data structures that are used in an interaction with an EIS instance. The following code snippet lists the data representation-related interfaces along with their packages:

```
javax.resource.cci.Record
javax.resource.cci.MappedRecord
javax.resource.cci.IndexedRecord
javax.resource.cci.RecordFactory
javax.resource.cci.Streamable
javax.resource.cci.ResultSet
```

- `Metadata-related` **interfaces**—Specifies metadata information for a resource adapter implementation and an EIS connection. The following code snippet lists the metadata-related interfaces along with their packages:

```
javax.resource.cci.ConnectionMetaData
javax.resource.cci.ResourceAdapterMetaData
javax.resource.cci.ResultSetInfo
```

- **Additional classes**—Provide the `ResourceException` and `ResourceWarning` exception classes used to handle an exception raised with respect to a resource adapter. The following code snippet lists the additional classes that are available in the CCI API:

```
javax.resource.ResourceException
javax.resource.cci.ResourceWarning
```

Now, let's discuss the noteworthy interfaces of the CCI API that are needed to connect an application server with heterogeneous EIS systems.

The ConnectionFactory Interface

The `javax.resource.cci.ConnectionFactory` interface allows you to establish a connection with an EIS instance through a resource adapter. An EIS resource adapter usually provides this interface. The following code snippet shows the methods of the `ConnectionFactory` interface:

```
public interface javax.resource.cci.ConnectionFactory implements
java.io.Serializable, javax.resource.Referenceable
{
  public RecordFactory getRecordFactory() throws ResourceException;
```

```
    public Connection getConnection() throws ResourceException;
    public Connection getConnection(
    javax.resource.cci.ConnectionSpec properties)
    throws ResourceException;
    public ResourceAdapterMetaData getMetaData() throws ResourceException;
}
```

The methods used in the preceding code snippet can be briefly described as follows:

- `getConnection()`—Retrieves a connection to an EIS instance. In the case of container-managed sign-on, no security information is passed by an enterprise application component. If you need to pass any resource adapter-specific security information and connection parameters, the enterprise application component may decide to use a getConnection variant with the javax.resource.cci.ConnectionSpec type of parameter.
- `getRecordfactory()`—Creates the `RecordFactory` instance.
- `getMetaData()`—Retrieves metadata information about the ResourceAdapter instance.

The ConnectionSpec Interface

Application components use the `javax.resource.cci.ConnectionSpec` interface to provide a connection request and specific properties to the getConnection() method of the ConnectionFactory interface. Parameters, such as user name and password, are passed to the getConnection() method of the ConnectionFactory interface, while establishing a connection to the EIS system. The following code snippet shows the declaration of the ConnectionSpec interface:

```
public interface javax.resource.cci.ConnectionSpec {}
```

Let's now describe the standard properties for the ConnectionSpec interface.

Table 17.3 shows the properties of the ConnectionSpec interface:

Table 17.3: Properties of the ConnectionSpec Interface

Property	Description
UserName	Specifies the name of a user connecting to an EIS instance
Password	Specifies the password for a user connecting to an EIS instance

The Connection Interface

The `javax.resource.cci.Connection` interface represents a connection to an EIS resource and is used to perform various operations with an underlying EIS. The following code snippet lists the methods of the Connection interface:

```
public interface javax.resource.cci.Connection
{
    public Interaction createInteraction() throws ResourceException;
    public ConnectionMetaData getMetaData() throws ResourceException;
    public ResultSetInfo getResultSetInfo() throws ResourceException;
    public LocalTransaction getLocalTransaction() throws ResourceException;
    public void close() throws ResourceException;
}
```

The methods used in the preceding code snippet can be briefly described as follows:

- `createInteraction()`—Creates an instance of the Interaction interface with reference to the Connection instance. This method is invoked on the instance of the Construction interface and allows an enterprise application component to access EIS functions and data.
- `getMetaData()`—Provides information about the EIS instance associated with a `Connection` instance. In other words, you can use the `ConnectionMetaData` interface to find EIS instance-specific information.
- `getResultSetInfo()`—Returns information about the ResultSet supported by the connected EIS system to a message listener.
- **`close()`**—Closes an established connection and destroys the instance of the Connection interface.

The Interaction Interface

The `javax.resource.cci.Interaction` interface interacts with a connected EIS system to perform various operations, such as retrieving data from an EIS system. The following code snippet defines the methods of the `Interaction` interface:

```
public interface javax.resource.cci.Interaction
{
  public Connection getConnection();
  public void close() throws ResourceException;
  public boolean execute(InteractionSpec ispec,Record input,Record output)
  throws ResourceException;
  public Record execute(InteractionSpec ispec, Record input) throws
  ResourceException;
  ...
}
```

An `Interaction` instance is used to perform the following tasks with an EIS instance:

- Update the output record and returns a Boolean value. The first `execute()` method defined in the preceding code snippet accepts three parameters: an input record, output record, and an `InteractionSpec` instance. The EIS functions specified by the `InteractionSpec` interface are also executed by the `first execute()` method.
- Produce and return an output record. The second `execute()` method defined in the preceding code snippet accepts two parameters, an input record and an `InteractionSpec` instance. The EIS function specified by the `InteractionSpec` interface is also executed by the `second execute()` method.

The InteractionSpec Interface

The InteractionSpec interface defines interactions by providing EIS-specific object properties, such as data types and schema, which are associated with an EIS system. The following code snippet shows the methods defined in the `InteractionSpec` interface:

```
public interface javax.resource.cci.InteractionSpec implements java.io.Serializable
{
  // Standard Interaction Verbs
  public static final int MYSYNC_SEND = 0;
  public static final int MYSYNC_SEND_RECEIVE = 1;
  public static final int MYSYNC_RECEIVE = 2;
}
```

The LocalTransaction Interface

The `javax.resource.cci.LocalTransaction` interface represents a local transaction (similar to the `UserTransaction` interface). Local transaction is controlled internally in a resource manager. The following code snippet shows the methods defined in the `LocalTransaction` interface:

```
public interface javax.resource.cci.LocalTransaction
{
  public void begin() throws ResourceException;
  public void commit() throws ResourceException;
  public void rollback() throws ResourceException;
}
```

If CCI implementation supports the `LocalTransaction` interface, the `Connection.getLocalTransaction` method returns a `LocalTransaction` instance. To perform a local transaction on an underlying EIS instance, the instance of the `LocalTransaction` interface is usually used by the enterprise application component. A resource adapter is allowed to implement the `javax.resource.spi.LocalTransaction` interface while it does not implement the application-level `javax.resource.cci.LocalTransaction` interface.

Now, let's explore JCA exceptions.

Exploring JCA Exceptions

JCA provides exceptions to handle unexpected circumstances in implementing application components. JCA provides the following two types of exceptions:

- The application exception
- The system exception

Let's describe both these exceptions in the following sections.

The Application Exception

An application exception is thrown whenever application components access an EIS resource. An application exception reports an error in the execution of a function on a target EIS. An application component is capable of handling these exceptions directly.

The System Exception

System exceptions are unexpected error conditions that may be thrown by an invocation of a method of system contracts, such as transaction management-related errors. These exceptions are handled by an application server or a resource adapter, depending on cause of the exception. You should note that the system exceptions are not reported in their original form directly to an application component.

Next, let's discuss the various system exceptions in JCA.

ResourceException

The `javax.resource.ResourceException` class is at the root of the System exception hierarchy, specified by JCA. The `ResourceException` class extends the `java.lang.Exception` class and is a checked exception. In CCI, the `ResourceException` class is the root of the hierarchy of the application exceptions. The `ResourceException` class provides the following information:

- Shows the error code for a particular resource adapter. This error code specifies the error condition.
- Represents an error by a String, which is resource adapter-specific. This String represents an exception message and can be retrieved by the `getMessage()` method of the ResourceException object.

A `ResourceException` exception can point to another exception and occurs as a result of some low level problem, such as hardware failure.

SecurityException

The javax.resource.spi.SecurityException class is a subclass of the ResourceExeption class. An error condition represented by this subclass corresponds to the Security contract, consisting of an application server and a resource adapter as its members. The SecurityException exception may occur due to the following reasons:

- Unsuccessful authentication of a resource or unsuccessful creation of a connection to EIS
- Incorrect security information, which is stored in a Subject instance that is passed to the Security contract
- Unsuccessful authentication of an EIS principal resource
- Unsuccessful establishment of a secure association between an enterprise application component and an EIS instance
- Unsupported security mechanism used by an EIS system or a resource adapter

LocalTransactionException

The `javax.resource.spi.LocalTransactionException` class is a subclass of the `ResourceException` class. The error conditions represented by this class correspond to the contract that is defined for local transaction management. The `LocalTransactionException` exception occurs due to the following reasons:

- Rollback of a transaction without the invocation of the commit() method in the `LocalTransaction` interface
- Incorrect transaction context during the execution of a transaction operation

- Initiation of a transaction with a thread on a `ManagedConnection` instance, which is related to the active local transaction

ResourceAdapterInternalException

The `javax.resource.spi.ResourceAdapterInternalException` class is a subclass of `ResourceException`, which represents all system-level error conditions that are associated with a resource adapter. The ResourceAdapterInternalException exception occurs due to the following reasons:

- Unsuccessful creation of EIS connection, probably because of an error in the communication protocol or resource adapter implementation
- Incorrect configuration of the ManagedConnectionFactory instance, which is used to create a new connection
- Incorrect implementation of a resource adapter

ApplicationServerInternalException

The `javax.resource.spi.ApplicationServerInternalException` class is a subclass of `ResourceException`. The exceptions depicted by the `ApplicationServerInternalException` class point particularly to application server errors. These errors generally occur due to errors made in the configuration of an application server.

ResourceAllocationException

The `javax.resource.spi.ResourceAllocationException` class is a subclass of `ResourceException`. Exceptions defined in this subclass are thrown when an application server or resource adapter fails to allocate system resources, such as threads and connections. Application server-specific `ResourceAllocationException` exceptions occur when the number of connections created is more than can be managed by a connection pool.

NOTE

The javax.resource.spi.CommException exception is a subclass of the ResourceException exception and occurs due to unsuccessful or interrupted communication between an enterprise application component and an EIS instance. Common CommException exceptions are raised due to communication protocol errors and invalid connections that occur because of server failure.

After learning about JCA exceptions, let's discuss how to package and deploy a resource adapter.

Packaging and Deploying a Resource Adapter

JCA specifies the interfaces that are generally used to package and deploy a resource adapter on an application server. Using these interfaces, you can plug in various resource adapters to an application server. Integration between a resource adapter and a Java EE application server is modular and portable. This modular integration is called a resource adapter module (the `.rar` file), which is similar to a Java EE application module (the `.ear` file) and includes Web and EJB components.

Now let's learn about the directory structure of a resource adapter.

Understanding Directory Structure of a Resource Adapter

A resource adapter provider creates and provides adapter interfaces and implementation classes, which are put in a `.rar` file or deployment directory. A resource adapter can be deployed either with the help of the `.rar` file or deployment directory. The `.rar` file or deployment directory contains the compiled classes that are required for the deployment of a resource adapter. These compiled classes are stored within subdirectories and the structure of these subdirectories is exactly the same as their Java package structures.

Resource adapters follow a common directory format. When the required classes and files are packaged in a `.rar` file, the same format is followed in the .rar file. Now, let's know the directory structure of the resource adapter.

Figure 17.12 displays the directory structure of a resource adapter:

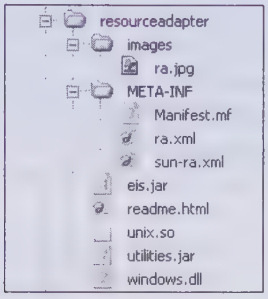


Figure 17.12: Showing the Directory Structure of a Resource Adapter

As you can see in Figure 17.12, the directory structure of a resource adapter consists of the following files:

- **ra.xml** – Contains standard deployment information
- **sun-ra.xml** – Contains additional deployment description
- **eis.jar** – Contains the Java classes and interfaces required by a resource adapter
- **readme.html** – Represents the documentation and related files that are required by a resource adapter
- **unix.so** – Represents the native library that is required by a resource adapter to interact with EIS on UNiplexed Information Computing System (UNICS), later known as the UNIX operating system
- **utilities.jar** – Contains the Java classes and interfaces required by a resource adapter
- **windows.dll** – Represents the native library that is required by a resource adapter to interact with EIS on Windows operating system

Let's now discuss the points to consider while packaging a resource adapter.

Packaging Considerations

As discussed earlier, you can deploy a resource adapter by using either a `.rar` file or deployment directory. Both the `.rar` file and the deployment directory include the compiled resource adapter interfaces and implementation classes developed by the resource adapter provider. Therefore, packaging a resource adapter is an important step in the successful deployment of the resource adapter. A `.rar` file may contain the following files:

- Deployment Descriptors (`ra.xml` and `sun-ra.xml`), which are contained in a subdirectory called `META-INF`.
- The `MANIFEST.MF` file that contains metadata information about an archive. A `JAR` tool implicitly creates a `MANIFEST.MF` file, which is an optional file.
- A resource adapter that contains multiple JAR files, such as `eis.jar` and `utilities.jar`. These JAR files provide the Java classes and interfaces used by a resource adapter.
- The documentation and related files, such as `readme.html` and `/images/ra.jpg`, which are used by the resource adapter.
- An `.ear` file containing resources required to deploy a resource adapter. In other words, if a resource adapter is packaged within a Java EE application, then you first need to create an .ear file that contains the resources required in the application. In addition, instead of deploying a `.rar` file, you can deploy the `.ear` file of the application server, which contains the resource adapter. The `MANIFEST.MF` file is automatically created while packaging the resource adapter on the application server. This file contains the `CLASSPATH` entry supported by the server.

Let's now learn to package a resource adapter.

Packaging a Resource Adapter

After arranging one or more resource adapters in a directory, you can package them in `.rar` files to successfully deploy the resource adapters. The following list indicates the typical sequence followed in packaging a resource adapter:

1. Develop a staging directory, which serves as a buffer to store updates
2. Copy the compiled resource adapter Java classes into the staging directory
3. Create JAR files by using JAR tools, which contain resource adapter Java classes
4. Add the JAR files at the top level of the staging directory
5. Create a META-INF subdirectory in the staging directory
6. Create Deployment Descriptor for the resource adapter, which is ra.xml file, and place it in the META-INF subdirectory
7. Add all the resource adapter configuration entries in the ra.xml file
8. Create another Deployment Descriptor of the resource adapter in the META-INF subdirectory, which is a `sun-ra.xml` file.
9. Add various details used at the time of deployment of the resource adapter, such as configuration details related to connection pooling and mapping a security role in the `sun-ra.xml` file.
10. Use the following command to create a `.rar` file:

```
jar cvf jar-file.rar -C staging-dir
```

By executing the preceding command, a `.rar` file is created that can be deployed on the Glassfish application server. The -C staging-dir option used in the preceding command informs the JAR tool to change the staging directory to the directory specified in the command. In our case, the staging-dir directory is used as the staging directory. By doing this, you can store directory paths in the `.rar` file, which are relative to the directory where you have staged the resource adapters.

Now, let's learn how to deploy a resource adapter.

Deploying a Resource Adapter

The process of deploying a resource adapter is the same as deploying Web and enterprise applications. A resource adapter can be deployed in an exploded directory format or as an archive (the `.rar` file).

Let's learn about Deployment Descriptor of a resource adapter, which contains information related to the deployment of a resource adapter.

Explaining the Deployment Descriptor for a Resource Adapter

A resource adapter requires two Deployment Descriptors to define its operational parameters. The JCA specification, Version 1.0 final release contains the definition of the `ra.xml` Deployment Descriptor, which is provided by Sun Microsystems. The `sun-ra.xml` Deployment Descriptor is specific to the Glassfish application server and specifies the operational parameters unique to the server. The Java EE Deployment API specification describes the general deployment procedure in detail.

Deployment Descriptor is defined by a resource adapter provider, which contains certain configuration details for the adapter. In other words, Deployment Descriptor for a resource adapter defines the following general information:

- Name of the resource adapter
- Description of the resource adapter
- Uniform Resource Identifier (URI) of a user interface for the resource adapter
- Name of the vendor providing the resource adapter
- Description of the licensing requirement for using the resource adapter
- Type of EIS system supported by the resource adapter, for example, the name of a specific database, ERP system, or mainframe transaction processing system without any versioning information
- String representing the version of the JCA specification supported by the resource adapter

Apart from providing Deployment Descriptor, you also need to provide the configuration properties file for the resource adapter. This file allows a resource adapter provider to give the `ResourceAdapter` instance configuration properties, which can be used by the resource adapter deployer to configure a `ResourceAdapter` JavaBean instance.

Let's now learn about the deployer of a resource adapter, which configures the resource adapter during the deployment of the adapter.

Explaining the Role of Parties Involved in Deployment of a Resource Adapter

Various parties, such as deployer, application server vendor, and application component provider play an important role in the deployment of a resource adapter. The following is the description of the tasks performed by these parties:

- **Deployer**—Configures a resource adapter in the target Java EE environment in which the adapter is to be deployed. The deployer provides the attributes in Deployment Descriptor needed for the configuration of the resource adapter. The deployer performs the following tasks in Deployment Descriptor:
  - Configures a property set for each ManagedConnectionFactory instance, which creates connections to multiple underlying EIS instances
  - Configures an application server for transaction management based on the level of transaction support specified by a resource adapter
  - Configures security in the operational environment, based on the security requirements specified by a resource adapter in its Deployment Descriptor.
- **Application server vendor**—Provides the JCA implementation in the Java EE application server provided by the vendor. Support for JCA in the application server helps to package, deploy, and run JCA component-based applications. The role of the application server vendor is also to provide a container and insulate the Java EE application components from the underlying system-level services.
- **Application component provider**—Develops enterprise application components that interact with EIS so that the EIS can use the functionalities of an enterprise application. The provider develops the components by using CCI, but does not provide implementations for the transaction, security, concurrency, and distribution services. Instead, these services are used by the application component from an application server.

With this, we come to the end of the chapter. Let's now recap the main points discussed in the chapter.

Summary

The chapter has described JCA and its key components. It has explained the life cycle management as well as the Workflow management of a resource adapter. In addition, the chapter has differentiated the JDBC and JCA technologies. Apart from this, you have learned about the Inbound Communication model, EJB invocation, the CCI API, and JCA exceptions. Towards the end, you have learned to package and deploy resource adapters.

The next chapter deals with Java EE design patterns.

Quick Revise

Q1. The package containing the `ActivationSpec` interface is ..... .

A. javax.resource.spi **B. javax.resource.cci**

C. javax.resource **D. javax.resource.spi.endpoint**

Ans. The correct option is A.

Q2. WorkListener interface belongs to the ..... package.

A. javax.resource.spi.endpoint **B. javax.resource.cci**

C. javax.resource.spi.security **D. javax.resource.spi.work**

Ans. The correct option is D.

Q3. What is common between interaction and connection?

A. Both are from the same package, `javax.resource.cci`.

B. Both are classes.

C. Both are methods.

D. Both are packages.

Ans. The correct option is A.

Q4. The `getConnection()` method declared in the `ConnectionFactory` interface throws an exception of type .....

A. UnavailableException **B. ResourceException**

C. NotSupportedException **D. None**

Ans. The correct option is B.

Q5. What is the need of JCA?

Ans. JCA is a Java technology that is used to connect an application server and EIS. Prior to the introduction of JCA, EAI was the only solution to connect an application server with EIS; however, the connection through EAI was not portable among various application servers. JCA served as a solution to allow an application server to connect with heterogeneous EIS systems. .

Q6. Define EIS.

Ans. EIS is an information system used by an entity such as an organization to manage large amounts of data to perform its day to day activities. EIS software consists of ERP, mainframe transaction processing, and non-relational databases.

Q7. Differentiate between JDBC and JCA.

Ans. JDBC provides an interface that enables applications to interact with only relational databases while JCA provides the architecture to integrate enterprise applications and EIS. JDBC uses a specific driver to establish a connection to a database, but JCA uses an EIS-resource adapter to interact with an EIS system.

Q8. What is a resource adapter?

Ans. A resource adapter is a system-level software driver, which allows an application client or server to connect to EIS. It provides a contract for the services offered by the application server, such as underlying mechanisms, transactions, security, and connection pooling.

Q9. Define work exceptions.

Ans. When an exception related to the Work Management contract is reported, an application server throws work exceptions. The following are some examples of work exceptions:

- WorkException
- WorkRejectedException
- WorkCompletedException

Q10. What is CCI?

Ans. CCI is a standard API that helps enterprise application components to access EIS. It provides common classes and interfaces to interact with various EIS. This interaction can be derived by application components and EAI frameworks.

Q11. How is a resource adapter packaged?

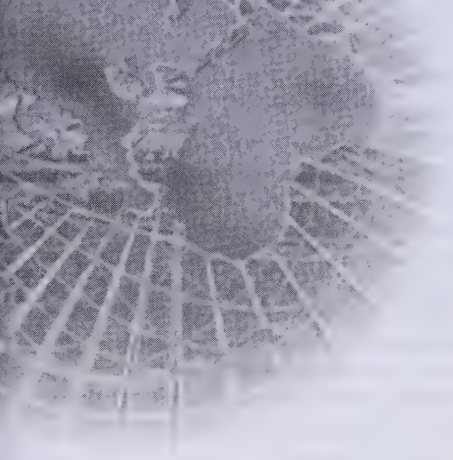
Ans. A resource adapter is packaged into a `.rar` file by using the `JAR` tool. The `.rar` file must have a configuration file, i.e. `ra.xml`, which stores deployment related information for the resource adapter.

Q12. Define the WorkEvent class.

Ans. The `WorkEvent` class is an abstract class that provides information about an event type, a source object, a listener associated with the `Work` instance, and any exception thrown during the processing of submitted Work instances.

Q13. What are System contracts?

Ans. JCA defines a number of system-level contracts to be implemented by a resource adapter and application server. These system-level contracts are classified into three categories: the Connection Management contract, the Transaction Management contract, and the Security contract. These contracts specify the simple interfaces between the application server and EIS.



18

Java EE Design Patterns

With the advent of Web technologies, almost all the industries use the Internet in many ways. Web applications are developed to reach customers, expand business domains, and provide better services to the clients. Java EE provides a platform for the development of distributed enterprise or Web applications. As the size of the enterprise increases, there arises a need for more effective Web solutions; thereby, resulting in an increase in the size and complexity of Web applications. Therefore, the Web applications need to be scalable, easily expandable, and reusable to ensure effective management of increasing complexity and size. Development of such Web applications requires some standard architecture. A lot of work has been done in this domain of developing Web applications. All the Web applications follow the 3-tier architecture, which provides a clear separation between business logic and presentation logic of the Web application. This ensures a standard format to be followed in the development and maintenance of the basic infrastructure of Java EE Enterprise Applications. Along with a standard architecture, various standard design patterns are also applied to enterprise applications to ensure consistency and delivery of high quality business solutions.

In this chapter, you learn about the Java EE application architecture. Next, you learn about the design patterns. Further, you learn about the role of design patterns in Web applications. Finally, the chapter discusses the different types of design patterns.

Let's start the chapter by learning about the Java EE application architecture.

Describing the Java EE Application Architecture

A proper technical architecture is necessarily required to ensure development and management of an enterprise application, which is scalable, robust, flexible, and reusable. Java EE allows various architectures to be used for managing enterprise applications. However, the 3-tier architecture is the most commonly used architecture. Java EE employs a component-based approach for developing, managing, and deploying enterprise applications. The 3-tier architecture consists of three tiers or layers, namely presentation tier or client tier, business tier or application tier, and data tier or enterprise information service tier. The client tier or presentation tier manages the user interface of the application. This tier is responsible for handling user inputs and interactions as well as presentation of data to the user. The business tier or application tier implements the business logic for the application. This tier is responsible for processing the input data and generating response accordingly. The data tier or enterprise information service tier handles the data for the application. This tier manages a database and stores persistent data for the application. Figure 18.1 shows a representation of the 3-tier architecture:

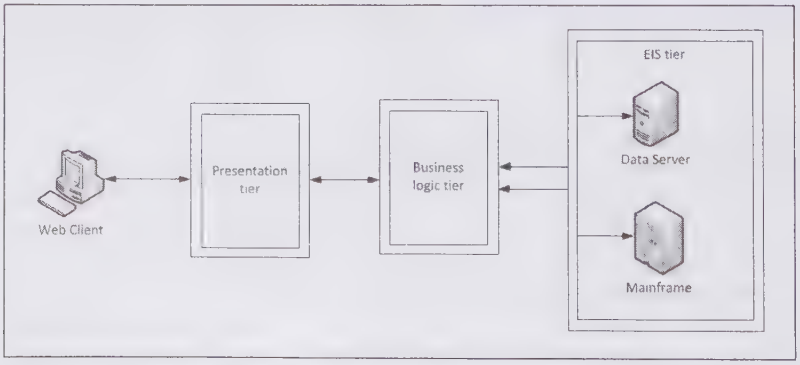


Figure 18.1: Displaying the 3-tier Architecture

Figure 18.1 shows a representation of the 3-tier architecture, where a client interacts with the presentation tier, which provides the user interface; the requests made by the client are processed at the business tier, which provides the business logic; and the client data is stored at the enterprise information service tier. For example, if a client wants to store some data on the data server, then instead of forwarding the request directly to the data server, the request is first forwarded to the presentation tier; the business tier then processes the request and validates the user before forwarding the data to the data server. In this way, the 3-tier architecture is used to design the enterprise applications..

However, designing Web applications is not an easy task. A lot of considerations need to be examined before reaching to a conclusion about how to develop a Web application or an enterprise application. To facilitate the development process for developing Web applications, a lot of design patterns have been proposed. Before discussing about the various available design patterns that are applied in Web application development and designing, let's first examine what is meant by a design pattern in context to a Web or an enterprise application.

Introducing a Design Pattern

Design patterns refer to language-independent solutions provided for solving commonly recurring design problems. A design pattern first describes the problem and then the solution that can be applied to the problem. The design patterns are implemented in the form of objects and classes that provide solutions to the problem. The concept of design patterns was first applied to object oriented programming. Later, this concept was also applied to the development of enterprise applications.

There are various problems concerning the designing issues of Web applications that may arise while developing the application. Some of the problems occur more often and creating a solution from scratch all over again every time the problem occurs, results in wastage of time and effort involved in the process of development of the Web or enterprise application. Therefore, design patterns have been developed which provide well examined and proven solution to recurring problems while designing applications. The design patterns can be divided into various categories on the basis of their application domains.

After understanding about a design pattern, let's learn in what ways it can be helpful in designing enterprise applications.

Discussing the Role of Design Patterns

Design patterns are used while designing the enterprise applications to provide standardized, proven, and tested solutions to recurring problems that arise during the development of enterprise applications. Design patterns offer various benefits to enterprise application development. The application of design patterns greatly speeds up the development process. Moreover, reusing design patterns help in recognizing and avoiding subtle issues that can later become a bottleneck to the development process. Design patterns provide code reusability and improve readability of code for developers who are familiar with the design patterns. They also help in improving communication between the developers by providing a vocabulary which uses well-known and well-understood names and terms for software interaction. The important benefits of design patterns can be summed up as follows:

- Provides tested solutions to frequently and repeatedly occurring problems
- Provides code reusability
- Helps in defining application architecture more efficiently and clearly
- Provides more reliability and transparency to the design of the application

After discussing the roles that design patterns play in an enterprise application, let's move forward and explore the types of design patterns that can be used in the development of such applications.

Exploring Types of Patterns

Design patterns became widely acceptable in the field of computer science after the publication of the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides in 1994. Java EE design patterns help in recognizing potential problems that have been observed while using various Java EE technologies and finding appropriate implementable solutions to those problems according to the application architecture. The various available patterns can be categorized on the basis of the architectural tier in which they are applied. Figure 18.2 shows some patterns as provided by Sun Java Center:

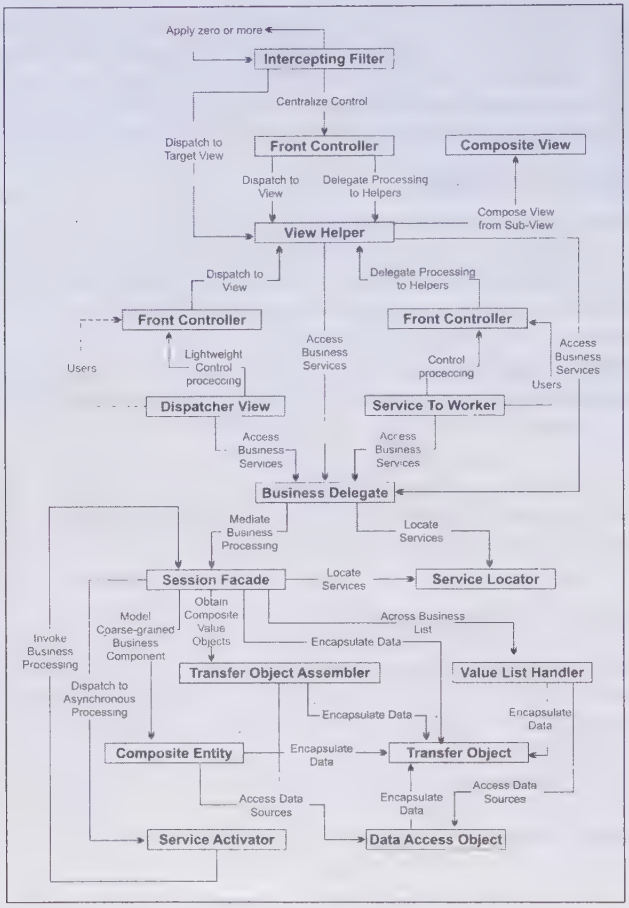


Figure 18.2: Displaying the Java EE Design Patterns from Sun Java Center

Table 18.1 lists the design patterns on the basis of the architectural tiers to which they belong:

Table 18.1: Describing the List of Design Patterns

Tier	Pattern Name	Description
Presentation Tier	Composite View	Manages the layout of a Web page and presentation of data on it. It represents a composite view, in which data is presented in the form of sections which are built by using reusable sub-views.
	Dispatcher View	Provides services for data processing, such as authentication; and request handling by offering limited amount of business processing. It solves a combination of problems handled by the Front Controller and View Helper patterns.
	Front Controller	Implements a central controller in an enterprise application to process the requests.
	Intercepting Filter	Intercepts the request reception and response transmission. It also helps in pre-processing or redirection of a request; and post-processing or content replacement of a response.
	Service to Worker	Maintains a separation between the models, views, and controllers. The workers and views are managed by a dispatcher object.
	View Helper	Helps in maintaining a separation between the presentation logic and

Table 18.1: Describing the List of Design Patterns

Tier	Pattern Name	Description
		business logic. The business logic is implemented for a view using the view helper class.
Business Tier	Business Delegate	Manages the complexity involved in lookup and exception handling for distributed components by introducing an intermediate business delegate class which reduces coupling between presentation tier and business tier.
	Composite Entity	Manages a network of interrelated persistent objects.
	Fast Lane Reader	Helps in accessing the tabular read-only data more efficiently. Persistent data can be accessed directly by the Fast Lane Reader component using Java Database Connectivity (JDBC).
	Service Locator	Allows the users to uniformly access business components and services. To accomplish this, the Service Locator pattern uses Java Naming and Directory Interface (JNDI).
	Session Facade	Provides a simple interface to a client by hiding all the complexities involved in the interaction between various business components by encapsulating the complex implementation details in a session bean that acts as a facade.
	Transfer Object	Provides a way to transfer bulk data from a remote server by using a serialized Java object.
	Value List Handler	Provides a handler object that enables the users to iterate over a read only list.
Integration Tier	Data Access Object	Allows the code for accessing and manipulating data to be encapsulated in a separate layer.
	Service Activator	Allows asynchronous invocation of the business services.

Table 18.1 lists the design patterns offered by Sun Java Center. The design patterns are categorized according to their implementation in Presentation, Business, and Integration tiers. The succeeding sections discuss some of the design patterns listed in Table 18.1 in detail. For each pattern, the discussion covers the definition of the problem, forces, possible solution, strategies, and consequences of applying the pattern.

The Problem section of each pattern discusses the problem that led to the creation of that pattern. The Forces section discusses the tasks need to be done in the context of the problem. The Solution section discusses how the concerned pattern has helped to solve the discussed problem. The Strategies section discusses the various strategies that can be used to implement the pattern and the result of the pattern is discussed in the Consequences section.

Let's start exploring some of the design patterns listed in Table 18.1, starting with the Front Controller pattern, in the next section.

The Front Controller Pattern

The Front Controller pattern provides a centralized control to the application request processing. It introduces a central controller component that acts as a common entry point for all application requests. The controller component provides the logic for handling user requests, invoking security services, managing content retrieval, delegating business processing, and selecting appropriate views. The controller component may delegate some processing to a helper component or dispatcher component. The helper component takes the responsibility of helping a view or controller in completing its processing. For example, the helper component may help in gathering the data required by the view. The data can be provided as raw data, or the helper component may format that data as required by the view. The dispatcher component helps the controller in managing and selecting the next view to be presented to a user.

Problem

The Presentation tier requires a centralized control to handle multiple user requests for effectively managing the activities of content retrieval, view management, and security service invocation. However, users can access the views directly without following a centralized mechanism. This approach requires the code for processing logic and system services to be duplicated for each view. When the responsibility of view navigation and management is left to the views themselves, then this may lead to commingled view content and navigation. For example, in an online shopping Web site, multiple user requests need to be handled at the same time and the sequence of steps involved in shopping process also needs to be controlled. There may be multiple users logged in the Web site at a time. This scenario requires that consistency and security should be maintained across multiple requests. A lot of code duplication can occur if the code that controls the input is scattered across multiple objects.

Forces

The problem described in the preceding section results in the following forces:

- Replication of code in multiple views
- Similar business requests responded by multiple views
- Complex logic for view management
- Completion of common system services on a per request basis

Solution

The solution is obtained by introducing a `Front Controller` component that acts as a common entry point for handling multiple client requests. The `Front Controller` component provides a centralized control mechanism for request handling that reduces code duplication and encourages code reusability. The Front Controller pattern helps to achieve the following tasks:

- Calling up the security services, such as authentication and access control
- Managing request handling
- Managing view selection and navigation
- Managing client's state
- Handling errors

Figure 18.3 shows the representation of the Front Controller pattern:

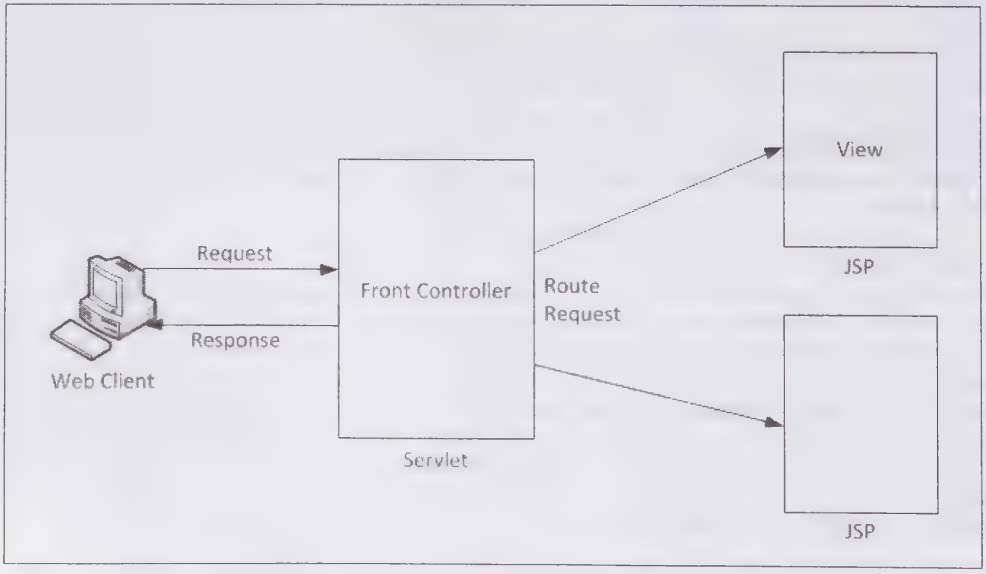


Figure 18.3: Displaying the Role of Front Controller

As shown in Figure 18.3, the central Front Controller component implemented as a servlet receives a request from a client and redirects it to respective views. It also forwards back the response to the client. Multiple controllers can also be used in an application, where each controller is mapped to a distinct set of services. Figure 18.4 shows the class diagram of `Front Controller`:

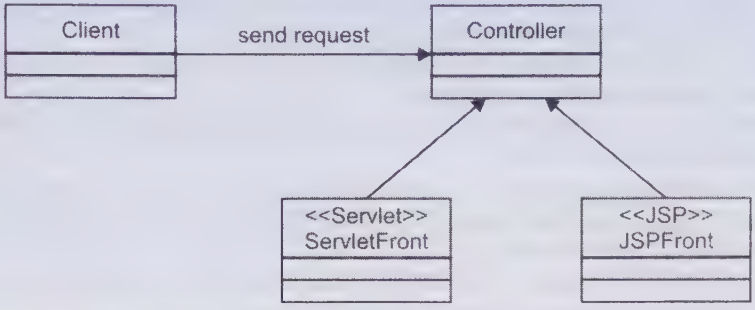


Figure 18.4: Displaying the Front Controller Class Diagram

The `Front Controller` component may use a separate dispatcher component for implementing the required workflow. It can also use helper components, which are implemented as JavaBean components and custom tags. Figure 18.5 shows a sequence diagram representing the interaction between the components involved in the Front Controller pattern:

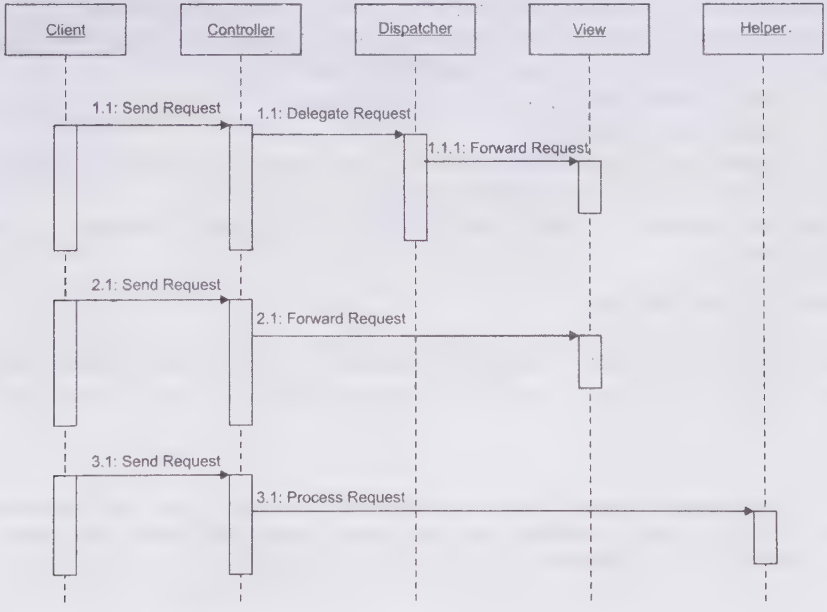


Figure 18.5: Displaying the Sequence Diagram for the Front Controller Pattern

The sequence diagram shown in Figure 18.5 depicts how a request is handled by the controller component.

Strategies

The following strategies can be used during implementation of the Front Controller pattern:

- **Servlet Front Strategy**—Implements the controller component as a servlet, which manages request handling aspects related to business processing and control flow. This is preferred over Java Server Pages (JSP) Page Front strategy.
- **JSP Page Front Strategy**—Implements the controller component as a JSP page. However, this strategy is not preferred because modifying the implementation code for business processing logic involves modifications to be made in a page that uses markup language.
- **Command and Controller Strategy**—Specifies that a generic interface should be provided to the helper components to reduce coupling between them.
- **Physical Resource Mapping Strategy**—Involves requesting resources by their specific physical resource name instead of logical names.
- **Logical Resource Mapping Strategy**—Involves requesting resources by their logical names.
- **Multiplexed Resource Mapping Strategy**-- Involves requesting a single physical resource by not just one but a complete set of logical names.
- **Dispatcher in Controller Strategy**—Involves implementation of the dispatcher functionality within the controller, where the dispatcher only has a minimal functionality.
- **Base Front Strategy**—Involves implementation of a controller base class that contains common code and provides default implementation for controllers. More specific controllers can extend the controller base class to acquire default functionality.
- **Filter Controller Strategy**—Implements some part of controller functionality using a filter.

Consequences

Following consequences are observed when the `Front Controller` pattern is used:

- **Centralizes control**—Implements a central controller component for handling requests and implementing business logic. Therefore, requests can be easily tracked and logged.
- **Improves manageability**—Provides a single entry point for all requests, which requires fewer resources; thereby, improving manageability.
- **Improves reusability**—Moves common code to a single controller component. Therefore, the code is not duplicated within multiple views; thereby, encouraging code reusability.
- **Improves role separation**—Divides responsibilities among various application components for effective role separation.

The Composite View Pattern

The Composite View pattern allows the management of layout and presentation of the data on a Web page in a more effective manner by providing a composite view which consists of multiple sub-views. The Web page is divided into sections each of which is handled by a separate view. This results in a Web page that consists of a set of views.

Problem

Web pages normally include content from various sources that is presented using sub-views. The content is incorporated in the Web page by embedding the code directly within each atomic view. However, the content of the views changes frequently. Embedding the code directly into individual views makes code modification and layout management difficult and error prone. Moreover, to reflect the modifications in each sub-view; the server may need to be restarted.

Forces

The problem discussed in the preceding section results in the following forces:

- The atomic content of the view more often changes
- Multiple composite views utilize similar sub-views which are presented either by using surrounding template text, or may be displayed at different location within the Web page
- It is difficult and error-prone to maintain and manage changes in the layout and code

- The frequent embedding of code, which controls the frequently changing portions, directly into views greatly affects the availability and maintenance of the system

Solution

The solution to the problem mentioned with respect to Composite View pattern is to implement a composite view which is made up of multiple atomic sub-views. This involves dynamic inclusion of each component of the Web page template into the page. This allows you to manage the layout independent of the content of the page. The composite view pattern provides the following benefits:

- Encourages code reusability and modular design
- Allows you to create pages using components that can be incorporated dynamically by combining them in a variety of ways
- Allows layout of the page to be modified independent of the page content
- Helps in creating a prototype for examining the layout of a site

The responsibilities of various components which participate in the Composite View pattern are listed as follows:

- **Composite view** – Provides a view which is an aggregate of multiple independent atomic sub-views.
- **View Manager** – Manages page structure and layout by managing and controlling the inclusion of code fragments in the composite view. It can be implemented as a JavaBean helper or custom tag helper.
- **Included view** – Refers to a sub-view which is an atomic portion of a larger view. The sub-view can either be composite or can itself include multiple views.

Figure 18.6 shows the class diagram representing the `Composite View` pattern:

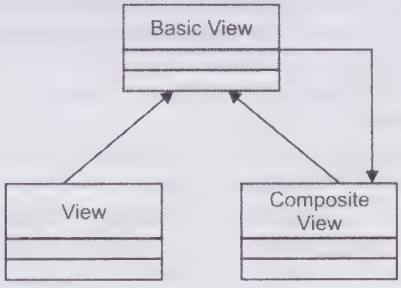


Figure 18.6: Displaying the Class Diagram Representing the Composite View Pattern

Figure 18.7 shows the sequence diagram for the Composite View pattern:

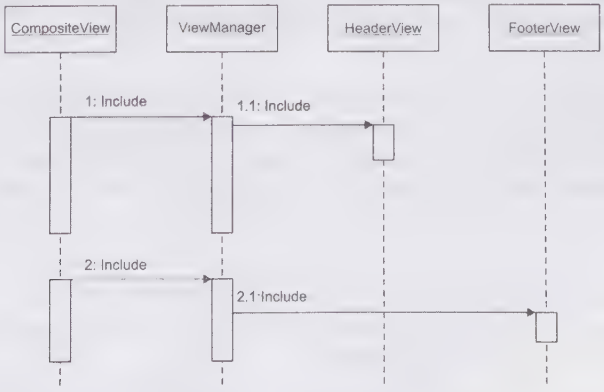


Figure 18.7: Displaying the Sequence Diagram for the Composite View Pattern

Figure 18.7 describes the sequence of interaction between the components of the Composite View pattern. The CompositeView component partitions the user interface into multiple views, and then, the ViewManager component controls the layout and structure of the Web page.

Strategies

The strategies to implement the Composite View pattern are as follows:

- **JavaBeans View Management Strategy** – Implements a JavaBean component for handling view management.
- **Standard Tag View Management Strategy** – Uses standard JSP page tags for managing views.
- **Custom Tag View Management Strategy** – Uses custom tags for implementing view management.
- **Transformer View Management Strategy** – Uses an Extensible Stylesheet Language (XSL) Transformer for handling view management.
- **Early-Binding Resource Strategy** – Refers to translation time content inclusion. As the name suggests, content is included in the page at translation time. This strategy is appropriate to be used for maintaining and updating static templates.
- **Late-Binding Resource Strategy** – Refers to runtime-content inclusion. As the name suggests, content is included in the page at runtime. This strategy is appropriate to be used for maintenance of composite pages.

Consequences

The following consequences are observed when Composite View pattern is used:

- **Promotes modular design** – Encourages modular design. The code of atomic portions of a template can be used in multiple views and at different locations of a page; thereby, improving maintainability.
- **Enhances flexibility** – Includes views conditionally at runtime on the basis of user role or security policy.
- **Enhances Maintainability** – Allows you to make changes to the content of portions of a template, without affecting its layout as the template code is not directly hard coded into a view. The changes made can be reflected immediately, on the basis of implementation strategy. The page layout can also be changed easily since modifications are centralized.
- **Reduces Manageability** – Implements sub-views as page fragments. When these atomic fragments are aggregated together, manageability issues may arise.
- **Performance Impact** – Creates a single layout by combining multiple sub-views. When views are included at runtime, some delay is observed. However, when views are included at translation time, some problems are encountered while changes are made to sub-views.

The Composite Entity Pattern

The Composite Entity pattern helps in effectively modeling a set of dependent interrelated objects when they are mapped to an Enterprise JavaBean object model. The interface of composite entity is itself coarse-grained, which internally manages communication among fine-grained dependent objects.

Problem

One of the repeatedly observed design problem is encountered when an object model is directly mapped to an Enterprise JavaBean (EJB) object model. It should be considered carefully whether to implement an object as an entity EJB or a plain Java object. It is preferable to model coarse-grained business entities using remote entity beans. However, problems are encountered when remote entity beans are used for modeling fine-grained entities.

Forces

The problem discussed in the preceding section results in the following forces:

- Implementing entity beans as coarse-grained objects. This helps in avoiding drawbacks observed in case of implementing entity beans as fine-grained objects.

- Mapping relational database schema to entity beans. This involves large number of fine-grained entity beans to be created. It is more preferable to have coarse-grained entity beans. At the same time, it is also recommended that the overall number of entity beans should be kept less.
- Mapping object model to EJB model. This results in creation of fine-grained entity beans which map to database schema. This mapping leads to problems concerning performance security issues, manageability, and transaction processing. Moreover, it is difficult and expensive to manage inter-entity bean relationship.
- Mapping of entity beans to database schema. This results in a tight coupling between entity beans and database schema where changes made to one are reflected in the other.

Solution

The solution to the problem stated with respect to the Composite Entity pattern is to implement the set of dependent persistent objects by using a composite entity instead of implementing the dependent objects as individual fine-grained entity bean. A persistent object stores its state in a data store and is shared by multiple clients. These persistent objects can either be coarse-grained objects or dependent objects. Dependent objects are usually managed by the coarse-grained parent objects and are not directly accessible by the client. A composite entity bean represents a coarse-grained object and all its dependent objects as well. It aggregates interrelated dependent persistent objects into single entity bean; thereby, reducing the overall number of entity beans in the application.

Figure 18.8 shows the class diagram representing Composite Entity pattern:

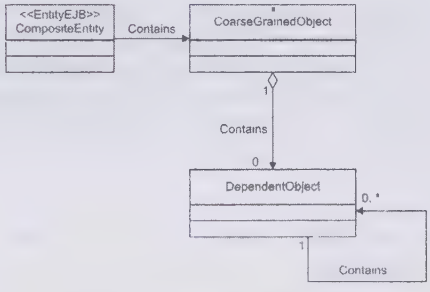


Figure 18.8: Showing the Class Diagram Representing the Composite Entity Pattern

The class diagram in Figure 18.8 shows that the CompositeEntity component contains the CoarseGrainedObject component, which in turn contains the DependentObject component. Figure 18.9 shows the sequence diagram for the Composite Entity pattern displaying the interaction among its various components:

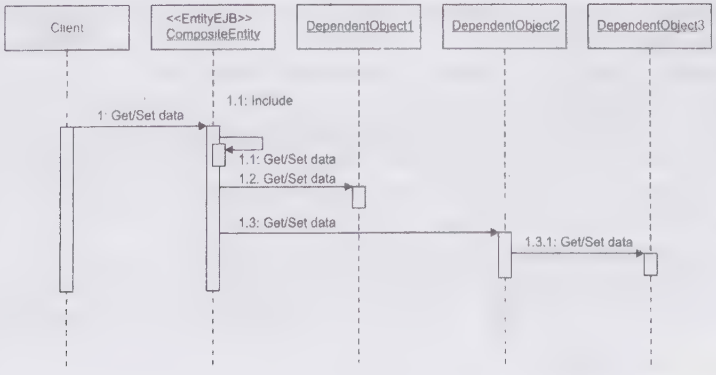


Figure 18.9: Showing the Sequence Diagram for Composite Entity Pattern

Strategies

The strategies used for implementing the Composite Entity pattern are as follows:

- **Composite Entity Contains Coarse-Grained Entity Strategy** – Specifies that the coarse-grained entity object is contained by the composite entity. The coarse-grained entity object maintains connection with the dependent objects as well.
- **Composite Entity Implements Coarse-Grained Entity Strategy** – Implies that composite entity is itself a coarse-grained entity object which implements the coarse-grained entity and thereby contains its attributes and methods. The dependent objects are represented as attributes of the composite entity which maintains and manages the relationship between the coarse-grained object and its dependent objects.
- **Lazy Loading Strategy** – Implies that all the dependent objects related to the composite entity are not loaded in an application while the composite entity bean is initialized by the EJB container. Only the most important required dependent objects are loaded during initialization. The remaining dependent objects are loaded on an on-demand basis when they are required or accessed by the client.
- **Store Optimization (Dirty Marker) Strategy** – Involves the creation, implementation, and usage of the `DirtyMarker` interface that contains methods, which can be used by the dependent objects to let the caller be aware of the changes in their state. This enables the caller of the objects to selectively save only the dependent objects that have changed since the last call to the `ejbStore()` operation while persisting the composite entity. Otherwise, the complete dependent object graph needs to be persisted to the database whenever the `ejbStore()` method is called by the EJB container.
- **Composite Transfer Object Strategy** – Allows the client to obtain the desired information about the coarse-grained object and its dependent objects from a composite entity with just one remote call using the Transfer Object component. This component collects all the required information into a single object using one remote call. When the client receives Transfer Object, then rest of the calls made by the client to Transfer Object are invoked locally to the client. The Transfer Object component can also gather data from a subset of dependent objects if required by the client.

Consequences

The consequences of using the Composite Entity pattern are listed as follows:

- **Elimination of Inter-Entity Relationship** – Groups all dependent objects into a single entity bean using a composite entity pattern; thereby, eliminating inter-entity relationships.
- **Improved Manageability** – Uses a composite entity for grouping fine-grained entities; thereby, resulting in fewer numbers of coarse-grained entities instead of more number of fine-grained entities. This, in turn, improves manageability.
- **Improved Network Performance** – Reduces inter-entity bean communication between fine-grained entity bean objects; thereby, resulting in an improved network performance.
- **Reduction in Database Schema Dependency** – Hides the database schema, which is mapped internally to the coarse-grained entity bean, from the client. Therefore, the client is not affected whenever changes are made to the database schema of the mapped coarse-grained entity bean.
- **Increased Object Granularity** – Reduces continuous communication between a client and business tier. The client interacts with a composite entity instead of numerous fine-grained entity beans. The data requested by the client is also sent collectively by returning Transfer Object using single remote call.
- **Facilitating Composite Transfer Object Creation** – Sends the requested data collectively using the composite Transfer Object which is invoked by a single remote call.
- **Overhead of Multi-level Dependent Object Graph** – Involves increased overhead in storing the dependent object graph for the composite entity if it has many levels. Using optimizing strategies involves the overhead of checking the objects whose states have been changed to save the objects selectively.

The Intercepting Filter Pattern

The Intercepting Filter pattern introduces a filter that intercepts and intermediates the request to be received and the responses to be transmitted. The filter allows pre-processing and post-processing of the request and

response, respectively. Therefore, the filter can either modify or redirect the request; and post-process or replace the content in the response.

Problem

All the requests received by the presentation tier cannot be directly forwarded to a request handler. Sometimes, the request needs to be modified or pre-processed before it is handed over to the request handler. Similarly, the responses generated in response to a request cannot always be forwarded back to the client directly. Sometimes, the content of the response need to be post-processed or replaced before it is sent to the client. The Intercepting Filter pattern solves this problem by introducing an intercepting filter between incoming requests and outgoing responses; thereby, allowing requests to be pre-processed and responses to be post-processed.

Forces

Following forces are involved in the problem stated in the preceding section:

- Implementation of common processing logic, such as, logging client information, and authentication by a centralized filter. This allows the processing to be carried out on a per request basis.
- Centralization of common logic.
- Addition and removal of services without affecting existing components.

Solution

The problem stated in the preceding section can be solved by introducing intercepting filters that implement common code to process common services for pre-processing incoming requests and post-processing outgoing responses. In this approach, whenever a change is required to be introduced in the processing logic for common services, the core request processing code need not be modified. The intercepting filters implementing the code for common services are independent of the main application and can be added or removed without affecting the application. The intercepting filters can also be chained together in sequence, one after the other.

Figure 18.10 shows the class diagram depicting the Intercepting Filter pattern:

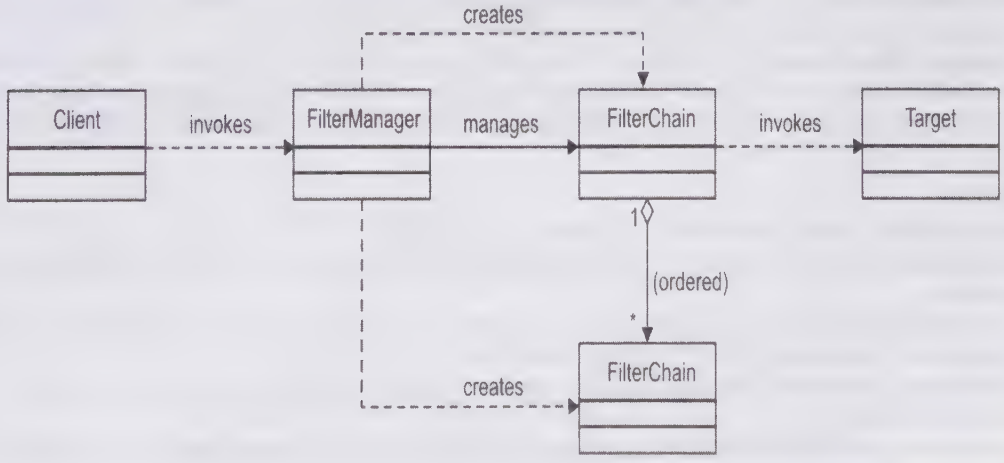


Figure 18.10: Showing the Class Diagram for the Intercepting Filter Pattern

Figure 18.11 shows the sequence diagram representing the interactions between various components of the Intercepting Filter pattern:

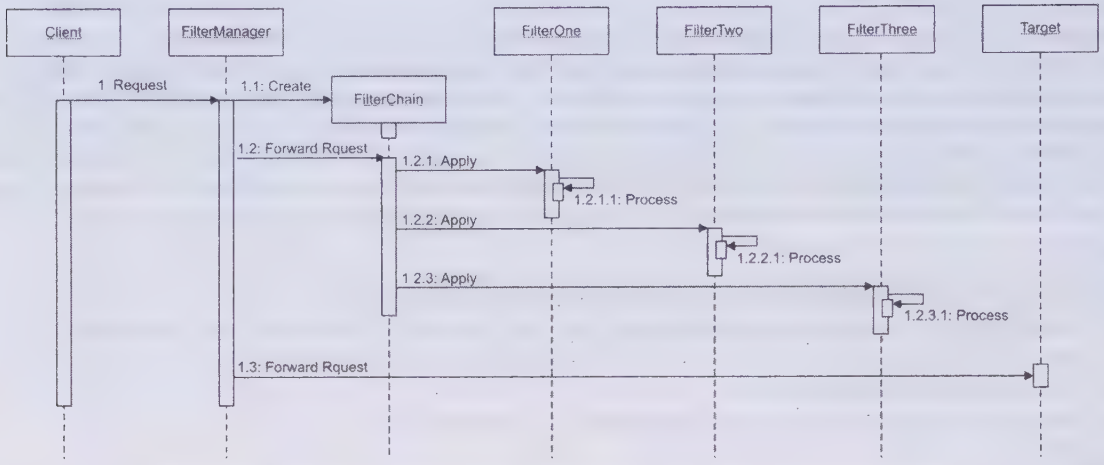


Figure 18.11: Showing the Sequence Diagram for the Intercepting Filter Pattern

Strategies

The Intercepting Filter pattern can be implemented by using the following strategies:

- **Custom Filter Strategy**—Implements the filter by using a user-defined custom strategy. However, it is less effective and less preferable because it does not request and response object in a standard manner.
- **Standard Filter Strategy**—Declares filters using the deployment descriptor. Creation of filter chains as well as addition and removal of filters to those chains follow a standard mechanism. Filters are added and removed declaratively by modifying the deployment descriptor.
- **Base Filter Strategy**—Implements a base filter that acts as a common superclass for all filters. Common features are implemented in the base filter and shared among all the other filters.
- **Template Filter Strategy**—Implements a base filter that is inherited by all other filters and provides template methods whose specific implementations are provided by the respective inheriting filters. The base filter only defines the methods indicating what is to be done. How to accomplish the process using the method is left to individual inheriting filter.

Consequences

The consequences of using Intercepting Filter pattern are as follows:

- **Centralized Control with Loosely Coupled Handlers**—Provides central control mechanism for common services.
- **Improved Reusability**—Implements filters as independent components which can be added or removed independently and reused across varying applications.
- **Declarative and Flexible Configuration**—Implements numerous services in a flexible manner without affecting the application.
- **Inefficient Information Sharing**—Implies that information sharing between filters can be inefficient and costly as they are loosely coupled.

The Transfer Object Pattern

The Transfer Object pattern, also termed as Value Object pattern, implements a Transfer Object component, which is a serializable class that aggregates related attributes together to form a composite value that can be returned as a return type by the methods. The instance of Transfer Object can be obtained by calling the coarse-grained business methods. The fine-grained values can be accessed locally within Transfer Object.

Problem

Sometimes the client requires to access bulk data from an enterprise bean. The requested data may include a set of attributes that are to be accessed together. In such a case, the business object's get method needs to be invoked multiple times for each attribute to fetch the values for all the attributes. This results in increased network traffic, high latency, and unnecessary consumption of server resources.

Forces

Following forces are involved in the problem stated in the preceding section:

- The client requests for the data by making multiple remote method calls through the remote interface to the bean.
- The data is read more frequently than being modified. The data is required by the client for presentation, display, and other read-only operations. The data is modified and updated less frequently.
- The client sometimes request for multiple attributes, which are accessed by making multiple remote method calls to the enterprise bean.
- The increased number of remote calls affects network traffic and performance.

Solution

The problem stated in the preceding section can be solved by using Transfer Object that aggregates related attributes together into a composite object. The client can access Transfer Objects by a single remote method call. Individual attributes can then be accessed locally from Transfer Object. This significantly helps in reducing network traffic as well as latency, and also provides better resource utilization.

Figure 18.12 shows the class diagram representing the Transfer Object pattern:

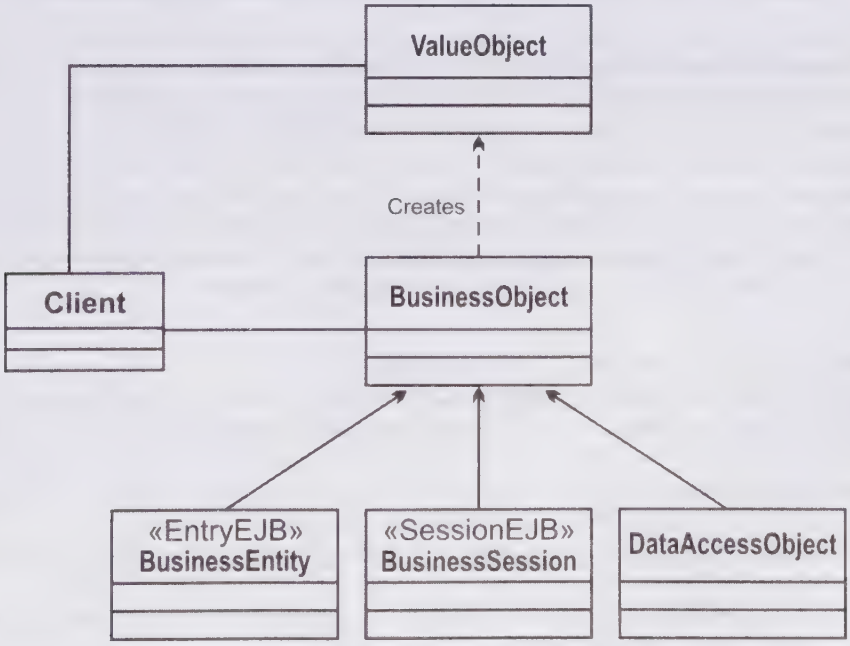


Figure 18.12: Showing the Class Diagram Representing the Transfer Object Pattern

As shown in Figure 18.12, Transfer Object is created on an on-demand basis by the enterprise bean. Figure 18.13 shows the sequence diagram depicting the interaction between various components of the Transfer Object pattern:

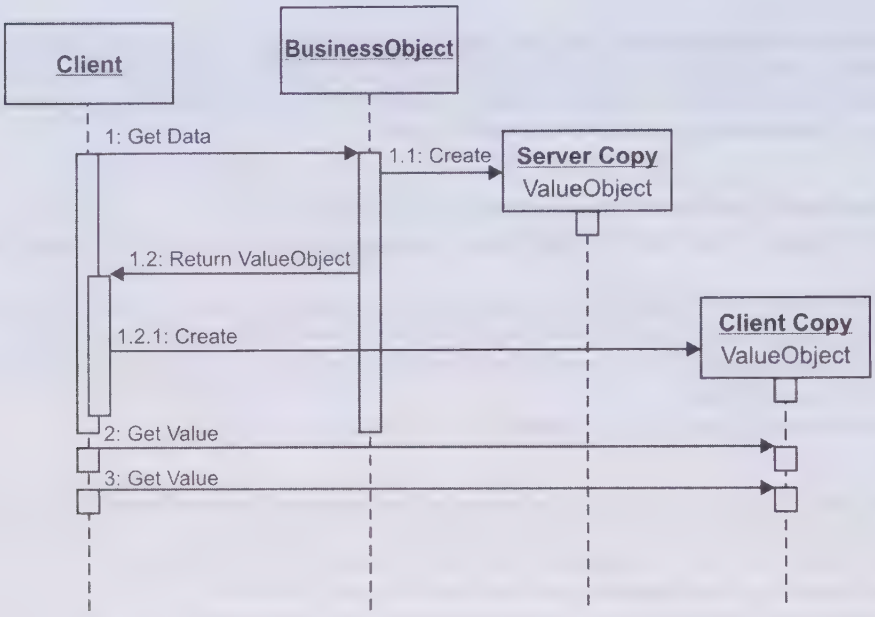


Figure 18.13: Showing the Sequence Diagram for the Transfer Object Pattern

Strategies

Following strategies can be applied to implement the Transfer Object pattern:

- **Updatable Transfer Object Strategy** – Implements Transfer Object that can make data accessible from the business object to the client, as well as from the client back to the business object.
- **Multiple Transfer Object Strategy** – Creates multiple Transfer Objects to manage a single business object.
- **Entity Inherits Transfer Object Strategy** – Specifies that the entity bean inherits the Transfer Object class. Therefore, the attributes specified in the Transfer Object class do not get duplicated in the entity bean.
- **Transfer Object Factory Strategy** – Allows on-demand creation of Transfer Objects by using the concept of reflection. Therefore, multiple Transfer Objects can be created dynamically.

Consequences

The consequences of using Transfer Object pattern are as follows:

- **Entity Bean and Remote Interface Simplification** – Uses entity bean's getData() method to fetch Transfer Object that contains the attribute values. Therefore, multiple get methods need not be implemented in the bean's remote interface.
- **More data Transfer in Fewer Remote Calls** – Involves only a single remote call to be made to Transfer Object. Individual attributes can then be accessed locally from Transfer Object.
- **Reduced Network Traffic** – Results in reduction of network traffic, as only one remote call to Transfer Object needs to be made.
- **Reduced Code Duplication** – Reduces code duplication when the Entity Inherit Transfer Object strategy or the Transfer Object Factory strategy is used.
- **Increased Complexity due to Synchronization and Version Control** – Increases complexity when multiple clients request for updates to the entity values.

The Session Facade Pattern

The Session Facade pattern uses a central high level component, which is implemented as a session bean and contains the functionality of the complex interactions that occur between various lower-level business components. The higher-level session bean thereby provides a single interface to the client for accessing the functionality of an application.

Problem

Applications sometimes involve complex business processes that comprise many business objects. In such cases, the participating business objects or business classes can be tightly coupled, resulting in higher dependence between the business objects, lower design clarity, and less flexibility towards modifications. Moreover, multiple method invocations need to be made in case of numerous business classes, which affects the network performance. In addition, the direct interaction between the client and the business objects results in tight coupling between them and makes the clients dependent on the business objects implementation.

Forces

Following forces are involved in the problem stated in the preceding section:

- Providing the client with a simpler interface; thereby, hiding all the underlying complexities in the interaction between the lower-level business objects.
- Reducing the number of participating business objects to provide better manageability, flexibility, and enhanced code clarity.
- Reducing the coupling between the business objects and the client.
- Centralizing the business logic for the application that needs to be exposed to the client.

Solution

The solution to the problem stated in the preceding section lies in aggregating and encapsulating the code for the complex interactions between the business objects into a centralized session bean. The session bean acts as a facade for the lower-level business processes. The session facade provides a simpler centralized higher-level interface for the application's functionality and manages the interaction between the lower-lever business processes. Figure 18.14 shows the class diagram representing the Session Facade pattern:

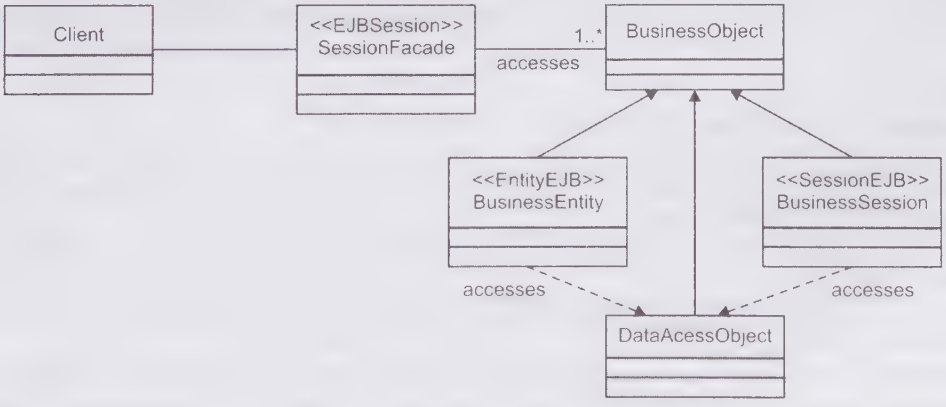


Figure 18.14: Showing the Class Diagram Representing the Session Facade Pattern

As shown in Figure 18.14, the business objects are not accessed directly by the client; however, every request to the business object is made through the session bean. Figure 18.15 shows the sequence diagram depicting the interaction between various components of the Session Facade pattern:

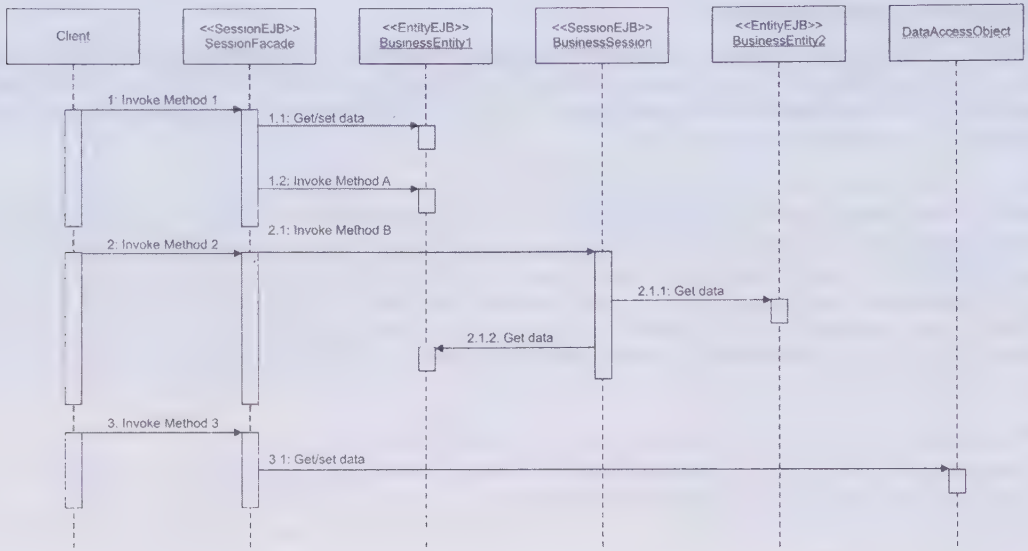


Figure 18.15: Showing the Sequence Diagram for the Session Facade Pattern

Strategies

Following strategies can be used to implement the Session Facade pattern:

- **Stateless Session Facade Strategy**—Implements a business process as a stateless session facade object by using a stateless session bean if the process is non-conversational. Being non-conversational means that the process implemented using stateless session faced strategy gets completed in a single call. The conversational state between subsequent method calls need not be saved.
- **Stateful Session Facade Strategy**—Implements the session facade as a stateful session bean object. This strategy is applied if the business process is conversational and requires multiple method invocations for completion. The conversational state between subsequent method calls need to be saved when this strategy is applied.

Consequences

The consequences of using Session Facade strategy are as follows:

- **Introduction of a Business-Tier Controller Layer**—Implies that the session facade component acts as a controller layer between the client and the application's business tier; thereby, manages all the client interactions with the application.
- **Providing a Uniform Interface**—Introduces a simple interface to the client by abstracting the underlying complex interactions.
- **Reduced Coupling and Increased Manageability**—Results in reduction of coupling between the various objects and the client.
- **Improved Performance**—Improves performance as the number of fine-grained objects is greatly reduced.
- **Providing Coarse-Grained Access**—Provides a coarse-grained abstraction of the application's functionality.
- **Centralized Security Management**—Implements security policy only at the level of the session facade.
- **Centralized Transaction Control**—Applies transaction control at the level of the session façade.
- **Exposing fewer remote interfaces to the client**—Offers a coarse-grained interface that provides higher-level abstractions to the client. This helps in reducing the large number of business objects that are otherwise exposed to the client.

The Service Locator Pattern

The Service Locator pattern uses a service locator object to centralize the look up process for distributed service components. This provides a central point of control to manage the look up services and also acts as a cache; thereby, eliminating the need for redundant look ups.

Problem

Enterprise applications need to access distributed components, such as EJB and Java Message Service (JMS) components that provide various business services. To access the distributed components, the application needs to use look up services, such as JNDI. Repeated look ups result in less readable and less maintainable code, which in turn causes performance problems.

Forces

Following forces are involved in the problem stated in the preceding section:

- The need of look up services, such as JNDI, to locate and use various distributed business components, such as connection factories and queues.
- The need to centralize the look up mechanism for locating the distributed components.
- The need to hide the involved and underlying complexities of the lookup process from the client.
- The need of repeated look up operations. These are resource-intensive and affect performance.
- The requirement of re-establishing the connection to an enterprise bean instance that has been accessed previously.

Solution

The solution to the problem stated in the previous section can be obtained by providing a service locator object that offers a centralized mechanism for the look up process to locate various requested distributed components required by the client. The service locator object abstracts the underlying complexities involved in the look up process. It also provides caching facilities; thereby, providing better management for repeated look up operations. Figure 18.16 shows the class diagram representing the Service Locator pattern:

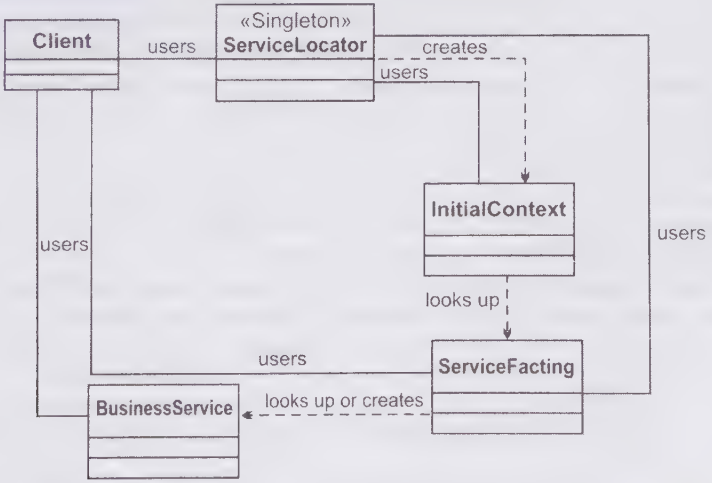


Figure 18.16: Showing the Class Diagram Representing the Service Locator Pattern

Figure 18.17 shows the sequence diagram depicting the interaction between various components of the Service Locator pattern:

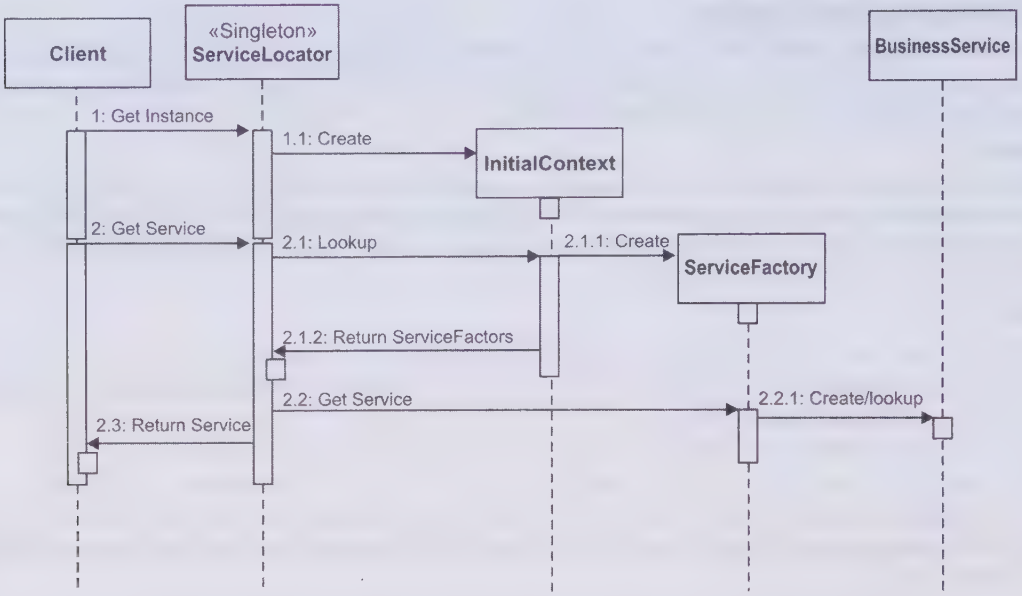


Figure 18.17: Showing the Sequence Diagram of the Service Locator Pattern

Strategies

The Service Locator pattern can be implemented using the following strategies:

- **EJB Service Locator Strategy**—Specifies that the service locator component uses an EJBHome object for implementing the ServiceFactory component. After obtaining the EJBHome object, the service locator component caches it for future use, so that the EJBHome object can be easily accessed by the client on next request; thereby, avoiding another JNDI look up. The EJBHome object can either be returned to the client who can use it for performing look up operations, or for creating, and removing enterprise beans. Alternatively, the EJBHome object can also be retained by the service locator component for later use.
- **JMS Queue Service Locator Strategy**—Specifies that the service locator uses QueueConnectionFactory objects to implement the ServiceFactory component. The QueueConnectionFactory object is obtained using its JNDI name. The service locator caches QueueConnectionFactory to use it in future, which avoids multiple look ups to be applied when QueueConnectionFactory is required again by the client. The QueueConnectionFactory object can be returned by the service locator to the client, who can use it to create a QueueConnection. The QueueConnection object is required for obtaining a QueueSession object, and creating Message, QueueSender, or QueueReciever objects.
- **JMS Topic Service Locator Strategy**—Specifies that the service locator uses TopicConnectionFactory objects to play the role of the ServiceFactory component. The TopicConnectionFactory object is obtained using its JNDI name. The service locator caches TopicConnectionFactory to use it in future when client requires it. This helps in avoiding repeated JNDI look up calls. The TopicConnectionFactory object can be returned back by the service locator to the client, who can use it for creating a TopicConnection. The TopicConnection object is required for obtaining a TopicSession, and creating the Message, TopicPublish, or TopicSubscribe objects.
- **Combined EJB and JMS Service Locator Strategy**—Combines the EJB and JMS strategies for providing service locator for all objects including enterprise beans and JMS components.
- **Type Checked Service Locator Strategy**—Allows the clients to pass constants instead of string names for JNDI service name and EJBHome class object to the getHome() method as arguments.

- **Service Locator Properties Strategy**—Allows property files or deployment descriptors to be used for specifying JNDI names and EJBHome class name.

Consequences

The consequences of using Service Locator pattern are as follows:

- **Abstracts Complexity**—Encapsulates the complex functionality of look up and creation process; thereby, hiding the details from the client.
- **Provides a Uniform Service Access to clients**—Specifies that the clients use a precise and user-friendly interface that allows all type of clients to uniformly access the business objects. This uniformity helps in better maintenance and development.
- **Facilitates the addition of New Business Components**—Allows the addition of new EJBHome objects for the enterprise beans that are created at a later stage during development and deployment.
- **Improves Network Performance**—Improves performance by aggregating the network calls that are required for locating and creating business objects.
- **Improves Client Performance by providing Caching**—Provides caching facilities for caching initial context objects and factory object's references to avoid redundant JNDI look up activity.

The Data Access Object Pattern

The Data Access Pattern separates the data processing logic from the data access logic. It abstracts the code for data retrieval and encapsulates it separately from the rest of the data manipulation code using a Data Access Object (DAO). The separation of code for accessing the data from the code for manipulating the data allows the data access logic to be modified or updated independently without affecting the code that uses the data.

Problem

In many cases, applications that provide code, which depends on a specific feature of data resources, combine the data processing logic with data access logic. Sometimes, applications need to access data from different resources. The data access code, in such cases, also varies according to the source of data. For example, the code for accessing data from a relational database will be different than the code that accesses data from a flat file. However, when the data processing code is combined with data access code, the code that accesses the data cannot be flexibly replaced or modified as and when the source of data changes.

Forces

Following forces are involved in the problem stated in the preceding section:

- Implementing the code for accessing the data into a persistent storage.
- Providing a uniform data access Application Programming Interface (API) and persistent storage APIs to various data sources.
- Separating persistent storage implementation from the rest of the code.
- Managing the data access logic by encapsulating it into a separate code block; thereby, facilitating maintainability and portability.
- Allowing the code to be flexibly modified and being adaptable when there is a change in the data source, storage type, or product vendor type.

Solution

The solution to the problem stated in the previous section lies in using a DAO that abstracts and encapsulates the logic for accessing the data from various data sources. DAO manages the establishment of a connection with the respective data source. It also fetches and stores the data that can be further used by some other processing block for manipulation and further processing. DAO hides the entire implementation details from the clients. The clients always see a uniform interface that does not change with the change in the data source. Therefore, the Data Access Object pattern allows DAO to flexibly adapt to the changes in the data sources without affecting the clients or the rest of the code. Figure 18.18 shows the class diagram for the Data Access Object pattern:

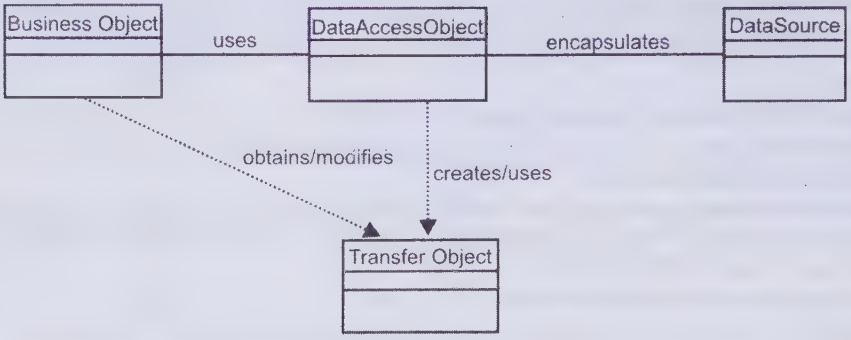


Figure 18.18: Showing the Class Diagram Representing the Data Access Object Pattern

Figure 18.19 shows the sequence diagram for the Data Access Object pattern depicting the relationship between its various participants:

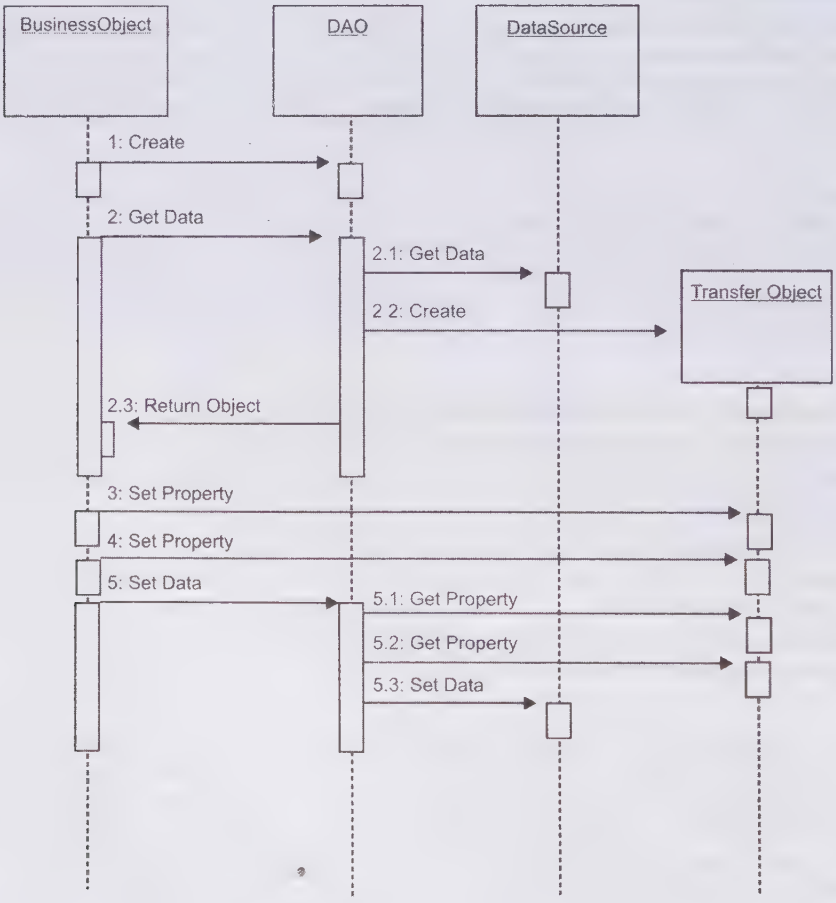


Figure 18.19: Showing the Sequence Diagram Representing the Data Access Object Pattern

Strategies

The Data Access Object pattern can be implemented by applying the following strategies:

- **Automatic DAO Code Generation Strategy**—Implements an application-specific code-generation utility to generate the code for all DAOs that the application requires. To implement this strategy, a relationship between the BusinessObject component, the DAO component, and the underlying implementation needs to be established.
- **Factory for Data Access Objects Strategy**— Uses Abstract Factory and Factory Method patterns to create DAO. This strategy is implemented using the Factory Method pattern to create the DAOs required by the application when the storage does not change between multiple implementations. When the storage is expected to change between various implementations, this strategy is implemented using the Abstract Factory pattern, which in turn uses the Factory Method pattern.

Consequences

The following consequences are observed when the Data Access Object pattern is used:

- **Enables Transparency**—Hides implementation details in DAO; thereby, providing a transparent access to the data source without revealing its specific implementation details.
- **Enables Easier Migration**—Allows the clients to switch between various database implementations.
- **Reduces Code Complexity in Business Objects**—Simplifies the code in business objects that use DAOs to fetch data for processing.
- **Centralizes Data Access into a Separate Layer**—Separates the data access code from the rest of the code of the application; thereby, allowing better management and maintenance of the application.
- **Adds Extra Layer**—Creates an additional layer between the data source and the data client that requires additional effort for implementation.

The View Helper Pattern

The View Helper pattern enforces separation of presentation and business logic. In this pattern, the presentation and formatting logic is handled by the view and the processing and data retrieval logic is handled by the View Helper class. JSP pages are used to create the views for handling the presentation part of the application, while JavaBeans helper classes are created to manage the business logic for the application. The separation of presentation and business logic promotes code reusability and makes the application more manageable.

Problem

The presentation logic of an application changes very often. However, it becomes very difficult to modify or change the presentation logic when it is tightly mingled with the code handling the business logic for the application. This is because when the code for business logic and presentation logic is tightly intermingeled, then any changes to be applied on the code handling the presentation logic also require that considerable relevant changes be made in the associated code handling the business logic. Therefore, the application becomes less manageable and flexible to change. This also results in a poor separation of roles between the Web developers who write the code for presentation logic and the software developers who write the code for business logic.

Forces

The following forces are involved in the problem stated in the preceding section:

- Avoiding the embedding of business logic in the view handling the presentation logic because that promotes copy-and-paste type of reuse which results in maintenance problems and bugs.
- Promoting a clear separation of roles between the Web developers and the software developers.
- Using one view to handle the responses to a specific business request.

Solution

The solution to the problem stated in the previous section is to provide a clear separation between the presentation logic and the business logic. The presentation logic is provided by the views which are JSP pages in most of the cases. The views handle formatting and presentation of content, and delegate the responsibility of

processing to the helper classes. The helper classes, normally implemented as JavaBeans or custom tags, handle the business logic for the application. These classes also store any intermediate data for the views. This separation of presentation logic from business logic makes the code more manageable and flexible to change as any change in the presentation logic can be made without affecting the business logic for the application. The View Helper pattern also provides a clear separation of roles between the Web developers and the software developers. Figure 18.20 shows the class diagram representing the View Helper pattern:

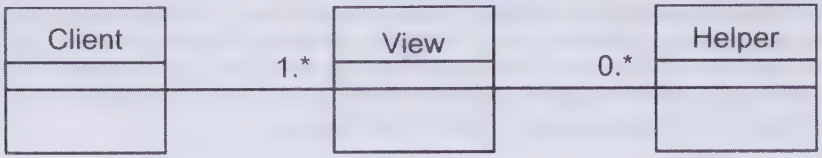


Figure 18.20: Showing the Class Diagram Representing the View Helper Pattern

Figure 18.21 shows the sequence diagram for the View Helper pattern depicting the interaction between its various participants:

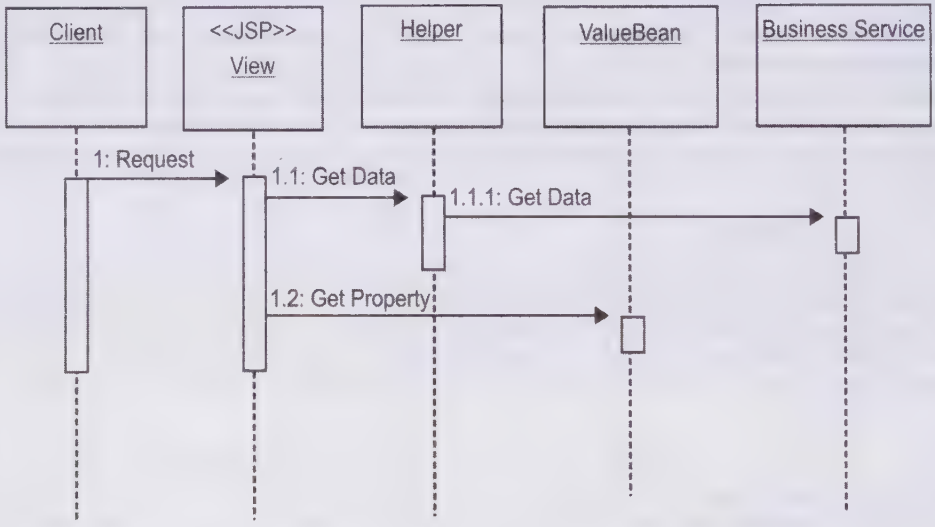


Figure 18.21: Showing the Sequence Diagram for the View Helper Pattern

Strategies

Following strategies can be applied for implementing the View Helper pattern:

- **JSP View Strategy**—Specifies that a view is implemented as JSP page. This strategy is preferred over the Servlet View strategy. JSP pages are used more commonly and provide a more elegant solution for creating the views as it is easier to code JSP pages as compared to creating servlets.
- **Servlet View Strategy**—Uses a servlet for implementing a view. However, it is more difficult to code servlets because servlets have mark up tags embedded directly within Java code. This makes it very difficult to modify and update the code.
- **JavaBean Helper Strategy**— Uses a JavaBean to implement a helper. The JavaBean helper encapsulates the code for implementing the business logic. Using JavaBeans to implement helper classes makes the code more manageable. Moreover, it is easier to create JavaBeans.
- **Custom Tag Helper Strategy**—Uses a custom tag component to implement the helper that encapsulates the business logic. However, developing custom tags is a complicated task as compared to creating JavaBeans.

To use the Custom Tag Helper strategy, the environment is required to be configured with various tags, tag library descriptors, and configuration files.

- **Business Delegate as Helper Strategy**—Specifies that a Business Delegate component should be introduced between the helper and the business tier. This allows the helper to directly invoke a business service without worrying about its implementation details. The business delegate can be implemented as a JavaBean and can be considered as a specialized type of helper.
- **Transformer Helper Strategy**—Implements the helper as an extensible Stylesheet Language Transformer.

Consequences

The consequences of using the View Helper pattern are as follows:

- **Improves reusability, manageability and application partitioning**—Makes the application more manageable by clearly separating business logic from presentation logic. This also improves code reusability and maintainability.
- **Improves Separation of Roles**—Provides clear role separation between the Web developers who code the presentation logic for the views and the software developers who code the business logic for the view helper classes.

The Dispatcher View Pattern

The Dispatcher View pattern considers the problems solved by the Front Controller and View Helper patterns, and combines the controller and the dispatcher components with the views and helpers for handling client requests and providing a dynamic response. In this strategy, the dispatcher handles view navigation and management. The dispatcher can be encapsulated by a controller, a view, or some other component. The controllers do not directly delegate the activity of content retrieval to the handlers. The activities are postponed until views are processed.

Problem

The problems considered in case of the Dispatcher View pattern comprise the problems considered in the Front Controller and View Helper patterns. The applications usually have the code for presentation logic intermingled with that of the business logic. This reduces the flexibility of applications to adapt to changes and modifications; thereby, making them less manageable. In addition, a central component for handling access control, view management, content retrieval, and request handling is not available. Moreover, applications usually have redundant code scattered across multiple views.

Forces

Following forces are involved in the problem stated in the preceding section:

- Complete authentication and authorization checks performed per request
- Minimized scriptlet code within views
- Encapsulation of business logic in separate components other than views
- Simplified control flow based on values supplied with request
- Limited and simpler view management logic

Solution

The solution to the problem stated in the previous section lies in combining the controller and the dispatcher with the views and helpers for handling the client requests and preparing a dynamic presentation as the response. In this strategy, the controllers do not delegate the content retrieval activities directly to the helpers. The dispatcher plays a minimized role and forwards the request to the views on the basis of input received from the controller; thereby, handling view navigation and management. On processing, the views handle the request and generate a response by accessing the helper classes. Figure 18.22 shows the class diagram for the Dispatcher View pattern:

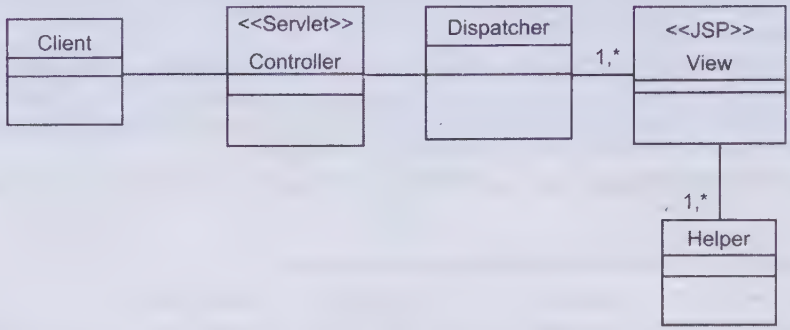


Figure 18.22: Showing the Class Diagram Representing the Dispatcher View Pattern

Figure 18.23 shows the sequence diagram for the Dispatcher View pattern depicting the interactions between its various participants:

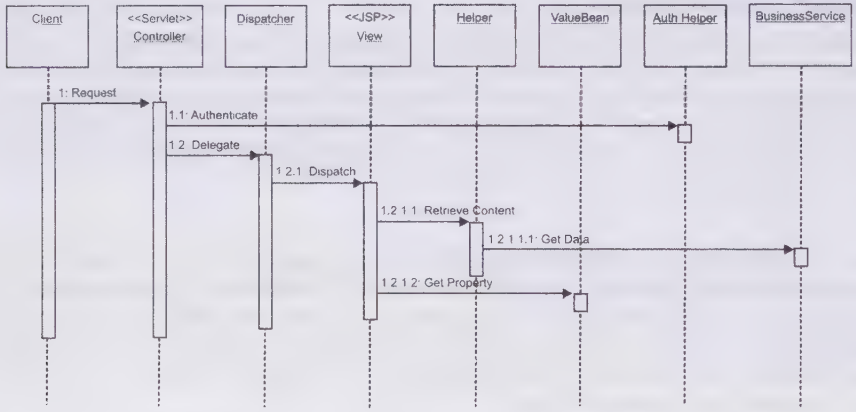


Figure 18.23: Showing the Sequence Diagram for the Dispatcher View Pattern

Strategies

The Dispatcher View pattern can be applied by using various strategies which have already been discussed in the previous sections. They are listed as follows:

- Servlet Front Strategy
- JSP Page Front Strategy
- JSP View Strategy
- Servlet View Strategy
- JavaBean Helper Strategy
- Custom Tag Helper Strategy
- Dispatcher in Controller Strategy
- Transformer Helper Strategy

Besides the preceding strategies, another strategy that can be used for implementing the Dispatcher View pattern is described as follows:

- **Dispatcher in View Strategy**—Specifies that in case when the controller has a limited role and therefore, is removed from the application implementation; then the dispatcher can be moved into a view. This design is

found useful when there is one view that is mapped to a particular request while a secondary view can be used infrequently.

Consequences

The consequences of using Dispatcher View pattern are as follows:

- **Centralizes control thereby improving reusability and maintainability**—Provides a centralized control for the process of request handling and dispatching; thereby, enhancing code reusability and making the application more manageable.
- **Improves application partitioning**—Provides clear separation between the presentation logic coded in views and the business logic coded in helpers.
- **Improves the separation of roles**—Provides clear role separation between Web developers who code the views and the software developers who code the helper classes.

The Service To Worker Pattern

The Service to Worker pattern shows various similarities with the Dispatcher View pattern. This pattern also deals with the problems considered in the Dispatcher View pattern; however, the solution is obtained and implemented slightly differently. In the Service to Worker pattern also, the focus is to deal with the intermingling of presentation and business logic as well as the need to implement a central component for managing content retrieval, access control, and views.

Problem

As stated for the Dispatcher View pattern in the previous sections, the Service to Worker pattern also focuses on the problems considered in the Front Controller and View Helper patterns. There is usually no clear separation between the presentation logic and business logic in the applications. The code providing presentation logic is mixed with the code representing the business logic. This makes the applications less flexible to adapt to new changes and modifications. Moreover, applications usually do not have a centralized component that can effectively handle access control, view management, content retrieval, and request handling. Code duplication in views is another issue that needs to be considered.

Forces

Following forces are involved in the problem stated in the preceding section:

- Executing particular business logic for servicing a request to fetch content that can be used for generating a dynamic response
- Managing view selection that may depend on the responses generated from invoking various business services
- Minimizing scriptlet code within views
- Encapsulating business logic in separate components other than views
- Generating content dynamically
- Using multiple views for responding to similar requests

Solution

As in case of the Dispatcher View pattern, the solution in this case is also obtained by combining the controller and the dispatcher with the views and helpers for handling the client requests and preparing a dynamic presentation as the response. However, in the Service to Worker pattern, unlike the Dispatcher View pattern, the controller delegates the content retrieval activities directly to the helpers. The dispatcher, in this case, submits the request to the appropriate view; thereby, handling view navigation and management. The views then handle the request and generate a dynamic response accordingly with the help of the helper classes. In the Service to Worker pattern, the dispatcher plays a comparatively larger role as compared to the Dispatcher View pattern, and can be implemented as a separate component or within a controller. Figure 18.24 shows the class diagram for the Service to Worker pattern:

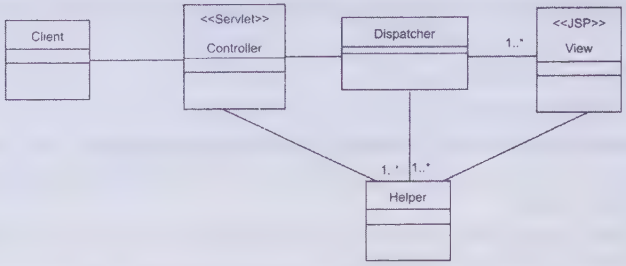


Figure 18.24: Showing the Class Diagram Representing the Service to Worker Pattern

Figure 18.25 shows the sequence diagram depicting the relationship between various participants of the Service to Worker pattern:

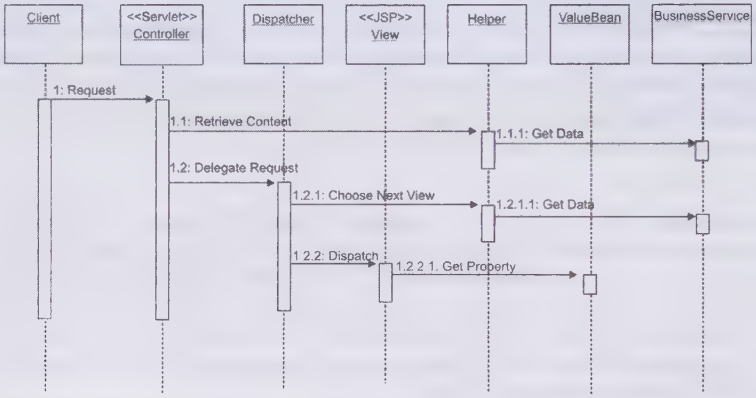


Figure 18.25: Showing the Sequence Diagram for the Service to Worker Pattern

Strategies

The Service to Worker pattern can be applied by using various strategies, which have already been discussed in the previous sections. They are listed as follows:

- Servlet Front Strategy
- JSP Page Front Strategy
- JSP View Strategy
- Servlet View Strategy
- JavaBean Helper Strategy
- Custom Tag Helper Strategy
- Dispatcher in Controller Strategy
- Transformer Helper Strategy

Consequences

The consequences of using the Service to Worker pattern are as follows:

- **Centralizes control thereby improving modularity and reusability**—Provides a centralized control for the process of handling business services and processing across multiple requests. The Service to Worker pattern also provides improved modularity by allowing clearer partitioning between various components and moving common code into controllers, helpers, and views.
- **Improves application partitioning**—Provides clear separation between the presentation logic encapsulated in views and the business logic encapsulated in helpers.

- **Improves the separation of roles**—Provides clear role separation between Web developers who code the views and the software developers who code the helper classes. This separation also reduces dependencies among the people working on the application.

This completes the discussion about design patterns. Before finishing the chapter, let's quickly summarize the topics covered throughout the chapter, in the next section.

Summary

In this chapter, you have learned about the Java EE application architecture. Next, the chapter introduced the concept of design patterns. Further, it discussed the role of design patterns. You have also learned the different types of design patterns, including the problems faced, forces involved, solution implemented by the pattern, strategies that can be applied to implement the pattern, and the consequences observed when the various patterns are applied.

The next chapter discusses Service Oriented Architecture (SOA) using Java Web Services in detail.

Quick Revise

Q1. What are the benefits of using the Data Access Object pattern?

A. The type of the actual data source can be specified at deployment time.

B. The data clients are independent of the data source vendor API.

C. It increases the performance of data-accessing routines.

D. It allows the clients to access the data source through EJBs.

E. It allows resource locking in an efficient way.

Ans. A, B

Q2. Which design pattern allows you to separate presentation and business logic into distinct components?

A. View Helper

B. Front Controller

C. Service Locator

D. Session Facade

Ans. A

Q3. Which filter allows the pre-processing and post-processing of requests and responses, respectively?

A. Session Facade

B. Front Controller

C. Intercepting Filter

D. Composite Entity

Ans. C

Q4. Which of the following is true in case of the Transfer Object design pattern?

A. The Transfer Object pattern is also termed as Value Object pattern

B. The Transfer Object pattern provides a centralized control to the application request processing

C. A Transfer Object aggregates related attributes together to form a composite value that can be returned as a return type by the methods

D. The Transfer Object pattern defines a separation between data access and data processing logic.

Ans. A, C

Q5. What is a design pattern?

Ans. Design patterns refer to language-independent solutions provided for solving commonly recurring design problems. A design pattern first describes the problem and then the solution that can be applied

to the problem. The design patterns are implemented in the form of objects and classes that provide solutions to the problem.

Q6. What are the benefits of using design patterns?

Ans. The important benefits of design patterns are as follows:

- Provides tested solutions to frequently and repeatedly occurring problems
- Provides code reusability
- Helps in defining application architecture more efficiently and clearly
- Provides more reliability and transparency to the design of the application

Q7. What are the benefits of using the Front Controller pattern?

Ans. The Front Controller pattern provides following benefits:

- **Centralizes control**—Implements a central controller component for handling requests and implementing business logic. Therefore, requests can be easily tracked and logged.
- **Improves manageability**—Provides a single entry point for all requests, which requires fewer resources; thereby, improving manageability.
- **Improves reusability**—Moves common code to a single controller component. Therefore, code is not duplicated within multiple views; thereby, encouraging code reusability.
- **Improves role separation**—Divides responsibilities among various application components for effective role separation.

Q8. What is Session Facade?

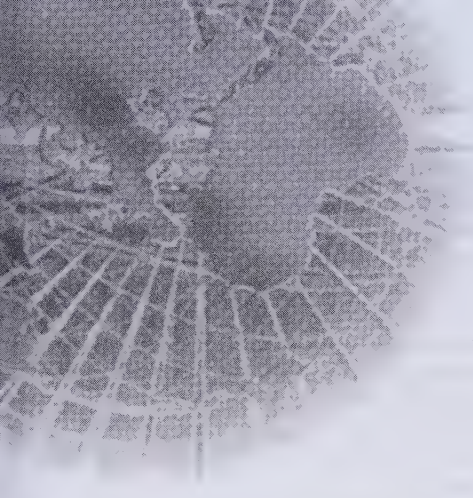
Ans. Session Facade is a design pattern that is used for developing enterprise applications. The Session Facade pattern uses a central high level component, which is implemented as a session bean and contains the functionality of the complex interactions that occur between various lower-level business components. The higher-level session bean thereby provides a single interface to the client for accessing the functionality of an application.

Q9. What is a Data Access Object (DAO)?

Ans. Data Access Object provides application components with a user-friendly and common interface to access the data from various multiple data sources. It reduces the dependency of various components on the details of the database implementations. The DAO communicates with the actual underlying database and handles the different types of data access mechanisms.

Q10. What is the use of Service Locator pattern?

Ans. The Service Locator pattern is useful for locating various business components and services. This pattern uses a service locator object to centralize the look up process for distributed service components. This provides a central point of control to manage the look up services and also acts as a cache; thereby, eliminating the need for redundant look ups.



19

Implementing SOA using Java Web Services

Service Oriented Architecture (SOA) defines how different components of a software application interact to execute the business logic of an enterprise application. In other words, SOA is a software design pattern that defines how loosely-coupled software application components interact to implement the business logic of the entire application. In SOA, the independent software application components involved in implementing the business logic are termed as software services, or simply services. These services may be defined as independent and self-sustained units of a software application that implement specific functionalities to execute the business logic. These functionalities are generally a part of the overall business logic of the software application. For example, consider a software application of a bank. The bank can provide specific functionalities, such as submitting an online registration form for opening a new bank account, and oreviewing the online bank account statement. These functionalities can be implemented using independent services. That is, we can have independent services to submit an online form and view the online bank statement. Though these services work independently, they are part of the banking software application. As a design pattern, SOA defines how these individual and independent services fit into the banking software application.

SOA primarily focuses on binding large and independent services, which normally interact through interfaces. It provides a feature of orchestration, which ensures scalability of SOA-based applications. You learn in detail about the concept of orchestration later in the chapter.

SOA is mostly implemented for a specific category of software services, known as Web services, which are interoperable software applications that work on the client/server model. As the name suggests, Web services can be accessed over a network, such as the Internet, through standard communication protocols, such as Hyper Text Transfer Protocol (HTTP). A Web service performs the standard task of an application based on the client/server model, i.e., receiving requests from a client, processing the requests, and providing the requested resources or appropriate responses. A simple E-Commerce Web site, such as eBay provides Web services for online shopping. The Java EE 6 specification defines various standards to develop and deploy Web services. This specification also specifies the guidelines to develop and deploy Web services based on SOA. In this chapter, we explore how to use these Web service specifications to implement SOA in Java EE 6 Web services. The chapter is divided into three sections: A, B, and C. Section A introduces the basic concepts of SOA and Java Web Services (JWS). Section B creates the theoretical base to understand the Java EE 6 Web service specifications that are used to implement SOA. Section C focuses on the implementation of these Web service specifications.

Section A:
Exploring SOA and Java Web Services

This section provides an overview of SOA and JWS. It also extends the discussion about implementing SOA in JWS. Implementing SOA through Web services makes data portable by ensuring consensus on the use of XML-based standards and protocols for data transmission, such as Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI). You explore these concepts in detail next.

Note that this and the next section help you to explore the concepts of SOA in relation to its implementation with JWS.

Overview of SOA

SOA integrates the business logic of applications, which are modularized into loosely-coupled Web services. These Web services are consumed by client applications to implement specific business logic. A basic model of a software application based on SOA is shown in Figure 19.1:

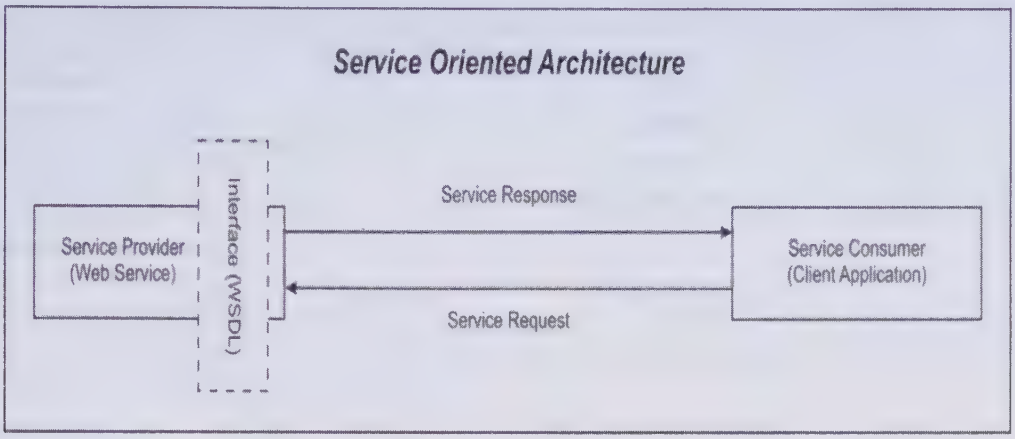


Figure 19.1: Showing a Software Application Model based on SOA

Figure 19.1 shows that the software application implementing SOA works on the client/server model. This type of application generally contains a Web service, which is hosted on the server side and acts as the service provider. Service requests are sent to the service provider, which processes and returns a response message or requested resources back to the client application, also known as the service consumer. For example, consider a simple Web service used to convert currencies. The Web service contains the logic to accept the input (in the form of a currency) and convert it into the required format (in terms of the other currency). This Web service, hosted on the server side, is provided by a third party that is considered as the service provider. A Web browser is used to access the currency converter Web service and this Web browser acts as the service consumer. When the Web browser accesses the currency converter Web service to process a conversion request, the process is termed as consuming the Web service.

SOA-based applications are platform independent, i.e., service providers can implement Web services in different platforms and languages, such as .NET or J2EE, and a service consumer or client application can use the service on a different platform or language. Enterprise applications can also add in new services or upgrade the existing services for implementing new business requirements.

The key characteristics of SOA are as follows:

- Ensures that the interfaces are implemented in a standard way, as well-formed XML documents Interfaces of SOA-based applications are described using a platform-independent Extensible Markup Language (XML) document, known as a WSDL document. .
- Uses XML schemas to define a message to communicate between service consumers and providers in heterogeneous environments.
- Allows you to register SOA-based enterprise applications in a service registry so that client applications can look up and invoke them. The service registry is defined by using UDDI.

Let's now explore the SOA environment in detail in the next section.

Describing the SOA Environment

In the previous section, we saw a basic model of an SOA-based application, which had one service provider and one service consumer. However, in real-life situations, SOA-based applications might span across numerous service providers and consumers. At times, big enterprise applications might also be based on SOA. These enterprise applications must conform to the SOA environment to run and manage SOA-based applications. The SOA environment specifies the required standards and runtime containers needed to implement SOA. Figure 19.2 displays the different layers of the SOA environment:

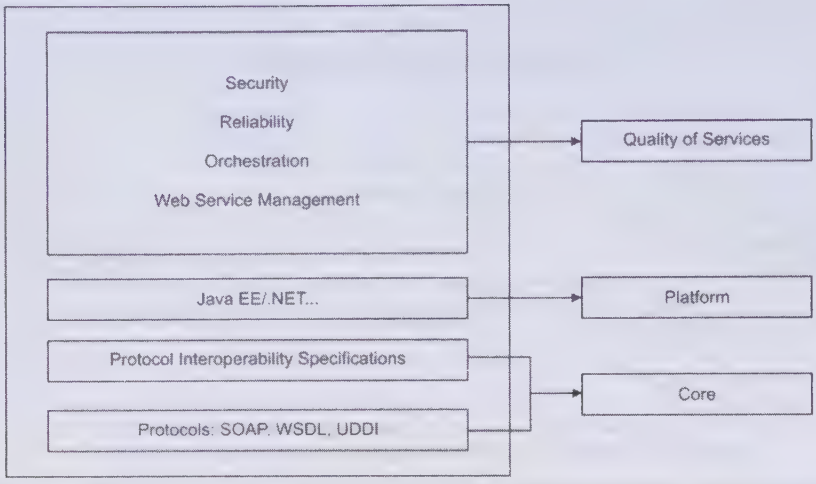


Figure 19.2: Displaying the Layers of the SOA Environment

Let's now explore these layers in detail.

The Core Layer

The Core layer of the SOA environment includes the following components:

- Protocols
- Protocol Interoperability Specification

Protocols

The protocols component includes the following protocols:

- **SOAP**—Acts as transport layer protocol to transmit messages between service clients and service providers.
- **WSDL**—Provides a format to describe the Web service interfaces. A WSDL file contains data type and message definitions, and defines how the Web services need to be bound to specific network addresses. To interact with a SOA-based Web service, service consumers need to obtain the WSDL of a Web service and then call the service by using SOAP.
- **UDDI**—Represents the protocol used by service providers to register services. Service consumers also use UDDI to search Web services in the UDDI registry.

Protocol Interoperability Specification

Web Services Interoperability (WS-I) Basic Profile is a specification that defines the standards for SOAP, WSDL, and UDDI protocols to ensure interoperability of Web services. WS-I Basic Profile is one of the core components required to test whether a Web service can be accessed across multiple platforms. The version of WS-I Basic Profile, i.e., 1.0, was released in 2004. Upgrade to version 2.0 of the specification is under progress.

The Platform Layer

The Platform layer contains different development platforms, such as Java EE and .NET, used for developing SOA-based applications. The Java EE platform provides features, such as scalability, reliability, and high performance, to the SOA environment. The Java EE specification includes Web services specifications, such as Java API for XML Binding (JAXB) to map XML documents to Java classes, in Web services. You learn about the different Java EE specifications used to develop Java EE Web services in detail, later in the chapter.

The Quality of Services Layer

The Quality of Services layer of the SOA environment caters to the requirements of enterprise applications, such as security, reliability, and transaction management. Organizations such as W3C and Organization for the

Advancement of Structured Information Standards (OASIS) have developed many specifications related to the quality of Web services, which are listed as follows:

- Security
- Reliability
- Policy
- Orchestration
- Web Service Distributed Management (WSDM)

These are described in detail next.

Security

The Web Service Security specification defines processes and methods to secure messages in various ways, such as by authenticating messages based on certain user credentials, or by using Security Assertion Markup Language (SAML) to secure Web service messages.

Reliability

Reliability means delivering a message to the recipient with an acknowledgement back to the sender and avoiding delivery of duplicate messages. The `WS-Reliability` and `WS-Reliable Messaging` standards handle reliability of Web service messages.

Policy

The WS-Policy standard defines a set of rules that an application uses to process a WS-Policy message. These set of rules are called policy standards or policy rules that must be followed to exchange Web service messages between service providers and service consumers. For example, a policy rule can implement encryption of a request to a Web service or declare the maximum acceptable size of a message by a Web service.

Orchestration

Web Service Orchestration (WSO), or simply orchestration, defines the interaction, automated arrangement, and management of different components of an SOA-based application. To be more specific, WSO defines the deployment structure of Web service components of an SOA-based application to automatically process the required business logic. Consequently, WSO also ensures that an SOA-based application is scalable, i.e., new components can be added or existing components can be modified with minimum effort as and when a business need arises. In other words, WSO can be understood as an extension of business process management that defines the interaction of individual components of a software application to process business logic. WSO is implemented by using orchestration scripts. These scripts map either to business processes or the workflow of an SOA-based enterprise application. WSO can be implemented by using different programming languages such as Business Process Execution Language for Web Services (BPEL4WS) and Web Service Choreography Interface (WSCI).

Web Service Distributed Management

The status of Web services need to be managed and monitored when the number of Web services increases in an SOA-based enterprise application. For example, you can manage and monitor the status of a Web service by verifying various parameters. Some of the parameters that can be used for managing and monitoring Web services are listed as follows:

- Examining the state of the Web service; whether or not the Web service is processing client requests
- Verifying the number of requests processed by the Web service
- Verifying the number of requests that could not be processed by the Web service due to some errors
- Verifying the number of requests that have been responded by the Web service
- Verifying the number of requests that have timed out

The status of Web services need to be monitored according to the WSDM standard, so that the WSDM-complaint management infrastructure of Web services can manage services running in a heterogeneous environment. The Web Service Coordination (WS-Coordination) and WS-Transaction specifications manage the interaction

between service providers and consumers, and handle transactions involving numerous Web services. The third-party Web service monitoring tools, such as Actional SOAPstation developed by Progress Actional, can also be integrated to manage and monitor the status of Web services in an SOA-based application.

After exploring the SOA environment, let's now proceed to the discussion about JWS.

Overview of JWS

The Web service technology is the best way to implement SOA. JWS, or simply Web services, are composed of modularized services (which encapsulate the business logic) that can be registered, searched, and called over the Internet, and provide standard interfaces for service consumers. The modular services are loosely coupled, which allows them to be accessed by anyone at any location using any platform.

NOTE

The generic term Web services is used instead of JWS throughout the chapter.

Let's take an example of a hypothetical SOA application to explain the theoretical integration between SOA and Web services. The application manages the orders placed by customers. It accepts customer orders in the form of SOAP requests and sends acknowledgements for confirmed orders in the form of SOAP responses. Let's label this application as SOA Composite App, as it is built using a set of two Web services, Purchase Order and Inventory Management. Figure 19.3 illustrates the process flow of the SOA Composite App application:

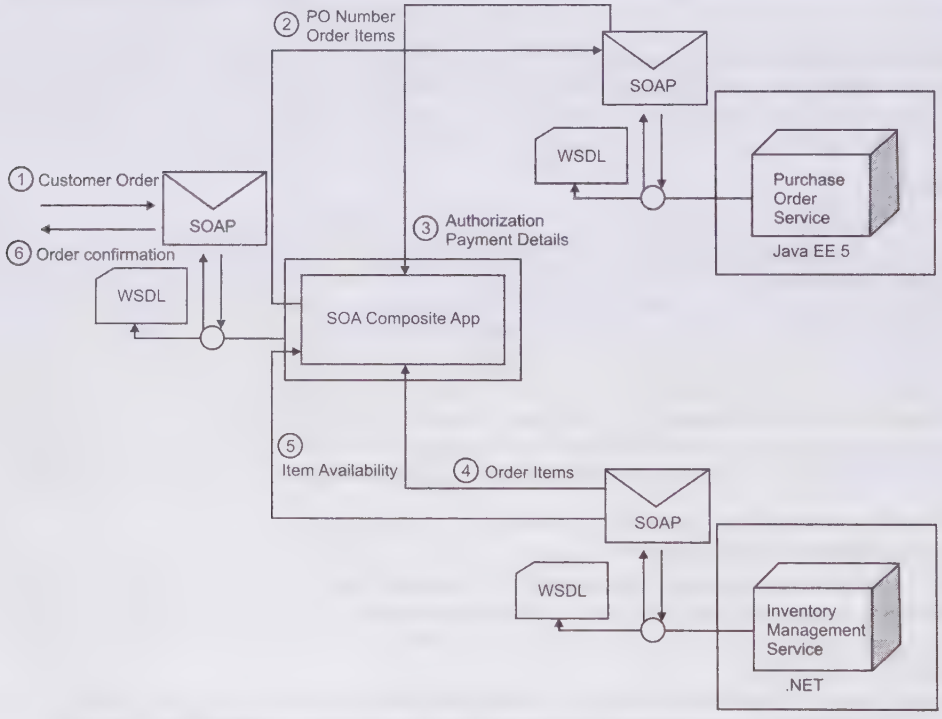


Figure 19.3: Displaying the Process Flow of the SOA Composite App Application

The process flow of the SOA Composite App application can be summarized through the following steps:

- A customer or service consumer sends an order, which contains the purchase order (PO) number and a list of products with their respective quantities. PO is sent in the form of a SOAP message.
- The PO number and list of products are entered in the `Purchase Order Service` Web service in the form of a PO. This Web service checks whether or not the PO contains all the required details. If the PO contains the required details, then the Web service prepares payment details of the order.

- The user credentials of the service consumer sending the PO are verified. If the verification succeeds, the authorization and payment details are sent to the `SOA Composite App` application.
- The list of product items is then sent to the `Inventory Management Service` Web service, which further verifies whether or not these items are available in the stock. The `Inventory Management Service` Web service then identifies the anticipated delivery dates for the product items ordered.
- The information regarding the availability of product items and their delivery dates are returned to the `SOA Composite App` application.
- Finally, an order confirmation message is sent to the customer. This message contains payment details and anticipated delivery dates of the ordered items.

As you can see in Figure 19.3, the `SOA Composite App` application uses two underlying Web services, `Purchase Order Service` and `Inventory Management Service`, to execute the business logic of the `SOA Composite App` application. It first processes the customer order with the `Purchase Order Service` Web service and then with the `Inventory Management Service` Web service. It then combines the output from both the Web services and finally creates an order confirmation message for the customer.

Role of WSDL, SOAP, and Java/XML Mapping in SOA

This section helps you to understand the role of Web service standards such as WSDL and SOAP, which are used to implement SOA by using JWS. Java/XML mapping is required to implement SOA in JWS as Java instances need to be converted to XML representations, and vice-versa. SOA-based applications are written by using XML messages and WSDL operations. As the WSDL document is a well-formed XML document, every SOA component in the SOA-based application is in the XML format. On the other hand, in the Java environment, applications are written by using classes and methods. Therefore, Java/XML mapping is required to use SOA components in the Java environment.

It is assumed that you have a basic knowledge of WSDL, SOAP, and XML; therefore, the chapter does not provide detailed information on these concepts.

NOTE

If you need to refresh your knowledge about WSDL, SOAP, and XML, you can refer to the W3C Web site, http://www.w3c.org.

Role of WSDL in SOA

WSDL is an XML-based language used by SOA components to interact with each other. A WSDL document defines the guidelines for communication between the components of an SOA-based application. Custom communication guidelines can also be provided for the interaction of SOA components. However, different enterprise applications might implement specific guidelines as per their requirements, which would reduce the scalability and reusability of SOA-based applications. In addition, these enterprise application-specific guidelines do not follow generalized standards that results in reduced interoperability of SOA-based applications. Therefore, the WSDL standard has been introduced to act as a generalized standard for communication between SOA components. This ensures enhanced scalability and interoperability of SOA-based applications. The WSDL standard specifies that XML messages be transferred over HTTP by using SOAP.

Let's take an example of a WSDL document. It describes a Web service that retrieves orders placed during a particular time period. This document accepts a date range as an input and generates an output containing the orders placed during the time period between the dates specified in the date range. Listing 19.1 shows the content of the WSDL document:

Listing 19.1: Showing the Code for a Sample WSDL Document

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://www.example.com/oeps/retrieveorders"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:oeps="http://www.example.com/oeps"
  xmlns:getord="http://www.example.com/oeps/retrieveorders"
```

```
    xmlns:faults="http://www.example.com/faults">
    <wsdl:types>
    <xs:schema targetNamespace="http://www.example.com/oeps">
    <xs:include schemaLocation="http://soabook.com/example/oms/orders.xsd"/>
    </xs:schema>
    <xs:schema targetNamespace="http://www.example.com/faults">
    <xs:include schemaLocation="http://soabook.com/example/faults/faults.xsd"
    />
    </xs:schema>
    <xs:schema elementFormDefault="qualified"
          targetNamespace="http://www.example.com/getord">
          <xs:element name="retrieveOrdersPeriod">
                <xs:complexType>
                      <xs:sequence>
                            <xs:element name="beginDate" type="xs:date"/>
                            <xs:element name="endDate" type="xs:date"/>
                      </xs:sequence>
                </xs:complexType>
          </xs:element>
          <xs:element name="retrieveOrdersPeriodResponse">
                <xs:complexType>
                      <xs:sequence>
                      <xs:element name="orders" type="oeps:OrderType"/>
                      </xs:sequence>
                </xs:complexType>
          </xs:element>
    </xs:schema>
    </wsdl:types>
</wsdl:definitions>
```

In Listing 19.1, the `wsdl:types` element contains two xs:schema definitions, which include two XML schemas located at different locations. The `targetNamespace` attribute of the first `xs:schema` element specifies that the `orders.xsd` schema belongs to the schema library of the Order Entry and Processing System Web services (oeps). Similarly, the second attribute, `schemaLocation`, specifies the location of the `orders.xsd` schema.

The second `xs:schema` element in Listing 19.1 specifies the relationship between the WSDL interface and the `faults.xsd` schema in the Web services infrastructure. The `xs:include` element is used to reference these schemas in the WSDL document. The `wsdl:types` element also contains two wrapper elements: `retrieveOrdersPeriod` and `retrieveOrdersPeriodResponse`.

A service consumer calls a WSDL document to invoke the required Web service. The call to the WSDL document is made through the SOAP protocol. Therefore, you must provide the SOAP binding declaration of a WSDL while implementing a Web service. You can provide the SOAP binding declaration by using the `wsdl:binding` element, as shown in Listing 19.2:

Listing 19.2: Showing the Code for SOAP Binding of a WSDL Document

```
<wsdl:binding name="RetrieveOrdersSOAPBinding"
    type="retrieveorder:RetrieveOrdersPort">
    <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="retrieveOrders">
          <wsdl:input>
                <soap:body use="literal"/>
          </wsdl:input>
          <wsdl:output>
                <soap:body use="literal"/>
          </wsdl:output>
          <wsdl:fault name="retrieveOrdersInFault">
                <soap:fault name="retrieveOrdersInFault"/>
          </wsdl:fault>
```

```
    </wsdl:operation>
  </wsdl:binding>
```

In Listing 19.2, the `name` attribute of the `wsdl:operation` element accepts the name of the WSDL document for which the SOAP binding declaration is provided.

Role of SOAP in SOA

SOAP is a protocol that allows SOA-based applications to exchange messages. In an SOA-based application, a service consumer sends a request to the service provider in the SOAP format. The service provider might be implemented on a platform different from the service consumer, such as Java EE 6. Therefore, the SOAP request received by a service provider is converted into the native format of the service provider. The response provided by the service provider is again converted in a SOAP message when it is received by the service consumer.

SOAP is based on XML and HTTP, and can be used across multiple platforms. It uses the Internet application layer protocol as the transport protocol to access Web services. In addition, SOAP defines the XML structure of the messages being exchanged among the SOA components. SOAP adds headers and envelopes to the message body of SOAP messages. A SOAP header provides information about the quality of service of the messaging infrastructure, such as security and reliability. The SOAP specification also provides a standard way of defining SOAP fault messages. The SOAP standard uses a processing model with multiple nodes, which can transmit and receive messages. Messages can be simply relayed from one SOAP node to the other.

The SOAP standard provides some header attributes, such as `env:mustUnderstand`, which specifies whether or not a SOAP node must process a SOAP header. If this attribute is set to true, the last SOAP node must either process the header block or return a SOAP fault message to the sender.

The example of a SOAP request message for the retrieveOrdersPeriod Web service is shown in Listing 19.3:

Listing 19.3: Showing the Code for theRetrieveOrdersPeriodSOAPRequest.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<env1:Envelope xmlns:env1="http://schemas.xmlsoap.org/soap/envelope">
  <env1:Body>
<retrieveorder:retrieveOrdersPeriod
  xmlns:retrieveorder="http://www.example.com/oms/getorders">
        <retrieveorder:beginDate>2008-03-10</retrieveorder:beginDate>
        <retrieveorder:endDate>2008-03-13</retrieveorder:endDate>
        </retrieveorder:retrieveOrdersPeriod>
  </env1:Body>
</env1:Envelope>
```

Listing 19.3 represents a SOAP request message that has no headers. The message body of the SOAP envelope contains only one element, `retrieveorder:retrieveOrdersPeriod`.

When a Web service receives a SOAP response message to this SOAP request, the response message must have the date value of the PURCHASE_DATE attribute set between the date ranges specified in the SOAP request message, as shown in Listing 19.4:

Listing 19.4: Showing the Code for a Sample SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?>
<env1:Envelope xmlns:env1="http://schemas.xmlsoap.org/soap/envelope">
  <env1:Body>
      <retrieveorder:retrieveOrdersPeriodResponse
            xmlns:retrieveorder="http://www.example.com/oeps/getorders">
            <Orders xmlns="http://www.example.com/oeps"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.example.com/oeps
            http://soabook.com/example/oeps/orders.xsd">
            <Order>
                  <OrderId>ABC1234567</OrderId>
                  <OrderHeader>
                  <SALES_ORG>Coca-Cola India</SALES_ORG>
                  <PURCHASE_DATE>2008-02-01</PURCHASE_DATE>
                  <CUST_ID>ABC0072123</CUST_ID>
                  <PAYMENT_METHOD>PO</PAYMENT_METHOD>
                  <PURCHASE_ORD_NO>PO-72123-0007</PURCHASE_ORD_NO>
```

```
                        <DELIVERY_DATE>2008-03-20</DELIVERY_DATE>
                        </OrderHeader>
                        <Products>
                        <product>
                        <PROD_ID>012345</PROD_ID>
                        <AMOUNT>50</AMOUNT>
                        <UNIT_OF_MEASURE>liter</UNIT_OF_MEASURE>
                        <PRICE_PER_UNIT>15</PRICE_PER_UNIT>
                        <DESCRIPTION>2 liter Fanta soft drink </DESCRIPTION>
                        </product>
                        <product>
                        <PROD_ID>543210</PROD_ID>
                        <AMOUNT>20</AMOUNT>
                        <UNIT_OF_MEASURE>liter</UNIT_OF_MEASURE>
                        <PRICE_PER_UNIT>21</PRICE_PER_UNIT>
                        <DESCRIPTION></DESCRIPTION>
                        </product>
                        </Products>
                        <OrderDescription>This order is rush.</OrderDescription>
                        </Order>
                    </Orders>
            </retrieveorder:retrieveOrdersPeriodResponse>
        </env1:Body>
    </env1:Envelope>
```

In Listing 19.4, the message body of the SOAP envelope contains only one element, `retrieveorder:retrieveOrdersPeriodResponse`, according to the `RetrieveOrders.wsdl` document. The content of this element is of the `oeps:OrdersType`, which is defined in the `orders.xsd` imported schema. Therefore, the structure of the SOAP message depends on both the WSDL document and the schema library of the associated Web services.

If a SOAP request is not valid, a SOAP fault message is returned to the sender, as shown in Listing 19.5:

Listing 19.5: Showing the Code for a Sample SOAP Fault Message

```
<?xml version="1.0" encoding="UTF-8"?>
<env1:Envelope xmlns:env1="http://schemas.xmlsoap.org/soap/envelope/">
  <env1:Body>
        <env1:Fault xmlns:faults="http://www.example.com/faults">
              <faultcode>env1:Client</faultcode>
              <faultstring>Invalid input message.</faultstring>
              <detail>
                    <faults:inputMessageValidationFailure
                    msg="The startDate is larger than the endDate."/>
              </detail>
        </env1:Fault>
  </env1:Body>
</env1:Envelope>
```

Listing 19.5 represents a simple SOAP fault message. The `env1:Client` value specifies that there is an error in the request message, which the Web service is unable to process. It uses the `detail` element to give specific details about the error.

Role of Java/XML Mapping in SOA

Java/XML mapping transforms a SOAP request message to a Java-compatible format to execute the SOA component in the Java environment. Let's take the example of a Web service implemented in Java EE. This Web service acts as a service provider, and can interpret messages in the Java format only. A service consumer, such as a Web browser, can interpret messages in the XML format (by using the SOAP protocol). Figure 19.4 shows the processing of a message by a service provider and a service consumer:

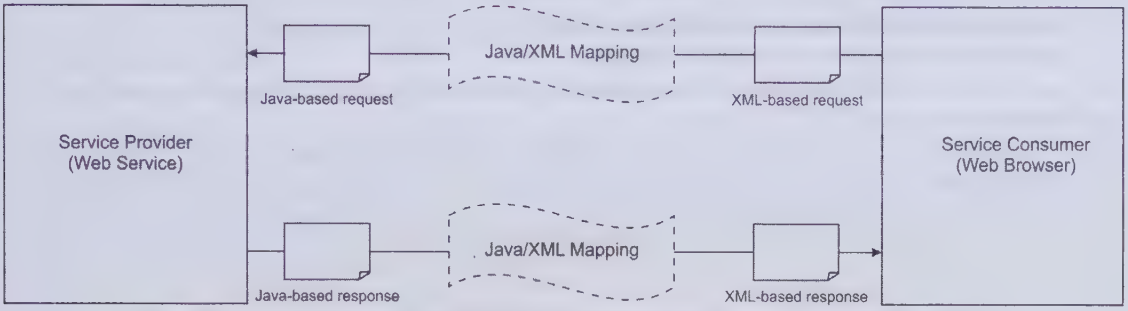


Figure 19.4: Showing the Transformation of Messages through Java/XML Mapping

Figure 19.4 shows the role of Java/XML mapping in message transformations. As you can see from Figure 19.4, an XML-based request from a service consumer is transformed into a Java-based request when it is sent to the service provider. This ensures that the service provider is able to interpret the request. Similarly, the service provider sends a Java-based response, which is transformed into an XML-based response, so that the service consumer can understand and interpret the response.

You need to determine the WSDL port for the message before associating a SOAP request message with a WSDL operation. According to the WSDL specification, the request dispatching process searches for the location of the HTTP port in the `soap:address` element of the `wsdl:port` element. From the `wsdl:port` element, you can also retrieve the `RetrieveOrdersPeriodSOAPBinding` binding, which implements the `RetrieveOrdersPeriodPort` element. It is not necessary to use the `soap:address` element inside the `wsdl:port` element as you can set the value for the `soap:address` element from the underlying HTTP request.

After determining the `wsdl:port` element, the request dispatching process determines the `wsdl:operation` element. Identifying the WSDL operation depends on the WSDL styles used in the `wsdl:binding` element. The `style` and `use` attributes of the `wsdl:operation` element define SOAP binding. These attributes describe how a Web service maps SOAP messages to WSDL operations. The `style` attribute can take two values, `rpc` or `document`, depending on the structure of the SOAP Body element, `env:Body`. The `use` attribute can also take two values, `literal` or `encoded`, depending on the serialization of the SOAP message. If the `use` attribute takes a `literal` value, the data of the SOAP message must conform to XML schema constraints defined in the WSDL's types section. If the `use` attribute takes an `encoded` value, the serialization of data is based on SOAP encoding in the SOAP 1.1 specification.

The possible values for the entire WSDL style of the WSDL document are as follows:

- rpc/literal
- Document/literal (by default unwrapped)
- Document/literal wrapped

Let's explore each WSDL style in detail and see the differences in the output of the SOAP messages when we use these styles.

The rpc/literal Style

The WSDL document based on the rpc/literal style has the following characteristics:

- The request message can have many parts and these parts must be defined by using the type attribute
- If the type attribute of any part of the request message uses complex type, it must be defined in the wsdl:types element

The SOAP message generated according to the rpc/literal style has the following characteristics:

- The root element under the env:Body element must be a wrapper element and must have the same name as that of the wsdl:operation element
- The parameter elements under the root element must have the same names as those specified in the name attribute of the wsdl:part elements

- The parameter elements are not qualified by namespace
- The name of the response element is not defined

Let's see a portion of a WSDL document named RetrieveOrdersPeriod_rpclt.wsdl, which follows the rpc/literal style, as shown in Listing 19.6:

Listing 19.6: Showing the Code for the RetrieveOrdersPeriod_rpclt.wsdl Document

```
<wsdl:types>
 <xs:schema elementFormDefault="qualified"
        targetNamespace="http://www.example.com/retrieveorder">
        <xs:import schemaLocation="http://www.example.com/oeps
        http://soabook.com/example/oeps/orders.xsd"/>
        <xs:import schemaLocation="http://www.example.com/faults
        http://soabook.com/example/faults/faults.xsd"/>
 </xs:schema>
</wsdl:types>
<wsdl:message name="req">
 <wsdl:part name="beginDate" type="xs:date"/>
 <wsdl:part name="endDate" type="xs:date"/>
</wsdl:message>
<wsdl:message name="res">
 <wsdl:part name="orders" element="oeps:OrdersType"/>
</wsdl:message>
```

The code of a SOAP request message which is written according to the rpc/literal style is shown in Listing 19.7:

Listing 19.7: Showing a Soap Request Message Based on the rpc/literal Style

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Body>
        <retrieveOrdersPeriod>
                <beginDate>2005-11-19</beginDate>
                <endDate>2005-11-22</endDate>
        </retrieveOrdersPeriod>
 </env:Body>
</env:Envelope>
```

Notice that the request message in Listing 19.7 does not contain namespaces.

The SOAP message based on the rpc/literal style is easy to understand but the contents of this SOAP message do not follow any XML schema.

The document/literal Style

The document/literal style is unwrapped; therefore, if you need to send multiple parameters in a SOAP message, the message body of the SOAP request message would contain the corresponding number of child nodes. A WSDL document based on the document/literal style has following characteristics:

- The input message can have multiple parts, and these parts must be defined using the element attribute.
- Full schema of SOAP messages needs to be written within the wsdl:types element. The element attribute of each part refers to element definition in the xs:schema element.

The SOAP message generated according to the document/literal style has the following characteristics:

- The SOAP-ENV:Body element does not need to specify the name of the wsdl:operation element.
- The parameter elements under the root element must have the same names as those specified in the name attribute of the wsdl:part elements.
- The parameter elements must be qualified by appropriate namespaces.
- Both request and response messages are unwrapped.

A portion of a WSDL document, named RetrieveOrdersPeriod_doclit.wsdl, which follows the document/literal style is shown in Listing 19.8:

Listing 19.8: Showing the Code for the RetrieveOrdersPeriod_doclit.wsdl Document

```
<wsdl:types>
 <xs:schema elementFormDefault="qualified"
        targetNamespace="http://www.example.com/retrieveorder">
        <xs:import schemaLocation="http://www.example.com/oeps
        http://soabook.com/example/oeps/orders.xsd"/>
        <xs:import schemaLocation="http://www.example.com/faults
        http://soabook.com/example/faults/faults.xsd"/>
        <xs:element name="beginDate" type="xs:date"/>
        <xs:element name="endDate" type="xs:date"/>
        <xs:element name="orders" type="oeps:OrdersType"/>
 </xs:schema>
</wsdl:types>
<wsdl:message name="req">
 <wsdl:part name="param1" element="retrieveorder:beginDate"/>
 <wsdl:part name="param2" element="retrieveorder:endDate"/>
</wsdl:message>
<wsdl:message name="res">
 <wsdl:part name="param1" element="retrieveorder:orders"/>
</wsdl:message>
```

The code for a SOAP request message which is written according to the document/literal style, is shown in Listing 19.9:

Listing 19.9: Showing the Code for a SOAP Request Message Based on the document/literal Style

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Body xmlns:getord="http://www.example.com/oeps/retrieveorders">
        <retrieveorder:beginDate>2005-11-19</retrieveorder:beginDate>
        <retrieveorder:endDate>2005-11-22</retrieveorder:endDate>
 </env:Body>
</env:Envelope>
```

The document/literal wrapped Style

The document/literal wrapped style is derived from the document/literal style by using wrapper elements around the parameter elements.

The WSDL document based on the document/literal wrapped style has the following characteristics:

- The input message must have only one part, and this part must be defined by using the element attribute.
- The single message part is a wrapper element and the full schema of the message must be written in the wsdl:types element.
- All parameters must be the immediate child nodes of the wrapper element.
- The response message contains the response in a wrapper element, and the local name of this wrapper element must be in the `wsdl:operationRS` form, where RS represents the response string.

The SOAP message generated according to the document/literal wrapped style has the following characteristics:

- The root element under the env:Body element must be a wrapper element and must have the same name as that of the wsdl:operation element.
- The root element must be qualified by the appropriate namespace.
- All parameter elements must be the immediate child nodes of the wrapper element. The local names of the parameter elements can be different from the names specified in the name attributes of the wsdl:part definition.
- The parameter elements must be qualified by the appropriate namespaces.

Listing 19.10 shows a portion of a WSDL document, named RetrieveOrdersPeriod.wsdl, which follows the document/literal wrapped style:

Listing 19.10: Showing the Code for the RetrieveOrdersPeriod.wsdl Document

```
<wsdl:types>
 <xs:schema targetNamespace="http://www.example.com/oeps">
 <xs:include schemaLocation="http://soabook.com/example/oeps/orders.xsd"/>
 </xs:schema>
 <xs:schema targetNamespace="http://www.example.com/faults">
 <xs:include schemaLocation="http://soabook.com/example/faults/faults.xsd"
 />
 </xs:schema>
 <xs:schema elementFormDefault="qualified"
        targetNamespace="http://www.example.com/retrieveorder">
        <xs:element name="retrieveOrdersPeriod">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="beginDate" type="xs:date"/>
                        <xs:element name="endDate" type="xs:date"/>
                </xs:sequence>
        </xs:complexType>
 </xs:element>

 <xs:element name="retrieveOrdersPeriodResponse">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="orders" type="oeps:OrdersType"/>
                </xs:sequence>
        </xs:complexType>
 </xs:element>
 </xs:schema>
</wsdl:types>
<wsdl:message name="req">
 <wsdl:part name="parameters" element="retrieveorder:retrieveOrdersPeriod/>
</wsdl:message>
<wsdl:message name="res">
<wsdl:part name="parameters" element="retrieveorder:retrieveOrdersPeriodResponse"/>
</wsdl:message>
```

Listing 19.11 shows the code for a SOAP request message which is written according to the document/literal wrapped style:

Listing 19.11: Showing the Code for a SOAP Request Message Based on the document/literal wrapped Style

```
<?xml version="1.0" encoding="UTF-8"?>
<env1:Envelope xmlns:env1="http://schemas.xmlsoap.org/soap/envelope">
 <env1:Body>
<retrieveorder:retrieveOrdersPeriod
 xmlns:retrieveorder="http://www.example.com/oeps/retrieveorders">
        <retrieveorder:beginDate>2008-03-10</retrieveorder:beginDate>
        <retrieveorder:endDate>2008-03-13</retrieveorder:endDate>
        </retrieveorder:retrieveOrdersPeriod>
 </env1:Body>
</env1:Envelope>
```

Let's now move ahead and explore the Web service specifications to implement SOA and the advanced features of Web services in the next sections.

Section B: Understanding Web Service Specifications to Implement SOA

Java EE has evolved as a premiere solution for server-side scripting of Web services. To ensure scalability and interoperability of Web applications, World Wide Web Consortium (W3C) has introduced various standards for Web services. These standards have also been integrated in the Java EE 6 specification, which includes various Web service specifications, such as WSEE 1.3, JAX-WS 2.2, and JAXB 2.2. The main purpose of Java Web service specifications is to make the development and deployment of Web services easy.

NOTE

W3C is a standards organization that defines the specifications for creating, configuring, deploying, and managing Web services.

Table 19.1 lists the Web service specifications included in the Java EE 6 specification:

Table 19.1: Web Service Specifications of Java EE 6

Specification	Version	Java Specification Request (JSR) ID No.
Java Application Programming Interface (Java API) for XML based Web Services	JAX-WS 2.2	JSR 224
Java Architecture for XML Binding	JAXB 2.2	JSR 222
Implementing Web Services	WSEE 1.3	JSR 109
Web Service Metadata for the Java platform	WS-Metadata 2.0	JSR 181
Simple Object Access Protocol (SOAP) with Attachments API	SAAJ 1.3	JSR 67
Java API for XML Registries	JAXR 1.0	JSR 93
Streaming API for XML	StAX 1.0	JSR 173

The succeeding sections provide a detailed description of the Web service specifications.

Exploring the JAX-WS 2.2 Specification

The JAX-WS 2.2 specification provides a Java API that you can use to create Web services. The primary objective of JAX-WS 2.2 is to simplify the development and deployment of clients and endpoints of Web services. An endpoint refers to an interface of a service provider or consumer, exposed as the destination of a SOAP message. The JAX-WS Web services specification can be divided into three categories: invocation, serialization, and deployment. These categories define specific standards and sub-specifications to invoke, serialize, and deploy Web services. Table 19.2 lists the sub-specifications of the JAX-WS Web service specification:

Table 19.2: Sub-Specifications of the JAX-WS Web Service Specification

Invocation Sub-Specification	Serialization Sub-Specification	Deployment Sub-Specification
Web service invocation with Java interface proxies	WSDL styles	Java/WSDL mapping
Web service invocation with XML		Static WSDL
Message context		XML service providers

Table 19.2: Sub-Specifications of the JAX-WS Web Service Specification		
Invocation Sub-Specification	**Serialization Sub-Specification**	**Deployment Sub-Specification**
Handler framework		XML catalogs
SOAP binding		Run-time endpoint publishing
HTTP binding		
Converting exceptions to SOAP faults		
Asynchronous invocation of Web service		
One-way operation		
Client-side thread management		
Pseudo reference passing		

Let's discuss each of these briefly.

The Invocation Sub-Specification Category

The Invocation sub-specification category defines the standards to invoke Web services. In addition, this sub specification category defines the standards related to transmission and processing of messages between We services. Some provisions offered by this sub-specification category are:

- Web service invocation with Java interface proxies
- Web service invocation with XML
- Message context
- Handler framework
- SOAP binding
- HTTP binding
- Conversion of exceptions to SOAP faults
- Asynchronous invocation
- One-way operations
- Client-side thread management
- Pseudo reference passing

Let's explore these in detail.

Web Service Invocation with Java Interface Proxies

JWSs are invoked by static or dynamic service instances by using the service endpoint interface (SEI) as a prox You can use the `Service.getPort()` method to create an instance of the Web SEI. This instance correspond to the `wsdl:port` element of a WSDL document of the Web service. The SEI must follow the JAX-W Java/WSDL and JAXB XML/Java mapping.

Web Service Invocation with XML

You can also invoke Web services by sending and receiving XML messages. In order to do so, you need invoke the `createDispatch()` method, which returns a `javax.xml.ws.Dispatch` instance on an instance the Web service. You only need to build SOAP messages and send them to the Web service instance.

Message Context

A message context specifies the namespaces and metadata about the elements of an XML message. The instan of the `javax.xml.ws.handler.MessageContext` class represents the message context. JAX-WS allow handlers, endpoints, and clients to access and modify the message context of XML request/response message Let's consider an example of message sequencing to understand the use of the message context:

- When multiple SOAP messages are sent to a destination, the request handler extracts the sequence of messages from the SOAP header elements
- The endpoint then processes the messages concurrently and sends the responses after processing all the previous messages in the sequence
- The sequence of responses can be changed according to the sequence information specified in the message context

Endpoint implementations access the message context with the help of Dependency Injection. (To learn more about dependency injection, refer to *Chapter 13, Working with EJB 3.1, Chapter20, Working with Struts2.1,* and *Chapter 21, Working with Spring3.0.)* The clients can access the message context by invoking the `getRequestContext()` and `getResponseContext()` methods of the `javax.xml.ws.BindingProvider` interface.

Handler Framework

The JAX-WS 2.2 specification defines request and response handlers to post-process requests and pre-process responses. You can use these handlers at both the client and the server side. For example, you can use a handler to post-process a client message to be sent to a Web service, and append some headers to the message. You can use another handler on the server side to pre-process the response messages and verify the appended headers to the response messages. You can configure a chain of handlers at runtime by using either deployment metadata with the `@HandlerChain` annotations or the `HandlerResolver` interface.

Two types of handlers are supported by JAX-WS namely, protocol and logical. Protocol handlers can access the entire structure of a SOAP message. However, logical handlers, can only access the information about the payload of a message by using either the `javax.xml.transform.Source` interface or the instance of the JAXB annotated class.

SOAP Binding

SOAP binding refers to the process of setting the message addressing properties in the SOAP header of a SOAP message to simplify the transmission of the SOAP message from one endpoint to other. W3C standards define detailed specifications for SOAP binding in terms of notational conventions and XML namespaces. The following properties of the SOAP header must be in conformation with the specifications provided by W3C:

- **Destination**—Specifies the destination property of a SOAP header. Destination represents the destination endpoint of the corresponding SOAP message.
- **SourceEndpoint**—Defines the source endpoint property of the SOAP header. The source endpoint specifies the endpoint from which the corresponding SOAP message was sent.
- **ReplyEndpoint**—Represents the reply endpoint property of the SOAP header. The reply endpoint represents the endpoint to which the SOAP response for the corresponding SOAP message must be sent.
- **FaultEndpoint**—Specifies the fault endpoint property of the SOAP header. If a SOAP message contains a fault and cannot be delivered to the destination, it is sent to the fault endpoint.
- **Action**—Represents the action property of the SOAP header.
- **MessageID**—Specifies the message ID property of the SOAP header.
- **Relationship**—Defines the relationship property of the SOAP header.
- **ReferenceParameters**—Specifies the reference parameters property of the SOAP header.

The JAX-WS 2.2 specification conforms to the specifications of W3C for SOAP binding.

HTTP Binding

The JAX-WS 2.2 specification provides support for XML/HTTP binding to deploy and consume RESTful Web services, which send and receive XML messages over the HTTP protocol, instead of using the SOAP protocol.

Conversion of Exceptions to SOAP Faults

The JAX-WS 2.2 specification provides mapping among the `java.lang.Exception` instances to SOAP fault messages. Consider the example of Java/XML mapping described earlier. As stated previously, the SOAP requests originating from a service consumer need to be mapped to Java, so that the service provider can interpret the SOAP requests. This mapping is required in case of Java exceptions as well. JWSs are implemented

in Java, and can raise exceptions due to various reasons. If the service consumer receives a Java exception, it would not be able to interpret the same. To ensure that exceptions are appropriately interpreted and handled by the service consumers, Java exceptions must be converted to the corresponding SOAP faults.

In earlier specification of JAX-WS, you had to map service exceptions to SOAP faults by writing the code for the same. In the JAX-WS 2.2 specification; however, the `WebFault` annotation maps service exceptions to SOAP fault messages and you do not need to write discrete code for the same.

At times, exceptions in Web services might not be Java-specific and might arise due to reasons, such as faulty construction of SOAP messages. Therefore, we need to convert these exceptions into SOAP fault messages. You can also convert application-specific exceptions to SOAP fault messages. However, you must ensure that Java runtime exceptions are not converted into SOAP fault messages.

Asynchronous Invocation

The JAX-WS 2.2 specification supports asynchronous invocation. In asynchronous invocation, the Web service being called does not wait for the request or response to be delivered before performing other operations. That is, in asynchronous invocation, the Web service can simultaneously perform other operations, which are not dependent on the request or response being called, while the request or response is being delivered to the specified destination. Consequently, the request or response performance of the application using the Web service is enhanced. JAX-WS facilitates asynchronous request-response communication by using two techniques: polling and callback. In polling, a client checks frequently to verify whether a response has arrived. In callback, a handler processes the response asynchronously when the response is available.

One-Way Operations

The JAX-WS 2.2 specification allows you to map Java methods to WSDL one-way operations. The WSDL of a Web service provides four types of operations that an endpoint can support, which are listed as follows:

- **One-Way**—Specifies that an endpoint needs to only receive a message
- **Request-Response**—Specifies that an endpoint must receive a message and send a response to the message
- **Solicit-Response**—Specifies that an endpoint must send a message and receive a response for the message
- **Notification**—Specifies that an endpoint only needs to send a message

These operations allow developers to use reliable messaging protocols, such as Web services reliable messaging, to implement the business logic using Web services.

Client-Side Thread Management

The JAX-WS 2.2 specification provides the option to set the `java.util.concurrent.Executor` instance inside the `javax.xml.ws.Service` instance to manage multiple threads that invoke a Web service. This type of thread management is usually done during asynchronous Web service invocation.

Pseudo Reference Passing

When you pass an instance of a Java class to a Java method call and assign this call to an object, the object gets the reference of the passed instance, and not a copy of the instance. When we make a call to a Web service by passing an instance, the copy of this instance is serialized, wrapped in a SOAP message, and sent over the network.

You cannot pass a reference of an instance of a Java class residing on your local address space by using SOAP. The JAX-WS specification supports a pseudo reference passing mechanism to pass references of these instances. The `Holder<name>` class holds the reference of the Java class name. Now, when you invoke a Web service, JAX-WS sends a duplicate copy of the instance of the `Holder<name>`class to the Web service and returns the changed instance of the `Holder<name>` class. The `Holder<name>` instance also stores the changed instance of the `Holder<name>` class received from a Web service.

The Serialization Sub-Specification Category

The serialization sub-specification category defines the standards for serializing the Web services. These standards are defined in the WSDL of a Web service. JAXB serializes request and response parameters; however, JAXB needs some instructions to package the serialized parameters into a SOAP message. The WSDL styles are

used to specify these instructions. For example, serialized parameters are packaged either as separate child nodes of the SOAP body element or as a single element child node that contains all parameters.

The JAX-WS 2.2 specification supports commonly used WSDL styles, such as rpc/literal and document/literal wrapped. You can specify the rpc style by using the `javax.jws.SOAPBinding` annotation with properties, such as style of RPC, LITERAL, and parameterStyle. You can specify the document wrapped style by using the `javax.jws.SOAPBinding` annotation with properties, such as style of DOCUMENT, LITERAL, and parameterStyle. The JAX-WS 2.2 specification uses request and response beans to wrap request parameters and the response value.

The Deployment Sub-Specification Category

The deployment sub-specification category specifies the standards to deploy Web applications. These sub-specifications include the following components:

- Java/WSDL mapping
- Static WSDL
- XML service providers
- XML catalogs
- Runtime endpoint publishing

Let's explore these in detail next.

Java/WSDL Mapping

Java/WSDL mapping defines the binding between WSDL operations and Java methods. Initially, the SOAP message requests for a WSDL operation. Then, Java/WSDL mapping is used to invoke the associated Java method and map the SOAP message to the parameters of this method. Java/WSDL mapping is also used to map the return value of the method to the SOAP response.

With the help of this mapping, a developer can first create a Java class, use the JAX-WS processor (java2wsdl or wsgen utility), and create a WSDL document of a Web service end point.

You can customize standard Java/WSDL mapping with embedded binding declarations. These binding declarations are provided by using the `jaxws:bindings` extension elements.

Following are some restrictions while performing Java/WSDL mapping:

- One-to-one mapping exists between the `wsdl:portType` element and the Java interface (in our case SEI), irrespective of the number of operations in this element
- Mappings for parameters of the SEI and its return type should necessarily be compatible with JAXB

Static WSDL

A `javax.xml.ws.Service` instance represents a JAX-WS Web service client. This instance corresponds to the `wsdl:service` element of the WSDL document of a Web service. You can create service instances statically or dynamically. To generate a service instance dynamically, use the `Service.create` factory method. You can generate your own service instance from the WSDL document by using the JAX-WS processing tool. For example, the wsimport processing tool of JAX-WS is provided by the Glassfish server.

XML Service Providers

You can deploy a Web service to send and receive XML messages without JAXB binding annotations. The JAX-WS API provides the `javax.xml.ws.Provider` interface to create Web services that do not require JAXB binding annotations. JAX-WS Java/WSDL and JAXB Java/XML mappings are not used during the invocation of this type of Web services.

XML Catalogs

The JAX-WS specification supports the feature of XML catalogs. This feature allows a WSDL document to import external schemas. In this way, the same schema information need not be duplicated in different WSDL documents. XML catalogs contain the mapping information where external references to imported schemas are mapped to local instances of these schemas.

Runtime Endpoint Publishing

JAX-WS supports the publishing of Web service endpoints at runtime. The instance of the `javax.xml.ws.EndPoint` class is used to assign an instance of the Web service implementation class to a URL. Dynamic publishing of endpoints is supported only by Java SE 6, while Java EE 6 does not support dynamic publishing of endpoints.

JAX-WS runtime creates the required server infrastructure to invoke a Web service. Initially, the HTTP context is created and finally SOAP requests are listened on a specified URL. During this process, WSDL document for endpoint is created dynamically. The generated WSDL document depends on source code annotations in the class implementing the Web service or metadata files deployed during invocation of the `Endpoint.setMetadata` method. The location for accessing the generated WSDL document is specified in the publish mechanism.

Exploring the JAXB 2.2 Specification

The JAXB 2.2 specification provides the standards to bind Java classes to XML schema components. It is necessary to know the difference between certain terms, which are listed as follows, before exploring JAXB 2.2:

- **Java/XML binding**—Specifies that each Java class is mapped to a unique XML schema component depending on its annotations. The Java/XML binding is defined by using JAXB 2.2 annotations in Java classes.
- **Java/XML type mapping**—Specifies the relationship between a Java element and an XML schema component. The serializer and deserializer define this type of mapping. The serializer changes the instances of the Java class into XML instances, which correspond to a specific XML schema. The deserializer does the opposite. A type mapping framework is usually used to create layered architecture applications.
- **Java/XML map**—Consists of a collection of type mappings. The Java/XML map contains the mapping information where a Java element is mapped to XML instances of different schema types.

While implementing SOA with Web services, you need to define the mapping between existing XML schemas and Java classes. Java/XML maps are also used when there may be numerous Java classes that need to map to a single XML schema. However, the schema compiler or the schema generator cannot be used to provide such type of mapping as they are used for one-to-one mapping. In such cases, you need to specify the mappings in Java classes by using JAXB 2.2 annotations. The JAXB 2.2 specification is restricted in the sense that you cannot use JAXB 2.2 annotations for mapping a Java class to XML instances of all the XML schema types.

Figure 19.5 shows the architecture of JAXB implementation:

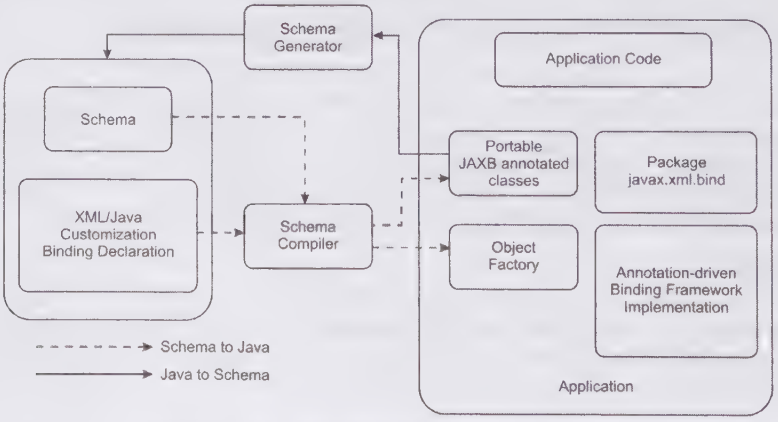


Figure 19.5: Displaying the JAXB Architecture

Figure 19.5 shows the following three main components of the JAXB architecture:

- **The schema compiler**—Binds a source schema to a collection of schema-based program elements by using the JAXB binding language.

- **The schema generator**—Performs the reverse operation to that of the schema compiler. It maps a collection of schema-based program elements to a source schema by using Java annotations.
- **The binding runtime framework**—Performs unmarshal and marshal operations. During unmarshalling, this framework accesses and validates XML content using schema-based program elements and converts the XML content into Java objects. During marshalling, this framework creates an XML document from the Java objects.

  The sub-specifications of the JAXB specification are as follows:
  - Mapping annotations
  - Binding runtime framework
  - Validation
  - Marshal event callbacks
  - Partial binding
  - Binary data encoding
  - JAXB binding language
  - Probability

Let's briefly discuss each of these next.

Mapping Annotations

The JAXB 2.2 mapping annotations are used to customize standard Java/XML binding. The schema generator accepts Java classes and mapping annotations to define customized mapping. In case there is no mapping annotation present in the Java class, the schema generator applies the default Java/XML binding.

The following code snippet uses a mapping annotation to map a Java bean property to an XML schema element:

```
public class PO
{
   ....
   @XmlElement(name="POnum") } )
   String getPONumber();
   void setPONumber();
};
```

When you need to bind an XML element to a Java bean property in the reverse direction, the schema compiler generates the previous mapping annotations in Java code.

JAXB 2.2provides the following standards on mapping annotations:

- You need to create corresponding type mappings in parallel to creation of Java classes
- You need to run the schema generator on Java classes to determine type mappings

It's easy to create Web services from a Java class as mapping annotations allow you to specify type mappings and serialization in Java classes that implement the Web services.

Binding Runtime Framework

The binding runtime framework invokes a Web service deployed in a JWS-complaint application. The steps to perform this operation are as follows:

- A SOAP message is received by the JWS compliant application at a deployed endpoint.
- Depending on the configuration of SOAP binding for the endpoint, the JAX-WS implementation creates the message context, which contains the SAAJ representation of the SOAP message. In this case, JAXB can be used for Message Transmission Optimization Mechanism/XML-binary Optimized Packaging (MTOM)/XOP processing, which creates the SAAJ representation of the SOAP message.
- The JAX-WS implementation calls the request handlers. The application uses the SAAJ interface to retrieve the request parameters of the SOAP message.
- After the request handlers complete the processing, the JAXB runtime binding framework unmarshals the contents of the SOAP message into a request bean. The JAX-WS implementation defines the request bean

depending on the parameter and return type packaging of deployment. The JAXB implementation binds the request bean to the SOAP message request. The terms marshal and unmarshal also mean serialize and deserialize, respectively.

- The request bean consists of Java objects that are passed as parameters to the Java method that invokes the Web service.
- The JAX-WS implementation uses the Java object that has been returned by the Java method to construct the instance of the response bean.
- Next, the JAXB runtime marshals the response bean.
- The JAX-WS modifies the message context to provide the SAAJ representation of the response, which is rendered as the SAAJ interface.
- The JAX-WS implementation calls the response handlers.
- The JAX-WS implementation delegates control to JAXB runtime to process the MTOM of the response message. You learn more about MTOM later in the chapter.
- The JAX-WS implementation transmits the response message to the transport protocol.

Figure 19.6 summarizes this process of invoking a Web service deployed as a JWS-complaint application:

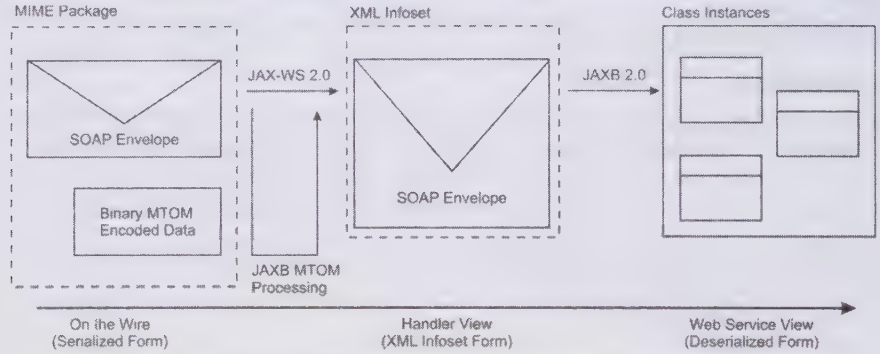


Figure 19.6: Displaying Java Web Service Invocation

In Figure 19.6, a SOAP request comes from the network wire in the form of a MIME package. The JAX-WS implementation converts the serialized SOAP request message into an XML infoset representation. The JAXB implementation unmarshals the infoset representation into JAXB annotated Java elements. The Java method that implements the Web service accepts these elements to invoke the Web service.

Implementing Validation

To successfully implement Web services, you need to handle invalid XML instances. An XML instance is invalid if it is not valid according to the WSDL/XML schema that defines the Web service. JAXB, by default, supports unmarshalling of invalid XML instances. This is done through the flexible unmarshalling mode that ignores some invalid conditions, such as out-of-order elements found in XML instances.

To unmarshal only valid XML instances, turn on the JAXB validation by invoking the `Unmarshaller.setSchema()` method with the `javax.xml.Validation.Schema` instance as a parameter. After the JAXB validation is turned on, by default, unchecked exception is thrown on the first error in an XML instance. You can define your own event handler by setting a `javax.xml.bind.ValidationEventHandler` instance with the `Unmarshaller.setEventHandler()` method to handle errors. Use the `javax.xml.bind.util.ValidationEventCollector` instance inside an event handler to collect all the errors that have occurred during unmarshalling.

Exploring Marshal Event Callbacks

Marshalling is the process of changing instances of JAXB annotated classes to XML infoset representations. This conversion is based on the changed JAXB standard Java/XML binding. Unmarshalling is a process of changing XML infoset representations to content objects that are represented in the form of XML tree structures. The

content objects are either JAXB schema-derived program elements or existing program elements mapped to the XML schema generated by the schema generator.

Callbacks allow application-specific processing during serialization. You can invoke callbacks before or after the serialization.

Exploring Partial Binding

The `javax.xml.bind.Binder` class binds a part of an XML document. For example, you can use this class for creating the JAXB binding of a SOAP header without processing the message body of the SOAP message. The following code snippet shows an example of partial binding of an XML document:

```
JAXBContext jaxbContext= JAXBContext.newInstance(…);
org.w3c.dom.Element persistElt = ... // get from SOAP header
Binder<org.w3c.dom.Node> myBinderDemo = jaxbContext.createBinder();
PersistedValueClass persistVC = (PersistedValueClass)
myBinderDemo.unmarshal(persistElt);
persistVC.setPersisted(true);
myBinderDemo.updateXML(persistVC); // updates the XML infoset
```

The preceding code snippet can be used to navigate and update the Document Object Model (DOM) interface to manipulate the SOAP header of the XML infoset representation. You do not need to work with the DOM interface if you manipulate SOAP header inside handlers.

Exploring Binary Data Encoding

Binary data encoding optimizes the transmission of binary data in SOAP messages. The optimization is achieved by encoding binary data (such as an image), extracting it from the SOAP envelope, attaching its compressed form to a MIME package, and holding references to encoded parts in the SOAP envelope. In addition, JAXB unpackages binary data before unmarshalling and packages it after marshalling. JAXB supports two binary data encoding types: MTOM/XOP and WS-I Attachments Profile Version (WSIAP).

MTOM is a W3C standard to accept content from an XML infoset, compress it, package it as a MIME attachment, and finally replace it with a reference in the infoset. The encoding used during packaging is XML-binary Optimized Packaging (XOP).

Exploring JAXB Binding Language

The binding language is used to make Java types compatible with XML types, so that Java types can be accepted as parameters and return types by Web SEIs. The JAXB binding language annotates XML instances to customize Java representation of XML schema. In other words, the JAXB binding language fine tunes the structure of Java elements generated by the schema compiler. The annotations in the binding language are generated by the schema compiler by using the source schema and binding declarations. Therefore, binding declarations indirectly control the serialization of Web services.

Mapping annotations are provided in a Java class; however, binding language customizations (or binding language declarations) can be given in the following three ways:

- With a global scope
- With a component scope
- As an XML schema or a separate configuration file

Let's explore these three ways to map the annotations next.

Binding Declaration with a Global Scope

The following code snippet shows a binding declaration with a global scope:

```
<jaxb:globalBindings>
 <jaxb:javaType name="long" xmlType="xs:date"
        parseMethod="pkg.DatatypeConverter.parseDate"
        printMethod="pkg.DatatypeConverter.printDate"/>
 </jaxb:javaType>
</jaxb:globalBindings>
```

The binding declaration in the preceding code snippet maps the XML instances of the `xs:datetype` to a Java long type. By default, XML instances of the `xs:datetype` are mapped to `javax.xml.datatype.XMLGregorianCalendar class`. The `parseMethod` and `printMethod` attributes denote the location of the methods used to unmarshal and marshal the type mapping to the JAXB implementation.

Binding Declaration with a Component Scope

The binding declaration may have a component scope if the `<jaxb:name>`binding tag is nested inside the `<xs:appinfo>` element. The following code snippet shows a binding declaration with a component scope:

```
<xs:complexType name="ComplexPO">
  <xs:sequence>
        <xs:element name="PO" type="javector:PurchaseOrder"
        maxOccurs="unbounded"/>
                <xs:annotation><xs:appinfo>
                        <jaxb:dom/>
                </xs:appinfo></xs:annotation>
        </xs:element>
  </xs:sequence>
</xs:complexType>
```

In the preceding code snippet, the PO element is mapped to the `org.w3c.dom.Element` instance of a DOM representation. When you are mapping an XML element to a DOM representation, you can specify the `javector:PurchaseOrder` custom serializer to serialize the DOM representation.

Binding Declaration as an External Configuration File

The external binding file uses XPath expressions to refer to specific schemas associated with the XML elements. The following code snippet shows an external binding file:

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0">
  <jaxb:bindings schemaLocation=".../mySchema.xsd">
        <jaxb:bindings node="//xs:complexType[@name='ComplexPO']">
                <jaxb:bindings node=".//xs:element[@name='PO']">
                        <jaxb:dom/>
                </jaxb:bindings>
        </jaxb:bindings>
  </jaxb:bindings>
</jaxb:bindings>
```

The use of external binding file ensures that you do not require XML-based execution process to implement the WSDL of a Web service. However, including JAXB implementation-specific XML in the WSDL document makes the document insecure.

Explaining Portability

JAXB mapping annotations help in achieving JAXB portability, which simply means that a runtime marshaller of any JAXB implementation can serialize a JAXB annotated class to an instance of the XML schema defining a Web service, or deserialize a schema instance to the instance of the JAXB annotated class.

In other words, JAXB portability allows you to deploy a Web service, implemented through a Java class with JAXB annotations, on any JAXB platform, such as JBoss or GlassFish. You do not need to recompile the schema or change the Web service to use vendor-specific implementation classes.

Exploring the WSEE 1.3 Specification

The WSEE 1.3 specification deals with the service architecture, packaging, and deployment of Web services. The objective of this specification is to make Web services portable on different Java EE application server implementations. The WSEE specification solves the problems related to the deployment of Web services as observed in case of J2EE 1.4. These problems can be solved through the use of annotations and easy packaging architecture. The WSEE specification provides the following sub-specifications:

- Port component
- Servlet endpoints
- Enterprise Java Beans (EJB)endpoints
- Simple packaging
- Handler programming model

Let's look at each feature in detail.

Port Component

The WSEE specification defines a port component as a component that is packaged and deployed on the Web container to implement a Web service. In J2EE 1.4, the WSEE 1.0 specification includes some other artifacts that are required to be deployed along with the port component, such as Service Endpoint Interface (SEI), the Web services Deployment Descriptor (`webservices.xml`), and the JAX-RPC mapping Deployment Descriptor. In Java EE 5, the WSEE 1.2 specification makes certain artifacts, such as the WSDL document, SEI, and webservices.xml, optional. That is, if you define the `webservices.xml` Deployment Descriptor, it overrides the deployment information specified in annotations.

Servlet Endpoints

The WSEE 1.3 specification uses Plain Old Java Objects (POJO) to implement a Web service if it is defined according to the requirements mentioned in the WS-Metadata for a Service Implementation Bean (SIB). This POJO is also known as a Servlet endpoint.

EJB Endpoints

The WSEE 1.3 specification allows a stateless session bean to implement a Web service deployed in an EJB container. This stateless session bean is also known as an EJB endpoint. You also need to create a service implementation bean from the EJB endpoint to execute the Web service functionality.

Simple Packaging

The WSEE 1.3 specification eliminates the need for Deployment Descriptors in simple cases. Even those used in complex situations are simpler when compared to the ones in J2EE 1.4.

Handler Programming Model

WS-Metadata defines many handler annotations. The WSEE 1.3 specification defines the programming and runtime behavior of these handler annotations. For example, the @HandlerChain annotation binds a chain of handlers with a port component. The WSEE 1.3 specification describes how to define a structure for the chain of handlers in the Web services Deployment Descriptor, `webservices.xml`.

Exploring the WS-Metadata 2.2 Specification

The JAX-WS and JAXB specifications cover most of the sub-specifications for the invocation and serialization of JWSs. The WS-Metadata specification provides standards for deploying the JWSs. As per the WS-Metadata 2.2 specification, you can now use annotations to deploy Java classes as Web services. The WS-Metadata 2.2 API provides annotations that are used to develop and deploy Web services on the Java SE 6 and Java EE 6 platforms. You can use the `Start From WSDL` Java development mode to develop and deploy JWSs by using the annotations feature.

For example, let's assume that you need to deploy the createPurchaseOrder() method of the PurchaseOrder Java class as a Web service operation, ns:createPO. This Web service operation includes the ns:PurchaseOrder element based on the given XML schema. You also want to publish the Web service as a WSDL interface for allowing users to access it.

If you want to include WS-Metadata annotations in the PurchaseOrder class, you should have access to its code and access rights to recompile and redeploy the application with the annotations. After you have redeployed the application with metadata annotations, you need to map the PurchaseOrder program elements to the ns:PurchaseOrder element with JAXB annotations. Assume that you create a customized mapping class with

JAXB mapping annotations to perform this mapping. To verify that the customized mapping is valid, you need to run the WSDL/XML schema generator and check the final WSDL document to verify whether it conforms with the given schema.

The `Start From WSDL` development mode can be used to generate the WSDL using wrapper classes. To generate the wrapper classes, you need to run the WSDL/XML schema compiler on the given WSDL standard, retrieve the required Java classes, and convert them into wrapper classes. This entire process invokes the PurchaseOrder class.

The sub-specifications of the WS-Metadata specification are:

- WSDL mapping annotations
- SOAP binding annotations
- Handler annotations
- Service implementation bean
- Start From WSDL and Java
- Automatic deployment

Let's look at these in detail.

WSDL Mapping Annotations

WSDL mapping annotations are also known as Web services annotations. These annotations are packaged in the `javax.jws` package. WSDL mapping annotations modify the default WSDL/Java mapping. For example, they are used to assign the WSDL operation name to a specific Java method. Annotation type declarations are contained in the `javax.jws` package, which need to be imported.

The `javax.jws package` defines the enumerations and annotations. The data types of the `javax.jws` package are described in Table 19.3:

Table 19.3: Data Types of the javax.jws Package

Name	Type	Description
`WebParam.Mode`	Enum	Specifies the flow control of the parameters
`HandlerChain`	Annotation	Links a Web service with an external handler chain
`Oneway`	Annotation	Indicates that the given @WebMethod annotation does not have any output messages but only input message
`WebMethod`	Annotation	Customizes a method that is exhibited as a Web service operation
`WebParam`	Annotation	Specifies the mapping between an individual parameter and a Web service message part along with an XML element
`WebResult`	Annotation	Customizes the mapping of parameter's return value to a WSDL part and an XML element
`WebService`	Annotation	Indicates that a Web service is implemented by a Java class or an interface used for implementing Web service is defined by the Java interface

SOAP Binding Annotations

WS-Metadata 2.2 SOAP binding annotations are present in the `javax.jws.soap` package. These annotations modify the SOAP binding style when performing SOAP mapping. You learned earlier that JAX-WS uses the document/literal wrapped value for the binding style by default.

Handler Annotations

Some WS-Metadata 2.2 annotations are used to deploy handlers. The `javax.jws.HandlerChain` annotation associates a Web service with a chain of handlers defined in an external file, which you can reference by using the `@HandlerChain.file` annotation. Note that JAX-WS-compliant endpoints cannot use the

`javax.jws.soap.SOAPMessageHandlers` annotation to deploy SOAP handlers as this annotation has been deprecated.

Service Implementation Bean

WS-Metadata specifies some conditions for a Java class to be deployed as a Web service. The Java classes that satisfy these conditions are known as Service Implementation Beans (SIBs). WS-Metadata annotations prevent you from container-specific deployment of Web services.J2EE 1.4 complies with the JAX-RPC specification, a previous version of JAX-WS 2.2, which requires SEIs to inherit the `java.rmi.Remote` class for implementing Java classes as Web services. However, as per the WS-Metadata 2.2 specification, both POJOs and EJBs acting as SIBs may be deployed as Web services.

Start From WSDL and Java

The `Start From WSDL and Java` development mode uses some WS-Metadata annotations to map the elements of a Java class or interface to the elements of a WSDL document. For example, the `@WebMethod.operationName` annotation associates a Java method to the `wsdl:operation` element. The demerits of using the `Start From WSDL and the Java` development mode have already been discussed earlier in the chapter. To avoid those demerits, you can use a well-implemented tool, such as Castor, which identifies nonconformance errors when a Web service implementation deviates from a standard WSDL contract.

Automatic Deployment

The WS-Metadata specification allows a Web service to be deployed by using the drag and drop option. Runtime deployment of Web services only depends on annotations used in Java class and there is no need for a vendor-specific deployment tool. GlassFish V3, a Java EE 6 implementation provides the drag and drop deployment model for deploying Web services.

Describing the SAAJ 1.3 Specification

The SAAJ specification lays down the standards for transmitting and processing SOAP messages in compliance with the JAX-WS handlers and JAXR implementations. You can write SOAP messaging applications directly by using this specification. The SAAJ API defined by the SAAJ specification, works on SOAP 1.1, 1.2, and SOAP with attachments specifications. The SAAJ 1.3 API provides the `javax.xml.soap` package that provides classes to create and send request-response messages.

Working with SAAJ and DOM APIs

The interfaces and classes of the SAAJ API extend the corresponding interfaces and classes present in the `org.w3c.dom` package. The `Node` interface of the SAAJ API inherits the `org.w3c.dom.Node` interface of the DOM API. Similarly, the `SOAPElement` interface of the SAAJ API inherits both the `Node` and `org.w3c.dom.Element` interfaces of the DOM API. In addition, the `org.w3c.dom.Document` interface is also implemented by the `SOAPPart` class.

As SAAJ nodes implement the DOM `Node` and `Element` interfaces, you can manipulate contents of a message by using any of the following:

- Only the DOM API
- Only the SAAJ API
- The SAAJ API first, and then using the DOM API
- The DOM API first, and then using the SAAJ API

The first three ways are easy to implement, as after creating a message, you can use either the DOM or the SAAJ API to manipulate the message content. However, when you use the DOM API first, retrieving the references to objects using the DOM API within the source tree of the message is not possible.

Let's look at a few examples to use the SAAJ API. To be precise, we perform the following actions:

- Creating a simple SOAP message
- Accessing the different message parts of the SOAP message

- Adding content to the SOAP Body object
- Sending a message
- Retrieving the content of a message
- Adding content to the header element
- Creating and adding attachments
- Retrieving Attachments

Let's explore these in detail.

Creating a Simple SOAP Message

You can create an instance of a message by invoking the `newInstance()` static method of the `MessageFactory` class. As `MessageFactory` is an abstract class, the SAAJ API uses the default implementation class of the `MessageFactory` class. The following code snippet shows how to create an instance of a message:

```
MessageFactory mfactory = MessageFactory.newInstance();
SOAPMessage msg = mfactory.createMessage();
```

The previous code snippet creates an instance of the `MessageFactory` class for SOAP 1.1 messages. The SOAP message contains the `SOAPPart` object, which further contains the `SOAPEnvelope` object in turn containing empty `SOAPHeader` and `SOAPBody` objects.

In order to create an instance of the `MessageFactory` class for SOAP 1.2 messages, pass the SOAP_1_2_PROTOCOL constant as an argument to the `newInstance()` method, as shown in the following code snippet:

```
MessageFactory mfactory =
MessageFactory.newInstance(SOAPConstants.SOAP_1_2_PROTOCOL);
```

In order to create an instance of the `MessageFactory` class that enables you to create either SOAP 1.1 or SOAP 1.2 messages, call the `newInstance()` method, as shown in the following code snippet:

```
MessageFactory mfactory =
MessageFactory.newInstance(SOAPConstants.DYNAMIC_SOAP_PROTOCOL);
```

Accessing the different Message Parts of a SOAP Message

The SAAJ API provides different methods to access the parts of a SOAP message. Table 19.4 lists the methods to access the different message parts of a SOAP message:

Table 19.4: Methods to Access Message Parts

Message Part	Syntax	
SOAPPart	`SOAPPart soapPrt = msg.getSOAPPart();`	
SOAPEnvelope	`SOAPEnvelope spEnv= soapPrt.getEnvelope();`	
SOAPHeader and SOAPBody	`SOAPHeader soapheadr = spEnv.getHeader();` `SOAPBody soapbdy = spEnv.getBody();`	`SOAPHeader soapheadr = msg.getSOAPHeader();` `SOAPBody soapbdy = msg.getSOAPBody();`

Adding Content to the SOAPBody Object

To add content to the `SOAPBody` object, you need to first create one or more `SOAPBodyElement` objects, and then add sub elements to these `SOAPBodyElement` objects by invoking the `addChildElement()` method. Finally, add content by invoking the `addTextNode()` method. You should create an associated `javax.xml.namepace.QName` object for each new `SOAPBodyElement` element that uniquely identifies the element. The `QName` object consists of a fully qualified namespace URI, local part, and a namespace prefix.

The following code snippet associates a QName object to a SOAPBodyElement object:

```
SOAPBody soapbdy = msg.getSOAPBody();
QName soapbdyElemName = new QName("http://wombat.ztrade.com",
"GetLastTradePrice", "n");
SOAPBodyElement soapbdyElem = soapbdy.addBodyElement(soapbdyElemName);
```

In the previous code snippet, as the `soapbdyElem` element contains no content, we need to add the stock symbol SUNW as a child element of the `soapbdyElem` element, as shown in the following code snippet:

```
QName qname = new QName("symb");
SOAPElement symb = soapbdyElem.addChildElement(qname);
symb.addTextNode("SUNW");
```

In the preceding code snippet, the symb local name is specified while creating the instance of the QName class, named qname. Further, the qname instance inherits namespace prefix and URI from the parent element.

As you know that SOAP part can contain only XML content, SAAJ API creates appropriate XML elements automatically on invocation of methods, such as `addBodyElement()`, `addChildElement()`, and `addTextNode()`.

NOTE

You cannot invoke the addTextNode() method on a SOAPHeader or SOAPBody object, as they contain only elements.

Let's now explore how a SOAP-based XML document is generated by using the SAAJ API. The following code snippet shows the SAAJ API code for generating the XML document:

```
SOAPMessage msg = mFactory.createMessage();
SOAPHeader soapheadr = msg.getSOAPHeader();
SOAPBody soapbdy = msg.getSOAPBody();
QName soapbdyElemName = new QName("http://wombat.ztrade.com",
"GetLastTradePrice", "n");
SOAPBodyElement soapbdyElem = soapbdy.addBodyElement(soapbdyElemName);
QName qname = new QName("symb");
SOAPElement symb = soapbdyElem.addChildElement(qname);
symb.addTextNode("SUNW");
```

After executing the code provided in the preceding code snippet, the XML document is created as shown in the following code snippet:

```
<SOAP-ENV:env
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:body>
        <n:GetLastTradePricexmlns:n="http://wombat.ztrade.com">
              <symb>SUNW</symb>
        </n:GetLastTradePrice>
  </SOAP-ENV:body>
</SOAP-ENV:env>
```

Sending a Message

All SOAP messages are transmitted over a connection between a sender and destination endpoint. The SAAJ API provides the `SOAPConnection` class to facilitate the sending and receiving of SOAP messages. Two types of messages are sent using the SAAJ API, request and response. You can send request messages by making a call to the `SOAPConnection.call()` method. The following code snippet shows how to send a request message:

```
SOAPConnectionFactory soapConnectFactory =
  SOAPConnectionFactory.newInstance();
SOAPConnection con = soapConnectFactory.createConnection();
. . .// code to create a request message and give it content
java.net.URL endpointURL =
  new URL("http://wombat.ztrade.com/quotes");
SOAPMessage responsemsg = con.call(msg, endpointURL);
con.close();
```

The `call()` method sends the request message `msg` to the specified endpoint `endpointURL`. The first argument to the `call()` method specifies the request message to be sent, and the second argument to this method specifies the destination of the request message.

Retrieving the Content of a Message

You need to invoke the `getValue()` method on the SOAPBodyElement object to retrieve the content of a SOAP message, as shown in the following code snippet:

```
SOAPBody soapBdy = responsemsg.getSOAPBody();
java.util.Iterator it =soapBdy.getChildElements(soapbdyElemName);
SOAPBodyElement bdyElem =(SOAPBodyElement)it.next();
String lstPrice = bdyElem.getValue();
System.out.print("The last price for SUNW is ");
System.out.println(lstPrice);
```

In the preceding code snippet, after retrieving the instance of the `SOAPBody` object, the `getChildElements()` method is invoked to get an iterator, `it`. The iterator object `it` contains child elements associated with the `soapbdyElemName` object. As we inserted only one `SOAPBodyElement` element earlier, the `next()` method call on the `Iterator` object returns a Java object that is further typecasted to the `SOAPBodyElement` type. The call to the `getValue()` method on the `bdyElem` instance returns the last price of the `SUNW` stock symbol.

Adding Content to the Header Element

You can add content to a header element of a SOAP message by creating the instance of the `SOAPHeader` class. The following code snippet adds the required content to the `Claim` header:

```
SOAPHeader soapheadr = msg.getSOAPHeader();
QName soapheadrElemName = new QName(
  "http://ws-i.org/schemas/conformanceClaim/","Claim", "wsi");
SOAPHeaderElement soapheadrElem =soapheadr.addHeaderElement(soapheadrElemName);
soapheadrElem.addAttribute(new QName("conformsTo"),
  "http://ws-i.org/profiles/basic/1.1/");
```

The `soapheadrElem` object contains an attribute that specifies information about WS-I conformance claim header. The `addHeaderElement()` method first creates a SOAPHeaderElement instance `soapheadrElem` and adds it to the `soapheadr` header. The header elements can contain only text strings. The following code snippet shows how to add a text string to the header element:

```
soapheadrElem.addTextNode("order");
```

Creating and Adding Attachments

To create and add attachments to a SOAP message, you first need to create an attachment by creating an `AttachmentPart` object, which can be created by invoking the `createAttachmentPart()` method on an instance of `SOAPMessage`, as shown in the following code snippet:

```
AttachmentPart attachmntPart = msg.createAttachmentPart();
```

The preceding code snippet creates an attachment having no content. You can add content to the empty attachment by using the `setContent()` method of the `AttachmentPart` object. The `setContent()` method accepts two parameters, a String object that represents the content and another `String` object that represents the MIME content type. The Java object may be String, stream, a `javax.xml.transform.Source` or `javax.activation.DataHandler` object. After adding content to the attachment, you need to add the attachment to a message. The following code snippet shows the code for adding a text-based attachment to a message:

```
String str = "Update address for Kogent Solutions Inc., " +
  "to  4435/7, ANSARI ROAD, DARYA GANJ, NEW DELHI 110002";
attachmntPart.setContent(str, "text/plain");
attachmntPart.setContentId("update_addr");
msg.addAttachmentPart(attachmntPart);
```

The preceding code snippet uses the `String` object in the first parameter as a Java `Object` and `text/plain` as the second parameter in the `setContent()` method. The preceding code snippet also adds the `ContentId` header with a value `update_addr`. Finally, the attachment is added to the `msg` instance.

Apart from adding plain text, you can also add images to the attachments by calling the `createAttachmentPart()` method with a `DataHandler` object as parameter. Again, after creating the

attachment with an image, you need to add it to a message. The following code snippet shows how to add an image as an attachment to a message:

```
java.net.URL imageurl = new URL(".../img.jpg");
DataHandler dataHandlr = new DataHandler(imageurl);
AttachmentPart attachmntPart=msg.createAttachmentPart(dataHandlr);
attachmntPart.setContentId("image");
msg.addAttachmentPart(attachmntPart);
```

Retrieving Attachments

To manipulate the content of an attachment, you need to access the attachment. Two overloaded versions of the `getAttachments()` method are provided by the `SOAPMessage class` to retrieve the `AttachmentPart` objects. The first version of the `getAttachments()` method accepts no argument and returns a `java.util.Iterator` object for all the `AttachmentPart` objects in a message. The second version of the `getAttachments()` method accepts the `MimeHeaders` object as an argument and returns the `AttachmentPart` objects corresponding to headers present in the `MimeHeaders` object. The following code snippet retrieves and prints the content of all the attachments in a SOAP message:

```
java.util.Iterator it = msg.getAttachments();
while (it.hasNext())
{
  AttachmentPart attachmntPart= (AttachmentPart)it.next();
  String attachmntId = attachmntPart.getContentId();
  String attachmntType = attachmntPart.getContentType();

  System.out.print("Attachment " + attachmntId +
  " has content type " + attachmntType);
  if (attachmntType.equals("text/plain"))
  {
        Object attachmntContent = attachmntPart.getContent();
        System.out.println("Attachment has content:\n" + attachmntContent);
  }
}
```

Describing the JAXR Specification

JAXR helps in accessing standard business registries over the Internet. Business registries normally consist of listings of businesses and the products or services the businesses offer. Therefore, business registeries are often described as electronic yellow pages.

In the Web service architecture, a registry plays a key role as it is used to publish, search, and use Web services. Business registries allow businesses to collaborate with each other dynamically in a loosely coupled way.This is one reason why business registries are gaining popularity as important components of Web services. Consequently, the need for JAXR, which enables enterprises to access standard business registries, is also growing. JAXR provides various APIs to access different XML registries. It also helps in describing the content and the metadata in an XML registry. In addition, it helps in writing portable registry client programs.

Let's now explore the JAXR architecture in detail.

JAXR Architecture

The JAXR architecture can be broadly divided into the following parts:

- **JAXRclient**—A client program accessing registry through JAXR provider
- **JAXRprovider**—A JAXR implementation providing access to some registry provider

Figure 19.7 shows the JAXR architecture:

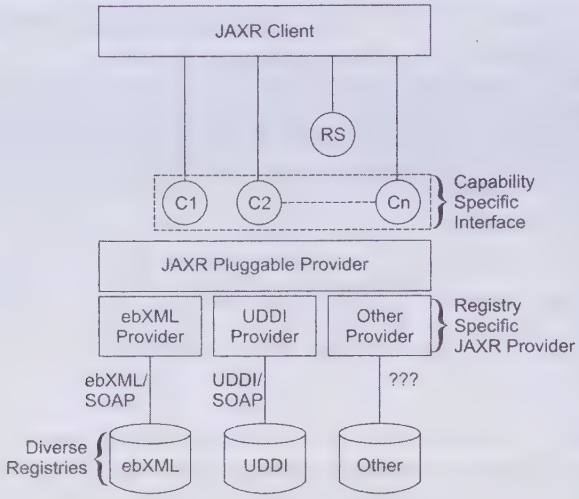


Figure 19.7: Showing the JAXR Architecture

JAXR Client

Any Java application or enterprise component can become a JAXR client by using the JAXR API. A JAXR client can contain the following interfaces:

- **Connection interface** – Represents a client session along with a registry provider by using the JAXR provider. Creating a connection object by using this interface with the registry provider is necessary to use the services of that registry. The `ConnectionFactory` interface is used to create the connection.
- **RegistryService interface** – Provides various getter methods used to retrieve `capability-specific` interfaces.
- **Other capability-specific interfaces** – Helps to perform activities, such as life-cycle management and query management.

JAXR Provider

A JAXR provider represents an implementation of the JAXR API. The JAXR provider helps a JAXR client to access a registry. A JAXR provider needs to implement the following two packages:

- **javax.xml.registry** – Contains interfaces and classes used to access a specific registry. It has four main interfaces:
- **Connection** – Represents the connection that exists between a client and a JAXR provider.
- **RegistryService** – Enables a client to access the registery.
- **BusinessQueryManager** – Allows clients to find information in a registry according to the `javax.xml.registry.infomodel` interfaces.
- **BusinessLifeCycleManager** – Allows clients to manipulate the information in a registry.
- **javax.xml.registry.infomodel** – Provides interfaces that define the types of objects residing in a registry, and how these objects are related to each other. `RegistryObject` is the key interface of this package. The sub-interfaces of the `RegistryObject` interface are `Organization`, `Service`, and `ServiceBinding`.

  JAXR providers can be of differnet types, such as:

  - JAXR Pluggable Provider
  - Registry-Specific JAXR Provider
  - JAXRBridge Provider
  - Registry Provider

Let's describe these in detail next.

JAXR Pluggable Provider

JAXR pluggable providers implement the JAXR API features that are not dependent on any particular type of registry. When the access to the requested registry is being provided by multiple JAXR providers, the use of JAXR pluggable provider provide abstraction to the client. It also provides the `ConnectionFactory` implementation, which can be used to create JAXR connections by using different registory-specific JAXR providers.

Registry-Specific JAXR Provider

Registry-specific JAXR providers implement the JAXR API in a manner that is specific to a registry. In case of registry-specific implementation, a client request is transformed into equivalent requests based on the specifications of the target registry and forwarded to the registry provider. Finally, registry-specific responses are also converted into equivalent JAXR response by the registry-specific JAXR provider.

JAXRBridge Provider

If a registry-specific JAXR provider acts as a bridge to the existing registry providers, it is known as a JAXR bridge provider. Consequently, JAXRbridge providers are not specific to a particular registry type; instead, they may be specific to a group of registries (such as OASIS, ebXML, or UDDI).

Registry Provider

Registry providers are implementations of various registry specifications, such as ebXML or UDDI.

Exploring the StAX 1.0 Specification

The StAX 1.0 specification is a joint effort of BEA Electronics and Sun Microsystems. The StAX specification provides various APIs to control the parsing of XML documents. The StAX APIs address the limitations of other parsing APIs, such as Simple API for XML (SAX) and DOM. Following are some uses of the StAX APIs:

- Help in unmarshalling and marshalling and parallel document processing of an XML document
- Parse simple predictable structures, graph representations, and WSDL, while processing a SOAP message
- Navigate a DOM tree in the form of stream of events
- Help parse particular XML vocabularies

Let's discuss the StAX APIs and factory classes in detail next.

StAX APIs

The StAX APIs allow you to perform iterative and event-based processing of XML documents. The XML documents are considered as filtered series of events. The StAX API comprises of two APIs, cursor API and iterator API.

The Cursor API

The StAX `cursor` API provides a cursor to navigate through an XML document. This cursor can point to an element of the XML document and can move in the forward direction only. Key interfaces in the cursor API are `XMLStreamReader` and `XMLStreamWriter`.

The `XMLStreamReader` interface provides methods to retrieve information, such as document encoding, element names, attributes, text nodes, and processing instructions. The following code snippet shows the syntax of some of the methods of the `XMLStreamReader` interface:

```
public interface XMLStreamReader
{
  public int next() throws XMLStreamException;
  public boolean hasNext() throws XMLStreamException;
  public String getText();
  public String getLocalName();
  public String getNamespaceURI();
  ...
}
```

You can call the `getText()` and `getName()` methods of the `XMLStreamReader` interface to retrieve the data at the current cursor location.

The `XMLStreamWriter` interface provides methods to handle event types, such as `StartElement` and `EndElement`. The following code snippet shows the syntax of some of the methods of the `XMLStreamWriter` interface:

```
public interface XMLStreamWriter
{
   public void writeStartElement(String localName)
          throws XMLStreamException;
   public void writeEndElement()
          throws XMLStreamException;
   public void writeCharacters(String text)
          throws  XMLStreamException;
   ...
}
```

The Iterator API

The StAX `iterator` API represents the data of an XML document in the form of a set of discrete event objects. When an XML parser reads the source XML document, these events are retrieved by the application in the order in which the parser reads them.

`XMLEvent` is the main interface of the iterator API, which contains the `XMLEventReader` and `XMLEventWriter` interfaces to read and write the iterator events, respectively.

The `XMLEventReader` interface has five methods. The following code snippet shows the syntax of some of the methods of the `XMLEventReader` interface:

```
public interface XMLEventReader extends Iterator
{
   public XMLEvent nextEvent() throws XMLStreamException;
   public boolean hasNext();
   public XMLEvent peek() throws XMLStreamException;
   ...
}
```

In the preceding code snippet, the `XMLEventReader` interface extends the `Iterator` interface.

The following code snippet shows the syntax of some of the methods of the `XMLEventWriter` interface:

```
public interface XMLEventWriter
{
   public void flush() throws XMLStreamException;
   public void close() throws XMLStreamException;
   public void add(XMLEvent e) throws XMLStreamException;
   public void add(Attribute attribute)
          throws XMLStreamException;
   ...
}
```

StAX Factory Classes

You can create XML stream readers, writers, and events by using the StAX factory classes, such as `XMLInputFactory`, `XMLOutputFactory`, and `XMLEventFactory`. You can configure these readers, writers, and events by setting properties of factory classes.

Let's explore these factory classes in detail next.

The XMLInputFactory Class

The `XMLInputFactory` class is an abstract class that creates instances of `XMLStreamReader` interface and then configures the implementation instances of XML stream reader processors. An instance of the `XMLInputFactory` class can be created by invoking the `XMLInputFactory.newInstance()` method. The

newInstance() method finds a particular XMLInputFactory implementation class by performing the following operations:

- Using the javax.xml.stream.XMLInputFactory system property
- Using the lib/xml.stream.properties file in the Java SE JRE directory
- Finding the class name in the META-INF/services/javax.xml.stream.XMLInputFactory files in JAR files of an application
- Using the platform default XMLInputFactory instance

The XMLOutputFactory Class

An instance of the XMLOutputFactory class can be created by invoking the XMLOutputFactory.newInstance() method. The newInstance() method of this class references the javax.xml.stream.XMLOutputFactory system property. The XMLOutputFactory class has only one property, javax.xml.stream.isRepairingNamespaces. This property creates default prefixes and links them with namespace URIs.

The XMLEventFactory Class

You can create an instance of the XMLEventFactory class by invoking its newInstance() method. The newInstance() method of this class references the javax.xml.stream.XMLEventFactory system property. The XMLEventFactory class has no default property.

Section C:
Using the Web Service Specifications

In Section B, you learned about the various Web service specifications used to implement SOA. In this section, let's look at the practical implementation of these Web service specifications to implement SOA with JWSs. Let's start by developing client-side SOA application components by using the JAX-WS 2.2 specification. Next, you learn to implement Java/XML binding and type mapping by using the JAXB 2.2 specification. You also learn to deploy Web services by using the WSEE 1.3 specification. After that, you create a simple messaging application by using the SAAJ 1.3 specification. You use the JAXR 1.0 specification to develop JAXR clients. Finally, you learn to use the StAX 1.0 specification to read and write to XML streams and XML files.

Using the JAX-WS 2.2 Specification

JAX-WS is primarily used to create client-side SOA components that consume (or invoke and use) Web services. JAX-WS is similar to Remote Message Invocation (RMI) in Java as it allows you to make a local method call to invoke a Web service deployed on a remote host. However, RMI requires that a Web service on remote host should be implemented as a Java application. In JAX-WS, service must provide a WSDL interface having a wsdl:portType definition. While creating the client-side SOA components, you need to ensure that proper Java/WSDL mapping has been configured. You can use JAX-WS to map the wsdl:portType element of the WSDL interface to the corresponding Java interface. This Java interface is known as Service Endpoint Interface (SEI), as it is the Java representation of a Web service endpoint.

A Web service endpoint can be a referenceable entity, processor or resource by which you can send or receive Web service messages. JAX-WS dynamically generates an instance of SEI, and you can invoke a Web service by making a call to a method of the implementation class of SEI. This implementation class is also known as the JAX-WS dynamic proxy class.

After you have created a SOA component, you can use the annotations provided by the JAX-WS runtime environment to marshal and unmarshal method calls to SEI.

In this section, let's explore the use of JAX-WS proxies to invoke Web services. You also learn to perform Java/WSDL mapping by using JAX-WS annotations. In addition, you learn to marshal and unmarshal method calls to SEI.

Invoking Web Services by using JAX-WS Proxies

The JAX-WS runtime environment provides the implementation class of SEI, which is an instance of `java.lang.relect.Proxy` class, or the proxy class. To invoke a Web service, the proxy class uses an instance of the `java.lang.reflect.InvocationHandler` class to implement standard WSDL to Java and Java to WSDL mapping. The `InvocationHandler` object changes the passed parameters into SOAP messages, which are sent to the Web service endpoint. Similarly, SOAP response messages are converted back into an instance of the return type of SEI.

Figure 19.8 shows the process of invoking a Web service by using a JAX-WS proxy:

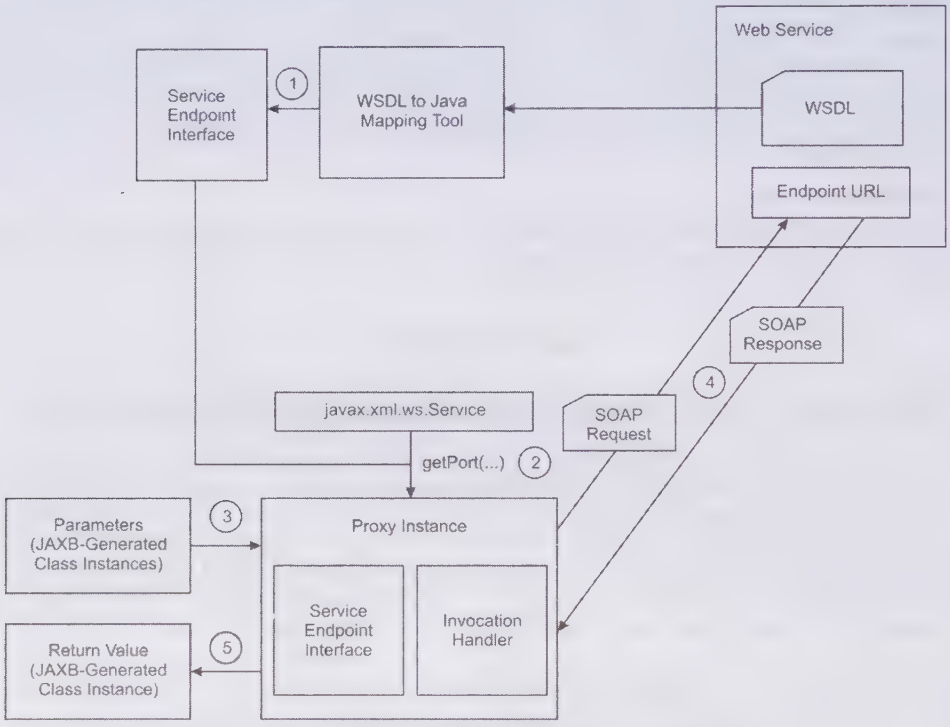


Figure 19.8: Invoking a Web Service Using a Java Proxy

Let's understand each step in Figure 19.8:

- A WSDL to Java mapping tool, such as wsimport tool of the GlassFish server, reads the WSDL and creates an SEI.
- The JAX-WS runtime invokes the appropriate `getPort()` method of the `javax.xml.ws.Service` class to create a proxy instance that implements the SEI.
- The Web service is invoked by a call to a method with different parameters. These parameters are instances of JAXB generated classes of the SEI.
- The dynamic proxy class uses an instance of the `java.lang.reflect.InvocationHandler` class to convert the passed parameters into a SOAP message, which is then sent to the Web service endpoint.
- Finally, the dynamic proxy class converts the SOAP response message into an instance of the return type of SEI.

Implementing JAX-WS WSDL to Java Mapping

To understand the concept of WSDL to Java mapping, let's consider a WSDL, `RequestOrder`. Figure 19.9 illustrates WSDL to Java mapping for the `RequestOrder` WSDL:

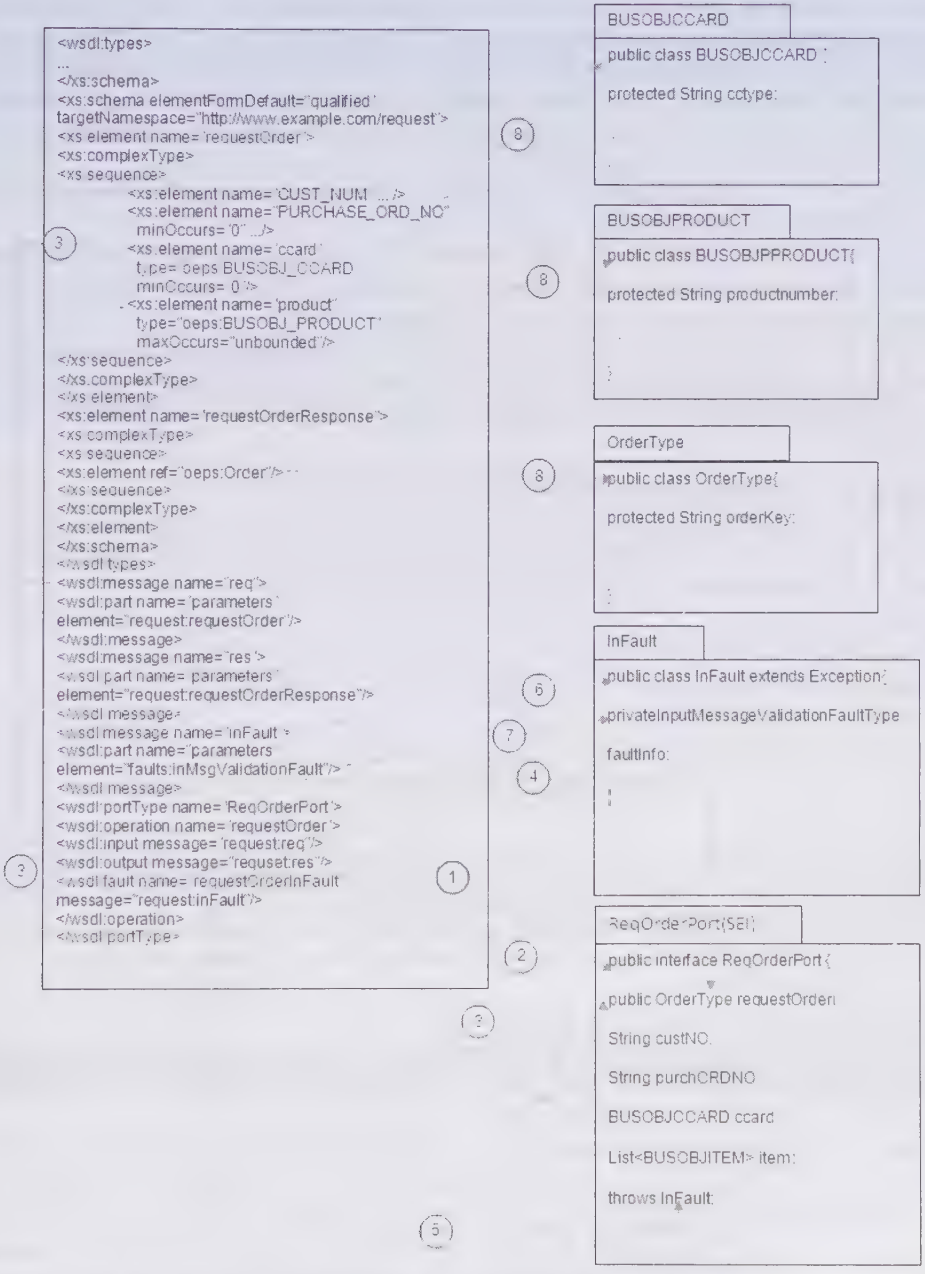


Figure 19.9: Showing the JAX-WS WSDL to Java Mapping in the RequestOrder Web Service

In Figure 19.9, the large box on the left shows the code of the `RequestOrder` WSDL. Remaining five boxes represent the mapping Java classes generated after the WSDL to Java mapping. The mapping is performed as per the following steps:

1. The `wsdl:portType` element of the WSDL with name `ReqOrderPort` is mapped to the SEI with the same name.

2. The `wsdl:operation` element named as `requestOrder`, inside the `wsdl:portType` element is mapped to a method of the SEI, with the same name, `requestOrder`.
3. The `wsdl:inputmessage` is mapped to the parameters of the `requestOrder()` method, such as `custNO`, `purchORDNO`, and `ccard`. The input message `request:req` has a single wrapper element `request:requestOrder` as declared by the `wsdl:part` element. The wrapper element is in turn defined within the `wsdl:types` section.
4. The `wsdl:outputmessage` element is mapped to the response type `OrderType`, which acts as a return type of the `requestOrder()` method.
5. The`wsdl:fault` element is mapped to thrown exception clause of the `requestOrder()` declaration.
6. The `wsdl:message element`, which refers to the `wsdl:fault element`, is mapped to the `InFault` Java class that extends the `java.lang.Exception` class.
7. The `wsdl:fault` element uses `wsdl:messageinFault` that has a single part. This part refers to a global element that is mapped to the `InputMessageValidationFaultType` fault bean.
8. The types defined in the `wsdl:types` element is mapped to Java classes using JAXB XML Schema to Java mapping.

Marshalling and Unmarshalling Method Calls to SEI

The JAX-WS runtime uses annotations to marshal a method call on SEI into a SOAP request message and unmarshal a SOAP response message into an instance of return type of SEI's method. The annotations used for marshalling and unmarshalling are given as follows:

- WebService
- WebMethod
- WebParam
- WebResult
- RequestWrapper
- ResponseWrapper

Let's discuss these in detail.

The WebService Annotation

The WebService annotation is used to annotate a class or an interface. The use of this annotation with an interface declares that this interface defined a Web service interface. When this annotation is used with a class, it declares that the class is the implementation of a Web service. The list of optional elements for the WebService annotation is provided in Table 19.5:

Table 19.5: Elements of the WebService Annotation

Element	Description
Name	Specifies Web service name
targetNamespace	Specifies the namespace for the wsdl:portType element (and associated XML elements)
serviceName	Specifies the service name of the Web service
portName	Specifies the port name of the Web service
wsdlLocation	Specifies the location of the WSDL describing the Web service
endpointInterface	Specifies the complete name of the SEI

The WebMethod Annotation

The WebMethod annotation is used to annotate a method of a Web service. It customizes a method exposed as a Web service operation. The list of elements for the WebMethod annotation is provided in Table 19.6:

Table 19.6: Elements of the WebMethod Annotation	
Element	**Description**
operationName	Specifies the name of wsdl:operation that matches the method annotated by using the WebMethod annotation
action	Specifies the action for the corresponding operation
exclude	Marks a method not to be exposed as a Web service operation

The WebParam Annotation

The WebParam annotation customizes the mapping for an individual parameter of an SEI method to a single WSDL message part or element. The list of elements for the WebParam annotation is specified in Table 19.7:

Table 19.7: Elements of the WebParam Annotation	
Element	**Description**
name	Specifies the parameter name as listed in the WSDL document. In case of RPC binding, it refers to the name of the wsdl:part element that represents the parameter. In case of document binding, it refers to the name of XML element that represents the parameter.
partName	Specifies the name of the wsdl:part element representing the parameter.
targetNamespace	Specifies the namespace for the parameter.
mode	Specifies the direction of the flow control of the parameter (IN, OUT, or INOUT).
header	Specifies whether the parameter is to be carried as part of the header or not. If true, the parameter is retrieved from a message header rather than the message body.

The WebResult Annotation

The WebResult annotation is used to customize the mapping of the return value of an SEI method to a WSDL part or XML element. Table 19.8 lists the elements of the `WebResult` annotation:

Table 19.8: Elements of the WebResult Annotation	
Element	**Description**
name	Specifies the return value name as listed in the WSDL document. In case of RPC binding, it refers to the name of the wsdl:part element that represents the return value. In case of document binding, it refers to the local name of the XML element that represents the return value.
partName	Specifies that name of the wsdl:part element that represents the return value.
targetNamespace	Specifies the XML namespace for the return value.
header	Specifies whether return value is to be carried as part of the header or not. Its value is set to true, if result is to be pulled from header.

The RequestWrapper Annotation

The RequestWrapper annotation is used to annotate methods in an SEI with the request wrapper bean used at runtime.

Table 19.9 lists the elements of the `RequestWrapper` annotation:

Table 19.9: Elements of the RequestWrapper Annotation	
Element	**Description**
localName	Specifies the local name of XML element that represents the request wrapper
className	Specifies the fully qualified name of the Java class implementing the request wrapper

Table 19.9: Elements of the RequestWrapper Annotation	
Element	**Description**
targetNamespace	Specifies the namespace for the XML request wrapper element

The ResponseWrapper Annotation

The ResponseWrapper annotation is used to annotate methods in an SEI with the response wrapper bean used at runtime.

Table 19.10 lists the elements of the ResponseWrapper annotation:

Table 19.10: Elements of the ResponseWrapper Annotation	
Element	**Description**
localName	Specifies the local name of XML element that represents the response wrapper
className	Specifies the fully qualified name of the Java class implementing the response wrapper
targetNamespace	Specifies the namespace for the XML response wrapper element

Invoking a Web Service using a Proxy

You can use the methods of the dynamic proxy class to invoke a Web service by retrieving a proxy instance of the Web service. You can obtain a proxy instance by any of the following three ways:

- Using the @WebServiceRef annotation
- Using a generated service class
- Using a dynamically configured service

The following code snippet shows how to use the `@WebServiceRef` annotation to invoke a Web service:

```
@WebServiceRef(RequestOrderService.class)
public static RequestOrderPort reqport;
```

You can also use the generated service class to invoke a Web service. In the following code snippet, we use the `RequestOrderService` generated service class to invoke the `RequestOrder` Web service:

```
RequestOrderService reqorderservice = new RequestOrderService();
RequestOrderPort reqport = reqorderservice.getRequestOrderPort();
```

The third method to invoke a Web service is by using a dynamically configured service class. You can use the `javax.xml.ws.Service` class to dynamically create an instance of an SEI proxy at runtime, as shown in the following code snippet:

```
URL wsdlURL = new URL("http://"+hostName+":"+portVal+"/chap19-endpoint-endpoint-
  1.0/requestOrder?wsdl");
QName servQName =
new QName("http://www.example.com/req", "RequestOrderService");
QName portQName =
new QName("http://www.example.com/req", "RequestOrderPort");
Service serv = Service.create(wsdlURL, servQName);
RequestOrderPort port =
(RequestOrderPort) serv.getPort(portQName, RequestOrderPort.class);
```

Using the JAXB 2.2 Specification

The Java/XML binding is defined by using JAXB 2.2 annotations in Java classes. Each Java class maps to unique XML schema components depending on the annotations. You can use the JAXB 2.2 specification to bind XML instances with Java classes by:

- Creating Java classes and using the JAXB 2.2 schema generator to generate XML schema from these Java classes
- Using the XML schema to generate Java classes by using the JAXB 2.2 schema compiler

Binding between XML Schema and Java Classes

The JAXB Java/XML binding defines a standard map from Java classes to an XML schema. The JAX-WS API uses this map to produce WSDL from a Java class deployed as a Web service. The JAXB implementation uses a schema complier to generate Java class elements from XML schema components. Each XML schema component consists of complex type elements. These elements are known as Java value classes. Each value class has get/set methods to access the content of the corresponding XML schema component. This implies one-to-one mapping between the elements and attributes of a complex type and the properties of a Java value class.

The JAXB binding language can change external characteristics, such as the names of the properties of Java value classes. For example, you can map the XML element, which occurs many times in an XML document, to a collection, as shown in the following code snippet:

```
<xs:element name="abc" type="Bar" maxOccurs="unbounded"/>
```

Let's consider another example of mapping a complex XML schema component to a single Java property, as shown in the following code snippet:

```
<xs:element name="phone">
  <xs:complexType>
        <xs:attribute name="acode" type="xs:string"/>
        <xs:attribute name="phno" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

To define the binding between the `<xs:element name= "phone">` complex element and the related Java property, you need to write the custom mapping class, which extends the `javax.xml.bind.annotation.adapters.XmlAdapter` class. The built-in XML schema types, such as `xs:string`, are mapped to basic Java primitive types, such as `java.lang.String`.

The schema compiler is used in conjunction with JAXB value classes to produce factory classes and create valid XML instances from a schema. If you use the Java implementation of the DOM API to create valid schema instances, an additional JAXP validator is required to fix the validation errors. If you use the JAXB API, schema compiler tools, such as `XmlBeans`, are required to validate XML instances.

The JAXB implementation uses a schema generator to generate schemas from existing classes by using a standard map. The schema generator searches for Java bean properties in a Java class, and maps them to attributes and elements of XML instances. If the Java class is not a Java bean, the schema generator is unable to generate appropriate XML schemas from the Java class.

Let's now learn how to customize the Java/XML binding.

Customizing JAXB Binding

You can customize JAXB binding in the following two ways:

- Schema to Java
- Java to Schema

Let us discuss each in detail.

Schema to Java

XML-specific constraints cannot customize JAXB classes to create Java-specific program elements, such as a package. For this, you can use custom JAXB binding declarations in an XML schema to customize machine-generated JAXB classes. You can customize XML schemas by:

- Using annotations inside a source XML schema
- Passing an external binding file containing all the binding customizations to a compiler

Java to Schema

In this binding, you need to customize Java program elements by using JAXB annotations so that they can be mapped to specific XML schema. The `javax.xml.bind.annotations` package includes various JAXB annotations to perform Java to schema mapping. Let's discuss each annotation one by one.

@XmlSchema Annotation

The @XmlSchema annotation is used for mapping a package to an XML target namespace. The following code snippet shows the default setting of the `@XmlSchema` annotation:

```
@XmlSchema (
  xmlns = {},
  namespace = "",elementFormDefault= XmlNsForm.UNSET;
  attributeFormDefault = XmlNsForm.UNSET,
)
```

@XmlAccessorType Annotation

The @XmlAccessorType annotation handles default serialization of fields and properties in a Java class. The following code snippet shows the default setting of the `@XmlAccessorType` annotation:

```
@XmlAccessorType (
  value = AccessType.PUBLIC_MEMBER)
```

@XmlAccessorOrder Annotation

The @XmlAccessorOrder annotation handles the default ordering of properties and fields of a Java class that are mapped to XML elements. The following code snippet shows the default setting for the `@XmlAccessorOrder` annotation:

```
@XmlAccessorOrder (
  value = AccessorOrder.UNDEFINED
)
```

@XmlSchemaType Annotation

The @XmlSchemaType annotation maps a Java type to an XML schema built-in type. The following code snippet shows the default setting of the `@XmlSchemaType` annotation:

```
@XmlSchemaType (
  namespace = "http://www.w3.org/2001/XMLSchema",
  type = DEFAULT.class
)
```

@XmlSchemaTypes Annotation

The @XmlSchemaTypes annotation contains multiple `@XmlSchemaType` annotations to be applied on a Java program element. It is used to define the value for the XMLSchemaType element for different types in a package and it has no default settings.

@XmlType Annotation

The @XmlType annotation maps a Java class or Enum type to an XML schema type. You can use the `@XmlType` annotation with an Enum type and a top level class. The following code snippet shows the default setting of the @XmlType annotation:

```
@XmlType (
  name = "##default",
  propOrder = {""},
  namespace = "##default",
  factoryClass = DEFAULT.class,
  factoryMethod = ""
)
```

@XmlRootElement Annotation

The @XmlRootElement annotation is used to map a top level Java class or an Enum type to an XML element. The following code snippet shows the default setting of the `@XmlRootElement` annotation:

```
@XmlRootElement (
  name = "##default",
  namespace = "##default"
)
```

@XmlEnum Annotation

The @XmlEnum annotation maps a Java Enum type to an XML simple type. The following code snippet shows the default setting of the `@XmlEnum` annotation:

```
@XmlEnum ( value = String.class )
```

@XmlEnumValue Annotation

The @XmlEnumValue annotation maps an Enum constant in the Enum type to its XML representation. You can use the `@XmlEnumValue` annotation with an Enum constant. It has no default settings.

@XmlElement Annotation

The @XmlElement annotation maps a JavaBean property to a local element of complex type of an XML schema. The name of the local element is same as that of the JavaBean property. You can use the `@XmlElement` annotation with a JavaBean property, non-static field, and a `@XmlElements` annotation. The following code snippet shows the default setting of the @XmlElement annotation:

```
@XmlElement (
  name = "##default",
  nillable = false,
  namespace = "##default",
  type = DEFAULT.class,
  defaultValue = "\u0000"
)
```

@XmlElements Annotation

The @XmlElements annotation contains numerous `@XmlElement` annotations to be applied on Java program elements. You can use the `@XmlElements` annotation with a JavaBean property and a non-static field. This annotation has no default settings and is usually meant for a JavaBean collection property, such as a list.

@XmlElementRef Annotation

The @XmlElementRef annotation maps a JavaBean property to a local element of complex type of an XML schema at runtime. The name of the local element is the same as that of the JavaBean property. You can also use the `@XmlElementRef` annotation with a non-static field, and a `@XmlElementRefs` annotation. The following code snippet shows the default setting of the @XmlElementRef annotation:

```
@XmlElementRef (
  name = "##default",
  namespace = "##default",
  type = DEFAULT.class
)
```

@XmlElementRefs Annotation

The @XmlElementRefs annotation contains numerous `@XmlElementRef` annotations to be applied on Java program elements. It has no default settings.

@XmlElementWrapper Annotation

The @XmlElementWrapper annotation generates a wrapper XML element around collections. You can use the `@XmlElementWrapper` annotation with a JavaBean collection property and a non-static field. The following code snippet shows the default setting of the @XmlElementWrapper annotation:

```
@XmlElementWrapper (
  name = "##default",
  namespace = "##default",
  nillable = false
)
```

@XmlAnyElement Annotation

The @XmlAnyElement annotation maps a JavaBean property to both an XML infoset representation and a JAXB element, or to one of them. The following code snippet shows the default setting of the `@XmlAnyElement` annotation:

```
@XmlAnyElement (
  lax = false,
  value = W3CDomHandler.class
)
```

@XmlAttribute Annotation

The @XmlAttribute annotation maps a JavaBean property to an XML attribute. You can use the `@XmlAttribute` annotation with a JavaBean property and a field. The following code snippet shows the default setting of the annotation:

```
@XmlAttribute (
  name =##default,
  required = false,
  namespace = "##default"
)
```

@XmlAnyAttribute Annotation

The @XmlAnyAttribute annotation maps a `java.util.Map` JavaBean property or field to a map of wildcard attributes. It has no default settings.

@XmlTransient Annotation

The @XmlTransient annotation avoids a JavaBean property being mapped to an XML representation. This annotation is usually used to resolve name collisions that may occur between a JavaBean property name and a field name. It has no default settings.

@XmlValue Annotation

The @XmlValue annotation maps a Java class to an XML schema complex type with a `simpleContent` or an XML schema simple type. You can use the `@XmlValue` annotation with a JavaBean property and a non-static field. It has no default settings.

@XmlID Annotation

The @XmlID annotation maps a JavaBean property to an XML ID. This annotation is used to maintain the referential integrity of an object during serialization and deserialization. It has no default settings.

@XmlIDREF Annotation

The @XmlIDREF annotation maps a JavaBean property to an XML IDREF. This annotation is also used to maintain the referential integrity of an object during serialization and deserialization. It has no default settings.

@XmlList Annotation

The @XmlList annotation maps a JavaBean property or field to a list simple type. This annotation allows the representation of multiple values as whitespace-separated tokens in a single XML element. It has no default settings.

@XmlMixed Annotation

The @XmlMixed annotation marks a multi-valued JavaBean property so that it can support mixed content. You can use the `@XmlMixed` annotation with the `@XmlElementRef`, `@XmlElementRefs`, and `@XmlAnyElement` annotations. It has no default settings.

@XmlMimeType Annotation

The @XmlMimeType annotation associates a MIME type with a JavaBean property to handle the XML corresponding representation. It has no default settings.

@XmlAttachmentRef Annotation

The @XmlAttachmentRef annotation enforces an XML representation of a JavaBean property or field of the `DataHandler` type to be a URI reference to the MIME content, which is stored separately in the form of an attachment. It has no default settings.

@XmlJavaTypeAdapter Annotation

The @XmlJavaTypeAdapter annotation uses an adapter that implements the `@XMLAdapter` annotation to perform custom marshalling. You can use the `@XmlJavaTypeAdapter` annotation with a JavaBean property, field, parameter, package, and the `@XmlJavaTypeAdapters` annotation. It has no default settings.

@XmlJavaTypeAdapters Annotation

The @XmlJavaTypeAdapters annotation contains numerous @XmlJavaTypeAdapter annotations to be applied on a Java program element. It has no default settings.

Exploring an Example of JAXB 2.2 Java/XML Binding

Let's take an example to demonstrate standard binding of a company record.

Listing 19.12 shows the schema as coded in the `company.xsd` file (you can find the `company.xsd` file in the `code/JavaEE/Chapter19/customjaxb` folder in the CD):

Listing 19.12: Showing the Code for the company.xsd File

```
<schema targetNamespace="http://www.example.com/kogent"
  elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:kogent="http://www.example.com/kogent">
<element name="Company">
<complexType>
<sequence>
<element name="CompanyAddress">
<complexType>
<sequence>
<element name="name" type="string"/>
<element name="street" type="string"/>
<element name="city" type="string"/>
<element name="state" type="string"/>
<element name="zipcode" type="string"/>
<element name="phnum" type="string"/>
</sequence>
 </complexType>
</element>
<element name="departments">
<complexType>
<sequence>
<element name="department" type="kogent:DeptType" maxOccurs="unbounded"/>
</sequence>
</complexType>
</element>
</sequence>
</complexType>
</element>
<complexType name="DeptType">
<sequence>
<element name="deptmanager" type="string"/>
<element name="cost" type="double"/>
</sequence>
<attribute name="deptname" use="required" type="string"/>
</complexType>
</schema>
```

Listing 19.12 shows a schema that has one global element with the name `Company`. The type of this element has not been specified. This element contains a sequence of two elements, `CompanyAddress` and `departments`. The `CompanyAddress` element represents the company address of the anonymous address type. The `departments` element represents the list of departments of the company. The sub-elements of the departments element include the `DeptType` type, which is defined later in Listing 19.12. The `DeptType` type

contains two elements, `deptmanager` and `cost`, and an attribute, `deptname`. The `cost` element is of `xs:double`type. The `deptname` attribute is of `xs:string` type.

The Java mapping of the `company.xsd` document is shown in Figure 19.10:

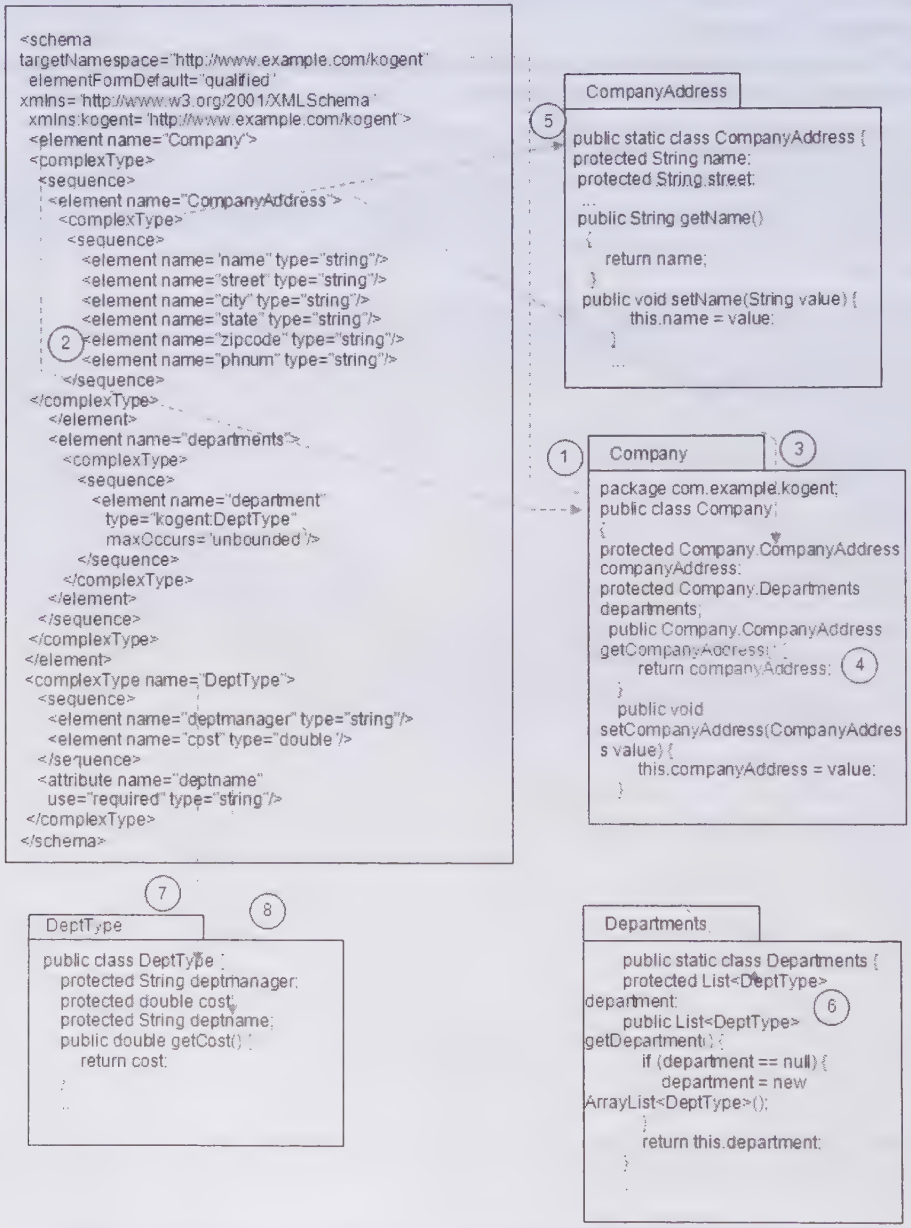


Figure 19.10: Showing the Standard JAXB Binding of the Company Schema

As shown in Figure 19.10, the JAXB standard binding generates two main Java classes from the `company.xsd` schema: `Company` and `DeptType`. The `Company` class has two nested classes, `Company.CompanyAddress` and `Company.Departments`.

The following is the sequence of steps required for binding in JAXB:

1. The target namespace, http://www.example.com/kogent, is mapped to the Java package, com.example.kogent.
2. The anonymous complex type of element, Company, is mapped to the Company class. By default, the anonymous complex type global element of a schema is mapped to a top level Java class.
3. The CompanyAddress element is mapped to the companyAddress property in the Company class. In default mapping, the local elements are mapped to the properties of the top level Java class.
4. The fourth step shows the getter and setter methods for the companyAddress property to map an XML element to the properties of a Java class.
5. The anonymous complex type of the companyAddress element is mapped to the nested Company.CompanyAddress class. By default, the anonymous complex type local element of a schema is mapped to the inner Java classes of a top level class. However, you can map all the local elements to top level classes by using the <jaxb:globalBindings localScoping="toplevel"/> global declaration.
6. The complex type DeptType is mapped to the second main class from the schema, the DeptType Java class. In default mapping, named complex types are mapped to the top level Java value class.
7. By default, the xs:double type is mapped to the java.math.Double type. You can customize this mapping by using the jaxb:javaType declaration or the @XmlElement.type annotation.
8. By default, attributes are also mapped to the properties of a Java class. The deptname attribute is mapped to the deptname property of the DeptType class.

Figure 19.11 shows how to manually generate the classes in the JAXB runtime using the `xjc` compiler:

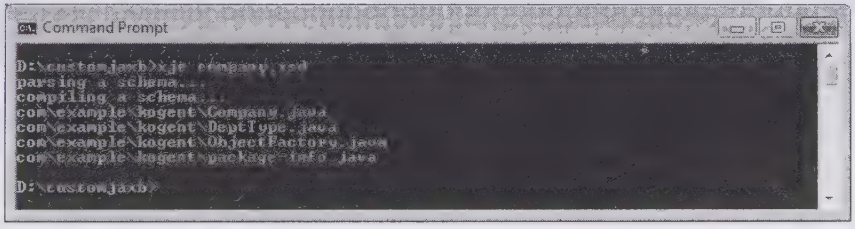


Figure 19.11: Displaying the Parsing of an XML Schema Instance

Now, let's move further to understand anonymous and named complex type mapping.

Mapping Anonymous Complex Type

To understand the mapping of anonymous complex type, let's take an example based on the `company.xsd` schema shown in Listing 19.12, which includes three anonymous types. The root anonymous type is mapped to the `Company.java` class and the other two anonymous types are mapped to its inner classes named, `CompanyAddress` and `departments` class.

Let's see a part of the code of the JAXB-generated `Company` value class (or the Java mapping of the `company.xsd` document). This class contains annotations to perform runtime marshalling. Listing 19.13 shows the code for the `Company` JAXB value class (you can find the `Company.java` file in the `code/JavaEE/Chapter19/customjaxb/com/example/kogent` folder in the CD):

Listing 19.13: Showing the Code for theCompany.java File

```
package com.example.kogent;
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
 "companyAddress",
```

```
    "departments"
})
@XmlRootElement(name = "Company")
public class Company {
    @XmlElement(name = "CompanyAddress", required = true)
    protected Company.CompanyAddress companyAddress;
    @XmlElement(required = true)
    protected Company.Departments departments;
    public Company.CompanyAddress getCompanyAddress() {
        return companyAddress;
    }
    public void setCompanyAddress(Company.CompanyAddress value) {
        this.companyAddress = value;
    }
    public Company.Departments getDepartments() {
        return departments;
    }
    public void setDepartments(Company.Departments value) {
        this.departments = value;
    }
    @XmlAccessorType(XmlAccessType.FIELD)
    @XmlType(name = "", propOrder = {
        "name",
        "street",
        "city",
        "state",
        "zipcode",
        "phnum"
    })
    public static class CompanyAddress {
        @XmlElement(required = true)
        protected String name;
        @XmlElement(required = true)
        protected String street;
        @XmlElement(required = true)
        protected String city;
        @XmlElement(required = true)
        protected String state;
        @XmlElement(required = true)
        protected String zipcode;
        @XmlElement(required = true)
        protected String phnum;
        public String getName() {
            return name;
        }
        public void setName(String value) {
            this.name = value;
        }
        public String getStreet() {
            return street;
        }
        public void setStreet(String value) {
            this.street = value;
        }
        public String getCity() {
            return city;
        }
        public void setCity(String value) {
            this.city = value;
        }
        public String getState() {
            return state;
        }
        public void setState(String value) {
```

```
                this.state = value;
        }
        public String getZipcode() {
                return zipcode;
        }
        public void setZipcode(String value) {
                this.zipcode = value;
        }
        public String getPhnum() {
                return phnum;
        }
        public void setPhnum(String value) {
                this.phnum = value;
        }
}
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
        "department"
})
public static class Departments {
        @XmlElement(required = true)
        protected List<DeptType> department;
        public List<DeptType> getDepartment() {
                if (department == null) {
                        department = new ArrayList<DeptType>();
                }
                return this.department;
        }
}
}
```

Listing 19.13 does not specify any value for the `name` attribute. Therefore, all complex types defined by using `@XmlType` annotations are anonymous types.

As you can see that, the `propOrder` attribute of the `@XmlType` annotation specifies the order of fields of complex types. This order must match with the order specified in the `<sequence>` definition of complex types in the `company.xsd` schema. The `@XmlType` annotation is necessary, as the Java class does not sequence the properties by itself. Listing 19.13 uses `@XmlRootElement` annotation that specifies the `Company` value for its `name` attribute and this value is the same as the name of the global element of the `company.xsd` schema. Listing 19.13 uses the `@XmlAccessorType` annotation to map fields and properties of Java class with the corresponding elements in a schema. Listing 19.13 uses `@XmlElement` annotations to map a specific property to a schema element definition. For example, properties such as `companyAddress` and `departments` are mapped to the `CompanyAddress` and `departments` elements of the `company.xsd` schema.

Listing 19.13 contains two inner classes, `CompanyAddress` and `Departments`, as there are only two anonymous complex types in the `company.xsd` schema. By default, schema element definitions with anonymous complex types are mapped to inner classes. The `CompanyAddress` class is simple to understand.

By default, the element which occurs several times is mapped to java.util.List<T> schema element, where T is the Java class of the element's type.

Mapping Named Complex Type

The `company.xsd` schema also contains a named complex type, `DeptType`. As we know that, the named complex types are not mapped to inner classes. Therefore, the `DeptType` named complex type is mapped to the external `kogent.DeptType` (Listing 19.14) Java class. Listing 19.14 shows the source code of the `DeptType.java` file (you can find the `DeptType.java` file in the `code/JavaEE/Chapter19/customjaxb/com/example/kogent` folder in the CD):

Listing 19.14: Showing the Code for the DeptType.java File

```
package com.example.kogent;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
```

```
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "DeptType", propOrder = {
   "deptmanager",
   "cost"
})
public class DeptType {
   @XmlElement(required = true)
   protected String deptmanager;
   protected double cost;
   @XmlAttribute(required = true)
   protected String deptname;
   public String getDeptmanager() {
         return deptmanager;
   }
   public void setDeptmanager(String value) {
         this.deptmanager = value;
   }
   public double getCost() {
         return cost;
   }
   public void setCost(double value) {
         this.cost = value;
   }
   public String getDeptname() {
         return deptname;
   }
   public void setDeptname(String value) {
         this.deptname = value;
   }
}
```

In Listing 19.14, the `cost` element of the `DeptType` complex type is mapped to the `cost` property with the `Double` type of the `DeptType` Java class. The `deptname` attribute of the schema is mapped to the `deptname` property of the `DeptType` Java class by using the `@XmlAttribute` annotation.

Implementing Type Mappings with JAXB 2.2

The section focuses on using custom Java classes with JAXB standard mapping to implement your own type mapping. The standard Java/XML type mapping of JAXB maps Java classes to XML schema elements using the following notation:

```
<{http://www.example.com/kogent}company,com.example.kogent.Company>
```

The preceding notation is of the <XMLType, JavaType> form; where XML type is the fully qualified name of the XML schema element and JavaType is the fully qualified name of the Java class.

You can map XML schema elements to custom Java classes. Let's try to implement the mapping between the `<kogent:Company>` and `<CustomCompany>` classes. In this mapping, the `CustomCompany` class is not a JAXB generated class; rather, we create this class. This mapping happens in two steps. In the first step, JAXB creates the JAXB-generated `com.example.kogent.Company` class and then maps this class to the `CustomCompany` Java class.

Listing 19.15 shows the code of the `CustomCompany.java` file (you can find `CustomCompany.java` file in the `code/JavaEE/Chapter19/customjaxb` folder in the CD):

Listing 19.15: Showing the Code for the CustomCompany.java File

```
package classes;
import java.util.ArrayList;
import java.util.List;
public class CustomCompany {
   private CustomAddress companyAddress;
   private List<CustomDept> deptList;
```

```
    public CustomCompany(String name, String street, String city, String state,
    String zip, String phone) {
          this(new CustomAddress(name, street, city, state, zip, phone));
    }
    public CustomCompany(CustomAddress addr) {
          this.companyAddress = addr;
          deptList = new ArrayList<CustomDept>();
    }
    public CustomAddress getCompanyAddress() {
          return companyAddress;
    }
    public List<CustomDept> getDeptList() {
          return deptList;
    }
}
```

Listing 19.15 uses the `companyAddress` property of the `CustomAddress` Java type instead of the JAXB-generated `com.example.kogent.CompanyAddress` type. It uses the `deptList` property of the `List<CustomDept>` type rather than the JAXB-generated `com.example.kogent.departments` type. The `deptList` property stores the information about the departments of the company.

Mapping an instance of the `classes.CustomCompany` class to an instance of the `kogent:Company` element requires that the `companyAddress` property is mapped to the `classes.CustomAddress` type and the `List<classes.CustomDept>` type is mapped to the `kogent:departments` element. Listing 19.16 shows the source code of the `CustomAddress.java` file (you can find the `CustomAddress.java` file in the `code/JavaEE/Chapter19/customjaxb` folder in the CD):

Listing 19.16: Showing the Code for the CustomAddress.java File

```
package classes;
public class CustomAddress {
  protected String name;
  protected String street;
  protected String city;
  protected String state;
  protected String zipcode;
  protected String phnum;
  public CustomAddress(String name, String street, String city, String state,
  String zipcode, String phnum) {
        this.name = name;
        this.street = street;
        this.city = city;
        this.state = state;
        this.zipcode = zipcode;
        this.phnum = phnum;
  }
}
```

In Listing 19.16, `CustomAddress` class is not a Java bean and acts only as a container for its sub properties, such as `name`, `street`, and `city`.

Let's take a look at the `CustomDept.java` class. Listing 19.17 shows the source code of the `CustomDept.java` file (you can find the `CustomDept.java` file in the `code/JavaEE/Chapter19/customjaxb` folder in the CD):

Listing 19.17: Showing the Code for the CustomDept.java File

```
package classes;
public class CustomDept {
  private String deptmanager;
  private float cost;
  private String deptname;
  public CustomDept(String deptmanager, float cost, String deptname)
  throws Exception {
        if (deptname == null) {
              throw new Exception("department must has some name");
        }
```

```
            this.deptname = deptname;
            this.cost = cost;
            this.deptmanager = deptmanager;
      }
      public float getCost() {
            return cost;
      }
      public void setCost(float cost) {
            this.cost = cost;
      }
      public String getDeptname() {
            return deptname;
      }
      public void setDeptname(String deptname) {
            this.deptname = deptname;
      }
      public String getDeptmanager() {
            return deptmanager;
      }
      public void setDeptmanager(String deptmanager) {
            this.deptmanager = deptmanager;
      }
}
```

In Listing 19.17, the `CustomProduct.cost` property is of float type while the `ProdType.cost` property is of double type.

Let's see the Java class that maps the JAXB-generated `com.example.kogent.Company` class to the `CustomCompany` Java class. Listing 19.18 shows the source code of the `CustomCompanySerializer.java` file (you can find the `CustomCompanySerializer.java` file in the `code/JavaEE/Chapter19/customjaxb` folder in the CD):

Listing 19.18: Showing the Code for the CustomCompanySerializer.java File

```
package classes;
import java.io.ByteArrayOutputStream;
import java.io.StringReader;
import java.math.BigInteger;
import java.io.File;
import java.util.List;
import javax.xml.XMLConstants;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import com.example.kogent.DeptType;
import com.example.kogent.Company;
public class CustomCompanySerializer{
 public static void main(String[] args) throws Exception {
        CustomCompanySerializer serializer =
        new CustomCompanySerializer();
        CustomCompany customCompany = new CustomCompany(
        "ABC",
        "Street No . 1",
        "Shakarpur", "ND", "110092",
        "9971782025");
        customCompany.getDeptList().add(new CustomDept( "Mr .Singh", (float) 2.99,
        "Content Solutions"));
        customCompany.getDeptList().add(new CustomDept("Mrs. Gupta", (float) 3.99, "HR
        Solutions"));
```

```
            customCompany.getDeptList().add(new CustomDept("Mr. Kumar", (float) 5.34,
            "Production"));
            Transformer xform = TransformerFactory.newInstance().newTransformer();
            xform.setOutputProperty(OutputKeys.INDENT, "yes");
            xform.transform(
            serializer.getXML(customCompany),
            new StreamResult(System.out));
    }
    public Source getXML(CustomCompany company) {
    // create the JAXB SimpleOrder
    Company jaxbCompany = new Company();
    // map the addresses
    CustomAddress customAddress = company.getCompanyAddress();
    Company.CompanyAddress companyAddress = new Company.CompanyAddress();
    companyAddress.setName(customAddress.name);
    companyAddress.setCity(customAddress.city);
    companyAddress.setPhnum(customAddress.phnum);
    companyAddress.setState(customAddress.state);
    companyAddress.setStreet(customAddress.street);
    companyAddress.setZipcode(customAddress.zipcode);
    jaxbCompany.setCompanyAddress(companyAddress);
    // map the items
   jaxbCompany.setDepartments(new Company.Departments()); // needed to avoid NPE
    List<DeptType> jaxbDeptList = jaxbCompany.getDepartments().getDepartment();
    for (CustomDept customDept : company.getDeptList()) {
    DeptType jaxbDept = new DeptType();
    //    jaxbItem.setPrice((double) myItem.getPrice());
    jaxbDept.setCost(Double.parseDouble(Float.toString(customDept.getCost())));
    jaxbDept.setDeptname(customDept.getDeptname());
    jaxbDept.setDeptmanager(customDept.getDeptmanager());
    jaxbDeptList.add(jaxbDept);
    }
    try {
    JAXBContext jaxbContext = JAXBContext.newInstance("com.example.kogent");
    Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
    SchemaFactory sf =
    SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
    Schema schema = sf.newSchema(
    new File("company.xsd"));
    jaxbMarshaller.setSchema(schema);
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    jaxbMarshaller.marshal(jaxbCompany, baos);
    return new StreamSource(new StringReader(baos.toString()));
    } catch (Exception e) {
            throw new RuntimeException(e);
    }
    }
   }
```

Listing 19.18 imports an instance of the JAXB-generated `com.example.kogent.Company` class, populates it with the data of the `CustomCompany` Java class, and marshals the JAXB instance to get an XML representation. Listing 19.18 first populates an instance of the `CustomCompany` Java class with the respective values. It invokes the `marshal()` method of the `Transformer` object with two arguments. The first argument makes a call to the `getXML()` method by passing the `CustomCompany` object. The second argument is of the `StreamResult` type.

The `getXML()` method creates an instance of the JAXB generated `Company` Java class. This method populates the `Company.CompanyAddress (companyAddress)` instance from the instance of `CustomAddress` obtained using the `company.getCompanyAddress()` method.

Then, the code provided in Listing 19.18 initializes a list of `departments` of the company to the empty list. This initialization is necessary as JAXB does not map the `kogent:departments` element to the `List<T>` type. The `for loop` populates the members of `List<DeptType>` from the members of the `company.getDeptList()`

method that is of the `List<CustomAddress>` type. The setter methods of the JAXB generated `DeptType` class are invoked to set its members.

Listing 19.18 also creates an instance of the `JAXBContext class` by passing the `com.example.kogent` package to the location where JAXB generated classes are found. It then creates an instance of the `Marshaller` class and activates validation on this instance by calling its `setSchema()` method. Finally, `jaxbCompany` is marshaled into `the ByteArrayOutputStream type` and returned as a StreamSource.

Figure 19.12 shows the output of the `customjaxb` application:

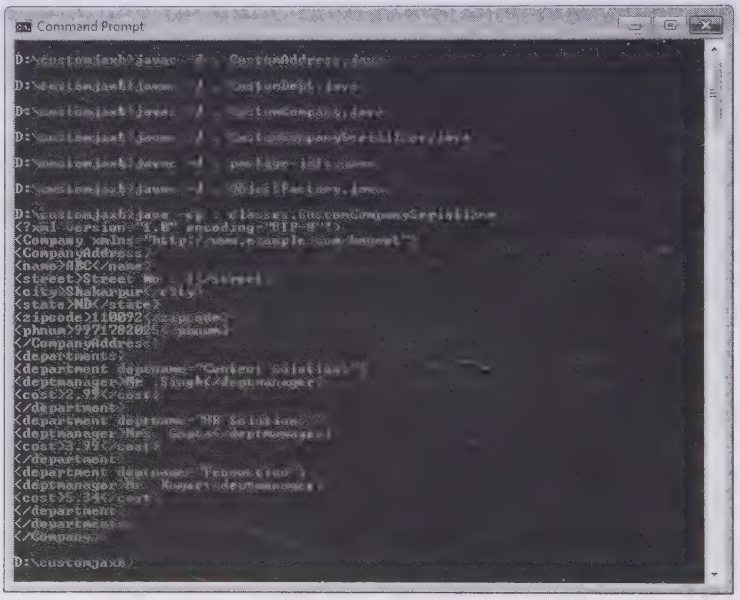


Figure 19.12: Showing the Output of the customjaxb Application

Implementing Type Mappings with JAXB 2.2 Annotations

The section focuses on the use of annotations in existing POJOs to eliminate the need of mapping code. The annotated POJOs are directly mapped to target schema.

Listing 19.19 shows the annotated version of the `CustomCompany.java` file (you can find the `CustomCompany.java` file in the `code/JavaEE/Chapter19/customjaxb_with_annotations` folder in the CD):

Listing 19.19: Showing the Code for the CustomCompany.java File

```
package classes;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlElement;
import java.util.ArrayList;
import java.util.List;
@XmlRootElement(name="Company")
public class CustomCompany {
  private CustomAddress companyAddress;
  private List<CustomDept> deptList;
  public CustomCompany() {}
  public CustomCompany(String name, String street, String city, String state,
  String zip, String phone) {
        this(new CustomAddress(name, street, city, state, zip, phone));
  }
  public CustomCompany(CustomAddress addr) {
```

```
        this.companyAddress = addr;
        deptList = new ArrayList<CustomDept>();
    }
    public CustomAddress getCompanyAddress() {
        return companyAddress;
    }
    public void setCompanyAddress(CustomAddress addr) {
        this.companyAddress = addr;
    }
    @XmlElementWrapper(name="departments")
    @XmlElement(name="department")
    public List<CustomDept> getDeptList() {
        return deptList;
    }
    public void setDeptList(List<CustomDept> deptList) {
        this.deptList = deptList;
    }
}
```

Listing 19.19 does not use annotations on all its components as some components are mapped by the JAXB Java to XML binding default standard. Listing 19.19 uses some additional annotations as compared to the `Company.java` JAXB generated class. It uses the `@XmlRootElement` annotation to map this class to the global element of schema. The `@XmlRootElement` annotation has an optional `namespace` element to specify the namespace of the global element of schema. It uses `@XmlElement` annotation to map an XML element to the `deptList` type.

Note that another property `companyAddress` does not use the `@XmlElement` annotation, as this property is mapped to XML element named `companyAddress` by default. Listing 19.19 uses the `@XmlElementWrapper` annotation so that each element in `deptList` list must wrapped under the `departments` element.

As earlier said, `@XmlRootElement` annotation does not specify namespace of target XML schema. By default, JAXB obtains target namespace from package name. For example, if package name is classes, the target namespace will be http://classes. Listing 19.20 shows the source code of the `package-info.java` file (you can find the `package-info.java` file in the `code/JavaEE/Chapter19/customjaxb_with_annotations` folder in the CD):

Listing 19.20: Showing the Code for the package-info.java File

```
@XmlSchema(
    namespace = "http://www.example.com/kogent",
    elementFormDefault=XmlNsForm.QUALIFIED)
package classes;
import javax.xml.bind.annotation.XmlNsForm;
import javax.xml.bind.annotation.XmlSchema;
```

Listing 19.20 uses the `@XmlSchema` annotation to map a package to an XML schema. The `namespace` element maps package `classes` to the `http://www.example.com/kogent` namespace.

Listing 19.21 shows the annotated version of the `CustomAddress` Java class (you can find the `CustomAddress.java` file in the `code/JavaEE/Chapter19/customjaxb_with_annotations` folder in the CD):

Listing 19.21: Showing the Code for the CustomAddress.java File

```
package classes;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "",
propOrder = {"name", "street", "city", "state", "zipcode", "phnum"})
public class CustomAddress {
  @XmlElement(namespace = "http://www.example.com/kogent")
```

```
    protected String name;
    @XmlElement(namespace = "http://www.example.com/kogent")
    protected String street;
    @XmlElement(namespace = "http://www.example.com/kogent")
    protected String city;
    @XmlElement(namespace = "http://www.example.com/kogent")
    protected String state;
    @XmlElement(namespace = "http://www.example.com/kogent")
    protected String zipcode;
    @XmlElement(namespace = "http://www.example.com/kogent")
    protected String phnum;
    // need a no-arg constructor
    public CustomAddress() {};
    //! </example>
    public CustomAddress(String name, String street, String city, String state,
    String zipcode, String phnum) {
          this.name = name;
          this.street = street;
          this.city = city;
          this.state = state;
          this.zipcode = zipcode;
          this.phnum = phnum;
    }
}
```

Listing 19.21 uses the `@XmlAccessorType` annotation, as this class does not have setter or getter methods for its properties. Listing 19.21 requires the use of `@XmlType` annotation as the `CustomAddress` class is mapped to an anonymous complex type. The value of the `name` element indicates that this class is mapped to an anonymous complex type. The `propOrder` element specifies the order of properties in the target XML schema `<sequence>` element. Listing 19.21 also uses the `@XmlElement` annotation for each property to specify namespace of each target XML element. The `@XmlElement` annotation is required as the `CustomAddress` class is mapped to anonymous complex type and there is no namespace to be inherited from parent class.

Listing 19.22 shows the annotated version of the `CustomDept` class (you can find the `CustomDept.java` file in the `code/JavaEE/Chapter19/customjaxb_with_annotations` folder in the CD):

Listing 19.22: Showing the Code for the CustomDept.java File

```
package classes;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlType;
@XmlType(name = "DeptType", propOrder = {"deptmanager", "cost"})
public class CustomDept {
  private String deptname;
  private float cost;
  private String deptmanager;
  public CustomDept() {};
  public CustomDept(String deptmanager, float cost, String deptname)
  throws Exception {
        if (deptname == null) {
              throw new Exception("department must has some name");
        }
        this.deptname = deptname;
        this.cost = cost;
        this.deptmanager = deptmanager;
  }
  public float getCost() {
        return cost;
  }
  public void setCost(float cost) {
        this.cost = cost;
  }
  @XmlAttribute
  public String getDeptname() {
        return deptname;
```

```
    }
    public void setDeptname(String deptname) {
          this.deptname = deptname;
    }
    public String getDeptmanager() {
          return deptmanager;
    }
    public void setDeptmanager(String deptmanager) {
          this.deptmanager = deptmanager;
    }
}
```

Listing 19.22 uses the `@XmlType` annotation to map the `kogent:DeptType` complex type to the `CustomDept` Java class. As it does not use any `@XmlAccessorType` annotation, mapping of properties to schema elements is performed by default. However, the `deptname` property needs to be mapped to the attribute of an XML element. Therefore, it associates the `@XmlAttribute` annotation with the `deptname` property.

Listing 19.23 marshals and unmarshals the instance of the `CustomCompanySerializer.java` class (you can find the `CustomCompanySerializer.java` file in the `code/JavaEE/Chapter19/customjaxb_with_annotations` folder in the CD):

Listing 19.23: Showing the Code for the CustomCompanySerializer.java File

```
package classes;
import java.io.ByteArrayOutputStream;
import java.io.StringReader;
import java.io.File;
import javax.xml.XMLConstants;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import classes.CustomCompany;
public class CustomCompanySerializer {
  public static void main(String[] args) throws Exception {
  CustomCompanySerializer serializer =
  new CustomCompanySerializer();
  CustomCompany customCompany = new CustomCompany(
  "ABC",
  "Street No . 1",
  "Shakarpur", "ND", "110092",
  "9971782025");
  customCompany.getDeptList().add(new CustomDept( "Mr .Singh", (float) 2.99, "Content
  Solutions"));
  customCompany.getDeptList().add(new CustomDept("Mrs. Gupta", (float) 3.99,
  "HR Solutions"));
  customCompany.getDeptList().add(new CustomDept("Mr. Kumar", (float) 5.34,
  "Production"));
  try {
  JAXBContext jaxbContext = JAXBContext.newInstance(CustomCompany.class);
  Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
  jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
  Boolean.TRUE);
  ByteArrayOutputStream ba = new ByteArrayOutputStream();
  jaxbMarshaller.marshal(customCompany, ba);
  System.out.println(ba.toString());
  Unmarshaller u = jaxbContext.createUnmarshaller();
  CustomCompany cc =
  (CustomCompany) u.unmarshal(new StringReader(ba.toString()));
  System.out.println("phone = " + cc.getCompanyAddress().phnum);
  } catch (Exception e) {
        e.printStackTrace();
  }
  }
}
```

Listing 19.23 creates an instance of the `CustomCompany` Java class and initializes it. It then creates a `JAXBcontext` instance by passing `CustomCompany.class` parameter value to the `newInstance()` method. Listing 19.23 both marshals the `CustomCompany` instance into XML representation and output XML representation on to the `System.out` stream. It also unmarshals XML representation into the `CustomCompany` Java class, and then prints the phone number of the company specified in the Java instance.

Figure 19.13 shows the output of the `customjaxb_with_annotations` application:

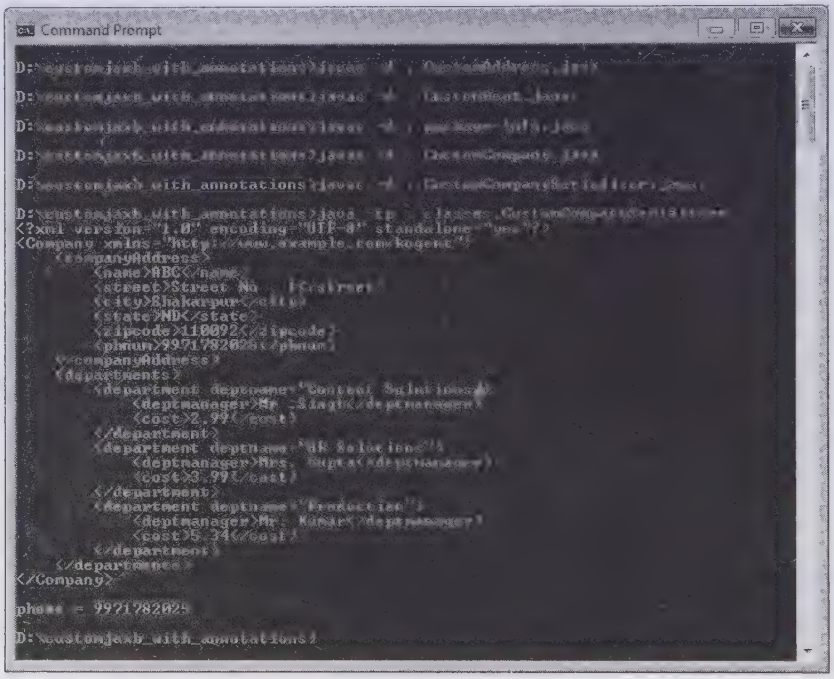


Figure 19.13: Showing the Output of the customjaxb_with_annotations Application

Using the WSEE and WS-Metadata Specifications

The WSEE and WS-Metadata specifications provide information about packaging and deployment of Web services. WS-Metadata provides a set of standard annotations to configure the JWS container and deploy a Java class as a Web service. The container generates artifacts, such as WSDL representation of a Web service, from these annotations. In earlier version of Java, such as J2EE, developers needed to provide a configuration file, such as the `webservices.xml Deployment Descriptor`, to configure a Java class as a Web service. As WS-Metadata provides annotations to deploy Java class declaratively as a Web service, use of `Deployment Descriptor` becomes optional in Java EE5 and Java EE 6.

The WSEE specification makes Web service applications portable across various Java EE application server implementations. It defines how the Java class and generated artifacts are packaged for deployment of a Web service application. It also defines the type of the Java class to be deployed as a Web service. The Java class can be either a servlet or an EJB endpoint. This specification also provides information about packaging these endpoints and Deployment Descriptors, if required.

In brief, WS-Metadata makes development and deployment of Web services easier by providing annotations. WSEE specification describes the correct use and packaging of WS-Metadata annotated classes and other generated artifacts.

The different deployment methods provided by these specifications are given as follows:

- Deployment using a servlet endpoint
- Deployment using an EJB endpoint

- Deployment without Deployment Descriptors
- Deployment with Deployment Descriptors

Deployment using a Servlet Endpoint

When you implement SIBs as servlet endpoints, you need to package them and other generated artifacts in a WAR file. Figure 19.14 shows the structure of this WAR file:

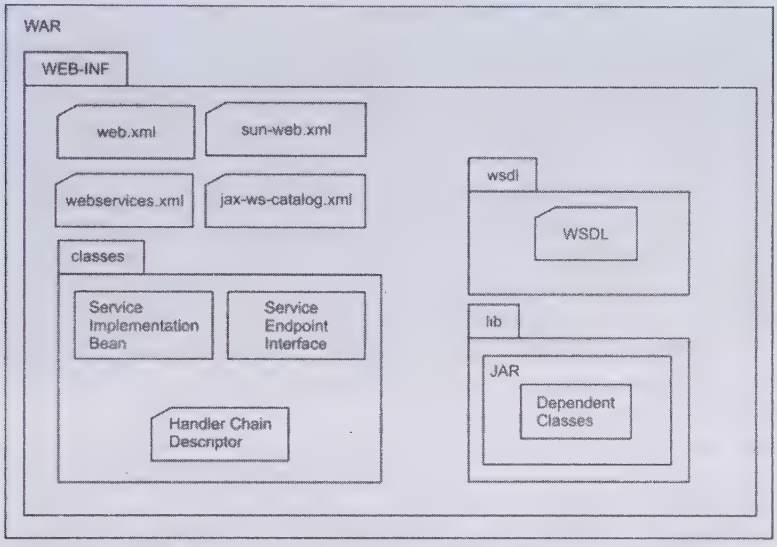


Figure 19.14: Packaging a Servlet Endpoint

Deployment using an EJB Endpoint

When you implement an SIB as an EJB endpoint (stateless session bean), you need to package it and other generated artifacts in an EJB-JAR file. Figure 19.15 shows the structure of this EJB-JAR file:

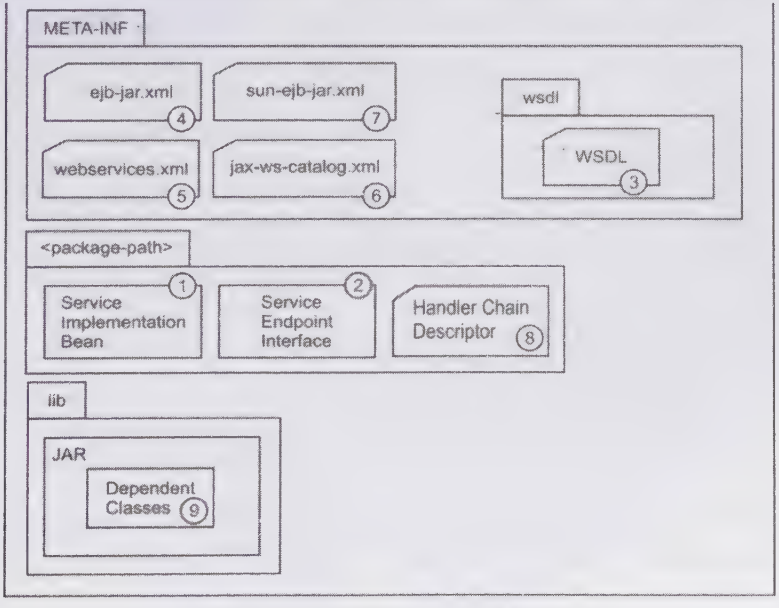


Figure 19.15: Packaging an EJB Endpoint

The following list explains each numbered component in Figure 19.15:

1. **Service implementation bean**—Remains inside the `<package-path>/` directory by convention, where `<package-path>` is a package name of the SIB bean class. You are not restricted to place the bean class in this location rather you can store the bean class in another jar file. The SIB class must use the `@Stateless` annotation and the `@WebService` or `@WebServiceProvider` annotation.
2. **SEI**—Remains inside the `<package-path>/` directory by convention, where `<package-path>` is a package name of the SEI bean class. This location is not restricted for the SEI interface rather you can place it at another location in your application directory. When SEI is present inside a port component, then the `@WebService.endpointInterface` attribute of SIB must have value equal to the name of the SEI.
3. **WSDL**—Remains inside the `META-INF/wsdl` directory by convention. This location is not restricted for the WSDL document rather you can place it at another location in your application directory. This document is optional.
4. **ejb-jar.xml**—Remains always under the `META-INF` directory. The `ejb-jar.xml` file must always be placed at the same location when present. However, this file is optional.
5. **webservices.xml**—Remains always under the `META-INF` directory. The `webservices.xml` file must always be placed at the same location when present. However, this document is optional.
6. **jax-ws-catalog.xml**—Remains always under the `META-INF` directory. The `jax-ws-catalog.xml` descriptor file is optional and is used when you use `OASIS XML` catalogs.
7. **sun-ejb-jar.xml**—Remains under the `META-INF` directory. The `sun-ejb-jar.xml` file is optional. It is Glassfish's EJB Deployment Descriptor.
8. **Handler chain descriptor**—Remains under the `<package-path>/` directory. This location is not restricted for the `Handler Chain Descriptor` document rather you can place it at external location. This file has no standard name and is optional.
9. **Dependent classes**—Represents the classes on which SIB or SEI depends upon. These are bundled in a JAR file at the root of the EAR file.

Deployment without Deployment Descriptors

Java EE 5 and Java EE 6 offer the flexibility of avoiding the use of Deployment Descriptors to configure Web services. You can deploy Web services without the Deployment Descriptor by using:

- An SIB
- A SEI

Using an SIB

Let's create a Web service named `ImplementingSB` to learn how an SIB is used to deploy a Web service. In order to do this, you need to annotate the SIB class with the `@WebService` annotation. Listing 19.24 shows the source code of the SIB class, `Greeting` (you can find the `Greeting.java` file in the `code/JavaEE/Chapter19/ImplementingSB/src` folder in the CD):

Listing 19.24: Showing the Code for the Greeting.java File

```
package src;
import javax.jws.WebService;
@WebService
public class Greeting {
   public String dispGreeting(String s) {
         return "Welcome: " + s;
   }
}
```

Listing 19.24 uses the `@WebService` annotation with the `Greeting.java` class. Therefore, it acts as an implementation class of the Web service. It contains the `dispGreeting()` method, which displays a welcome message. Run the `apt` command from the Command Prompt. The `apt` command is a command line utility used

for annotation processing. Create a folder named `generated` under the `ImplementingSB` folder. Run the following `apt` command and generate the required wrapper classes:

```
D:\ImplementingSB>apt -d generated src/Greeting.java
```

The `apt` command generates JAX-WS source files in the `src` folder under the `jaxws` directory and JAX-WS class files in the `generated/src/jaxws` directory. This tool also compiles the `Greeting.java` SIB class and places it under `generated/src` folder, as shown in Figure 19.16:

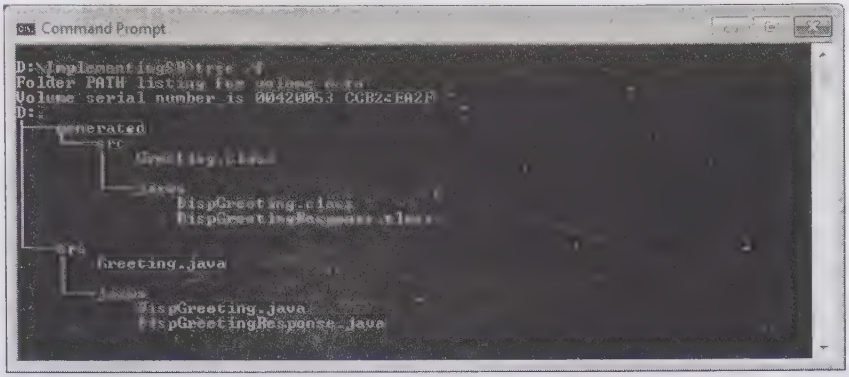


Figure 19.16: Showing the Generated JAX-WS Source and Class Files

You now need to package the `ImplementingSB` application. Create another folder `c:\ImplementingSB`. Directory structure of `c:\ImplementingSB` folder is as shown in Figure 19.17:

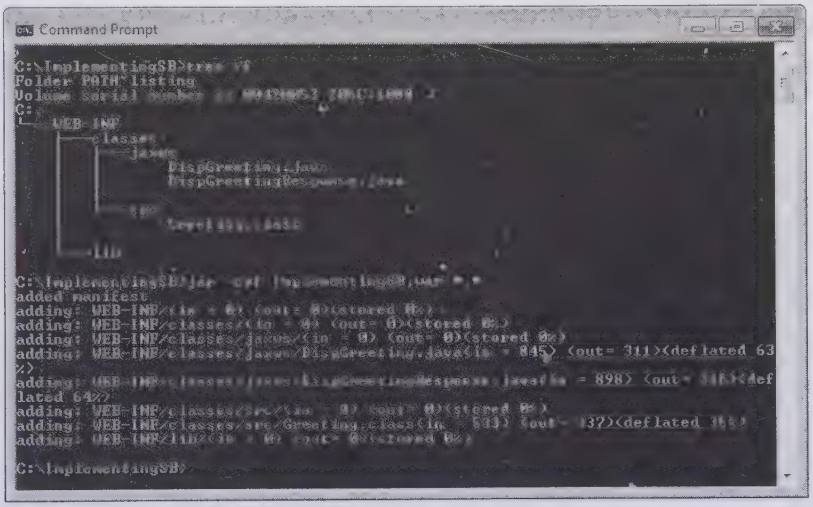


Figure 19.17: Showing the Directory Structure of ImplementingSB Web Service Application

Figure 19.17 also shows the command to package the ImplementingSB application, which generates the ImplementingSB.war file. Next, copy the `ImplementingSB.war` file in the `autodeploy` directory of the domain of the Glassfish V3 application server. GlassFish server automatically generates internal descriptors and WSDL. WSDL document is published on the URL `http://localhost:8080/ImplementingSB/GreetingService?wsdl`. Run `wsimport` command on the deployed WSDL URL to generate some classes. Execute the following command to run `wsimport`:

```
D:\ImplementingSB>wsimport -p client -keep
http://localhost:8080/ImplementingSB/GreetingService?wsdl
```

Verify all the generated classes and Java source files in the `d:\ImplementingSB\client` folder, as shown in Figure19.18:

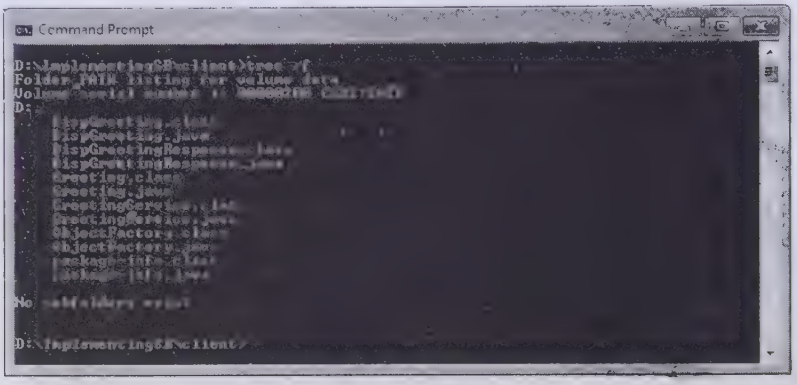


Figure 19.18: Showing all the Java and Class Files Generated by the wsimport Tool

Let's create a Web service client, `ClientApp.java`, to invoke the Web service endpoint and its `dispGreeting()` method. Listing 19.25 shows the source code of the `ClientApp.java` file (you can find the `ClientApp.java` file in the `code/JavaEE/Chapter19/ImplementingSB` folder in the CD):

Listing 19.25: Showing the Code for the ClientApp.java File

```
package client;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.lang.Exception;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
public class ClientApp
{
  public static void main(String args[])
  {
  try
  {
  URL wsdlURL = null;
  wsdlURL= new URL("http://localhost:8080/ImplementingSB/GreetingService?wsdl");
  InputStream is = (InputStream) wsdlURL.getContent();
  Transformer t = TransformerFactory.newInstance().newTransformer();
  t.setOutputProperty(OutputKeys.INDENT,"yes");
  t.setOutputProperty(OutputKeys.METHOD,"xml");
  t.setOutputProperty(OutputKeys.MEDIA_TYPE,"text/xml");
  t.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION,"yes");
  t.transform(new StreamSource(is), new StreamResult(System.out));
  System.out.println();
  QName srvcQName = new QName("http://src/", "GreetingService");
  QName portQName = new QName("http://src/", "GreetingPort");
  Service srvc = Service.create(wsdlURL, srvcQName);
  Greeting port = (Greeting) srvc.getPort(portQName, Greeting.class);
  String result = port.dispGreeting("Java Programmer");
  System.out.println(result);
  }
  catch (Exception e)
  {
        System.out.println(e);
  }
  }
}
```

Listing 19.25 creates a URL instance with the generated URL of the `http://localhost:8080/ImplementingSB/GreetingService?wsdl` WSDL. Listing 19.25 retrieves the contents of the WSDL document and places it into the InputStream, `is`. The listing then sets some properties

related to how the WSDL document should be displayed at the command prompt. Listing 19.25 uses the `transform()` method of the `Transformer` object to transform the source stream into the output stream. Listing 19.25 then uses the `Service.create()` method to configure a `Service` instance with the URL of WSDL document and the `srvcQName` instance of the QName class. After creating the service instance, it invokes the `Service.getPort()` method, which accepts the Qname of `wsdl:portType` and Greeting.class as parameters. The `getPort()` method returns the implementation instance of the `GreetingPort` class.

Compile the `ClientApp.java` file using the `javac` command and place the generated `ClientApp.class` file in the `D:\ImplementingSB\client` folder.

Figure 19.19 shows the output of the compilation and execution of the `ClientApp.java` file:

Figure 19.19: Showing the Output of the ImplementingSB Web Service Application

Figure 19.19 displays the generated WSDL document, `GreetingService.wsdl`, and the output string `Welcome: Java Programmer`, of the `dispGreeting()` method.

Using a SEI

You can also use an SEI to deploy a Web service. Let's create a Web service named `sibwithsei` to learn how a SEI can be used to deploy a Web service. Let's first create an SIB class. Listing 19.26 shows the source code of the SIB class named `Greeting` (you can find the `Greeting.java` file in the `code/JavaEE/Chapter19/sibwithsei/src` folder in the CD):

Listing 19.26: Showing the Code for the Greeting.java File

```
package src;
import javax.jws.WebService;
@WebService(endpointInterface="src.GreetingInf")
public class Greeting {
   public String dispGreeting(String s) {
         return "Welcome: " + s;
   }
   public String dispGoodbye(String s) {
         return "Goodbye: " + s;
   }
}
```

Listing 19.26 uses the `@webService` annotation with the `Greeting` class. This SIB class references an endpoint interface using the `src.GreetingInf` location. Listing 19.27 shows the code for the `GreetingInf.java` file (you can find the `GreetingInf.java` file in the `code/JavaEE/Chapter19/sibwithsei/src` folder in the CD):

Listing 19.27: Showing the Code for the GreetingInf.java File

```
package src;
import javax.jws.WebService;
@WebService
public interface GreetingInf {
   public String dispGreeting(String s);
}
```

Compile the `GreetingInf.java` class using the following command:

```
D:\sibwithsei>javac -d . src/GreetingInf.java
```

Compile the `Greeting.java` class using the following command:

```
D:\sibwithsei>javac -d . src/Greeting.java
```

Run the `apt` command and generate the required wrapper classes. Make sure that the `d:\sibwithsei\generated` folder is created before the execution of the following command:

```
D:\sibwithsei>apt -d generated src/Greeting.java
```

To deploy the `sibwithsei` application, create another folder `c:\sibwithsei`. The directory structure of the `c:\sibwithsei` folder is shown in Figure 19.20:

Figure 19.20: Showing the Directory Structure of the sibwithsei Web Service Application

Create the required WAR file using the command shown in Figure 19.20. Copy the `sibwithsei.war` file in the `autodeploy` directory of the domain of the Glassfish V3 application directory. GlassFish server automatically

generates WSDL and internal descriptors. WSDL document is published on the URL `http://localhost:8080/sibwithsei/GreetingService?wsdl`.

Run the `wsimport` command on the deployed WSDL URL by executing the following command:

```
D:\sibwithsei>wsimport -p client -keep http://localhost:8080/sibwithsei/Greeting
Service?wsdl
```

Verify all the generated classes and Java source files in the `d:\sibwithsei\client` folder.

Next, let's create a Web service client to invoke the Web service endpoint and its `dispGreeting()` method. Listing 19.28 shows the source code of the Web service client file named as `ClientApp.java` (you can find the `ClientApp.java` file in the `code/JavaEE/Chapter19/sibwithsei` folder in the CD):

Listing 19.28: Showing the Code for the ClientApp.java File

```
package client;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.lang.Exception;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
public class ClientApp
{
  public static void main(String args[])
  {
  try
  {
  URL wsdlURL = null;
  wsdlURL= new URL("http://localhost:8080/sibwithsei/GreetingService?wsdl");
  InputStream is = (InputStream) wsdlURL.getContent();
  Transformer t = TransformerFactory.newInstance().newTransformer();
  t.setOutputProperty(OutputKeys.INDENT,"yes");
  t.setOutputProperty(OutputKeys.METHOD,"xml");
  t.setOutputProperty(OutputKeys.MEDIA_TYPE,"text/xml");
  t.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION,"yes");
  t.transform(new StreamSource(is), new StreamResult(System.out));
  System.out.println();
  QName srvcQName = new QName("http://src/", "GreetingService");
  QName portQName =    new QName("http://src/", "GreetingPort");
  Service srvc = Service.create(wsdlURL, srvcQName);
  GreetingInf port = (GreetingInf) srvc.getPort(portQName, GreetingInf.class);
  String result = port.dispGreeting("Java Programmer");
  System.out.println(result);
  }
  catch (Exception e)
  {
        System.out.println(e);
  }
  }
}
```

Listing 19.28 is same as Listing 19.25; however, it creates a URL instance with the URL of the generated WSDL document, `http://localhost:8080/sibwithsei/GreetingService?wsdl`. Listing 19.28 uses the `Service.create()` method to configure the `Service` instance with the URL of WSDL document and the QName of the `wsdl:service` element. After creating the service instance, it invokes the `Service.getPort()` method, which accepts the Qname of `wsdl:portType` and `GreetingInf.class` as parameters. The `getPort()` method returns the implementation instance of the `GreetingPort` class.

Compile the `ClientApp.java` file by using the `javac` command, and place the generated `ClientApp.class` file in the `D:\sibwithsei\client` folder.

Figure 19.21 shows the output of the compilation and execution of the `ClientApp.java` file:

Figure 19.21: Showing the Output of the sibwithsei Web Service Application

Figure 19.21 displays the generated WSDL document, GreetingService.wsdl, and the output string Welcome: Java Programmer of the dispGreeting() method.

Deployment with Deployment Descriptor

The JAX-WS specification makes the use of the `webservices.xml` Deployment Descriptor optional, as annotations specify the information required to deploy a Web service. However, you need to use the `webservices.xml` Deployment Descriptor when you are either overriding or not using annotations in the source code of a Web application.

Let's look at an example to use a Deployment Descriptor to deploy a Web service. Let's create the `WelcomeHandler` class to extract the environment information from the `web.xml` file. The `WelcomeHandler` class accesses env-entrys of the `port` component using either the JNDI lookup or the `@Resource` annotation. Listing 19.29 shows the source code of the `WelcomeHandler.java` file (you can find the `WelcomeHandler.java` file in the `code/JavaEE/Chapter19/descwebservice/src` folder in the CD):

Listing 19.29: Showing the Code for the WelcomeHandler.java File

```
package src;
import java.util.Set;
import javax.annotation.Resource;
import javax.xml.namespace.QName;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;
public class WelcomeHandler implements SOAPHandler<SOAPMessageContext> {
  public static final String APPEND_STRING = "src.WelcomeHandler.appendStrg";
  @Resource(name="appendedString")
  String injectedString = "undefined";
  public boolean handleMessage(SOAPMessageContext context) {
  if ( ((Boolean)context.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY)).
  booleanValue() ) return true;
  try {
        context.put(APPEND_STRING, injectedString);
        context.setScope(APPEND_STRING, MessageContext.Scope.APPLICATION);
        System.out.println("WelcomeHandler has appendedString = " + injectedString);
        return true;
  } catch (Exception e) {
        e.printStackTrace();
        throw new WebServiceException(e);
  }
  }
  public Set<QName> getHeaders() {
        return null;
  }
  public boolean handleFault(SOAPMessageContext context) {
        return true;
  }
  public void close(MessageContext context) {}
}
```

Create the `SecondHandler.java` file to allow the `webservices.xml` Deployment Descriptor to configure the associated SIB class, as shown in Listing 19.30 (you can find the `SecondHandler.java` file in the `code/JavaEE/Chapter19/descwebservice/src` folder in the CD):

Listing 19.30: Showing the Code for the SecondHandler.java File

```
package src;
import java.util.Set;
import javax.annotation.Resource;
import javax.xml.namespace.QName;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;
public class SecondHandler implements SOAPHandler<SOAPMessageContext> {
  public static final String APPEND_STRING = "src.WelcomeHandler.appendStrg";
  @Resource(name="appendedString")
  String injectedString = "undefined";
  public boolean handleMessage(SOAPMessageContext context) {
  if ( ((Boolean)context.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY)).
  booleanValue() ) return true;
  try {
        context.put(APPEND_STRING, "NEW !!! "+injectedString);
        context.setScope(APPEND_STRING, MessageContext.Scope.APPLICATION);
        System.out.println("The second handler has appended this: " + injectedString);
        return true;
  } catch (Exception e) {
        e.printStackTrace();
        throw new WebServiceException(e);
  }
  }
```

```
    public Set<QName> getHeaders() {
          return null;
    }
    public boolean handleFault(SOAPMessageContext context) {
          return true;
    }
    public void close(MessageContext context) {}
}
```

Create the `Handler` chain configuration file, required to configure handlers, as shown in Listing 19.31 (you can find the `handler.xml` file in the `code/JavaEE/Chapter19/descwebservice` folder in the CD):

Listing 19.31: Showing the Code for the handler.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns:jws="http://java.sun.com/xml/ns/javaee">
  <handler-chain>
        <handler>
              <handler-class>src.WelcomeHandler</handler-class>
        </handler>
  </handler-chain>
</handler-chains>
```

Create the SIB class `Greeting.java` that implements the Web service, as shown in Listing 19.32 (you can find the `Greeting.java` file in the `code/JavaEE/Chapter19/descwebservice/src` folder in the CD):

Listing 19.32: Showing the Code for the Greeting.java File

```
package src;
import javax.annotation.Resource;
import javax.jws.HandlerChain;
import javax.jws.WebService;
import javax.xml.ws.WebServiceContext;
@HandlerChain(file="handler.xml")
@WebService
public class Greeting {
  @Resource
  WebServiceContext websrvccntxt;
  public String dispGreeting(String s) {
  String appStr =
  (String) websrvccntxt.getMessageContext().get(WelcomeHandler.APPEND_STRING);
  return "Welcome: " + s + "[appended by handler: " + appStr + "]";
  }
}
```

To deploy the Servlet based Web service, create the `web.xml` Deployment Descriptor, as shown in Listing 19.33 (you can find the `web.xml` file in the `code/JavaEE/Chapter19/descwebservice/WEB-INF` folder in the CD):

Listing 19.33: Showing the Code for the web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:j2ee="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.5"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <servlet>
        <servlet-name>Greeting</servlet-name>
        <servlet-class>src.Greeting</servlet-class>
        <load-on-startup>0</load-on-startup>
  </servlet>
  <servlet-mapping>
        <servlet-name>Greeting</servlet-name>
        <url-pattern>/my-pattern</url-pattern>
  </servlet-mapping>
```

```
    <env-entry>
          <env-entry-name>appendedString</env-entry-name>
          <env-entry-type>java.lang.String</env-entry-type>
          <env-entry-value>ABCDEFGHI...</env-entry-value>
    </env-entry>
</web-app>
```

The Web service requires GlassFish-specific `sun-web.xml` descriptor to specify the context root of the application. Listing 19.34 shows the code for the `sun-web.xml` descriptor file (you can find the `sun-web.xml` file in the `code/JavaEE/Chapter19/descwebservice/WEB-INF` folder in the CD):

Listing 19.34: Showing the Code for the sun-web.xml File

```
<sun-web-app>
<context-root>descwebservice</context-root>
</sun-web-app>
```

Listing 19.35 shows the generated `webservices.xml` descriptor (you can find the `webservices.xml` file in the `code/JavaEE/Chapter19/descwebservice/WEB-INF` folder in the CD):

Listing 19.35: Showing the Code for the webservices.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<webservices xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.2"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://www.ibm.com/webservices/xsd/javaee_web_services_1_2.xsd">
  <webservice-description>
  <webservice-description-name>GreetingService</webservice-description-name>
  <port-component>
  <port-component-name>Greeting</port-component-name>
  <wsdl-service xmlns:ns1="http://src/">ns1:GreetingService</wsdl-service>
  <wsdl-port xmlns:ns1="http://src/">ns1:GreetingPort</wsdl-port>
  <service-impl-bean>
        <servlet-link>Greeting</servlet-link>
  </service-impl-bean>
  <handler-chains>
        <handler-chain>
              <handler>
                    <handler-name>handler</handler-name>
                    <handler-class>src.SecondHandler</handler-class>
              </handler>
        </handler-chain>
  </handler-chains>
  </port-component>
  </webservice-description>
</webservices>
```

Compile the `WelcomeHandler` and `SecondHandler` Java classes from the Command Prompt using the following commands:

```
D:\descwebservice>javac -d . src/WelcomeHandler.java
D:\descwebservice>javac -d . src/SecondHandler.java
```

Compile the `Greeting.java` class using the following command:

```
D:\descwebservice>javac -d . src/Greeting.java
```

Run the `apt` command and generate the required wrapper classes. Make sure that the `d:\descwebservice\generated` folder is created before the execution of the following command:

```
D:\descwebservice>apt -d generated src/Greeting.java
```

To deploy the Web service, create another folder `c:\descwebservice`. The directory structure of the `c:\descwebservice`folder is shown in Figure 19.22:

Figure 19.22: Showing the Directory Structure and Packaging of the descwebservice Application

Create the required WAR file by using the command shown in Figure 19.22. Copy the `descwebservice.war` file in the `autodeploy` directory of the domain of the Glassfish V3 application server. GlassFish server automatically generates WSDL and internal descriptors. WSDL document is published on the URL `http://localhost:8080/descwebservice/my-pattern?wsdl`.

Run `wsimport` command on the deployed WSDL URL, as shown in the following command:

```
D:\descwebservice>wsimport -p client -keep http://localhost:8080/descwebservice/
my-pattern?wsdl
```

Verify all the generated classes and the Java source files in the `d:\descwebservice\client` folder.

Let's create a Web service client to invoke the Web service endpoint and its `dispGreeting()` method. Listing 19.36 shows the source code of the Web service client:

Listing 19.36: Showing the Code for the ClientApp.java File

```
package client;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.lang.Exception;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
public class ClientApp {
  public static void main(String args[]){
  try{
  URL wsdlURL = null;
  wsdlURL= new URL("http://localhost:8080/descwebservice/my-pattern?wsdl");
  InputStream is = (InputStream) wsdlURL.getContent();
  Transformer t = TransformerFactory.newInstance().newTransformer();
  t.setOutputProperty(OutputKeys.INDENT,"yes");
```

```
        t.setOutputProperty(OutputKeys.METHOD,"xml");
        t.setOutputProperty(OutputKeys.MEDIA_TYPE,"text/xml");
        t.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION,"yes");
        t.transform(new StreamSource(is), new StreamResult(System.out));
        System.out.println();
        QName srvcQName = new QName("http://src/", "GreetingService");
        QName portQName =      new QName("http://src/", "GreetingPort");
        Service srvc = Service.create(wsdlURL, srvcQName);
        Greeting port = (Greeting) srvc.getPort(portQName, Greeting.class);
        String result = port.dispGreeting("Java Programmer");
        System.out.println(result);
        }
        catch (Exception e){
              System.out.println(e);
        }
        }
    }
```

Listing 19.36 is same as Listing 19.25 but it creates a URL instance with the generated URL of the WSDL, `http://localhost:8080/descwebservice/my-pattern?wsdl`.

Compile the `ClientApp.java` file by using the `javac` command and place the generated `ClientApp.class` file in the `D:\descwebservice\client` folder.

Figure 19.23 shows the output of the compilation and execution of the `ClientApp.java` file:

Figure 19.23: Showing the Output of the descwebservice Application

Implementing the SAAJ Specification

In this section, we create a simple SAAJ-based application. This application sends a simple SOAP Message using the `SendingServlet` servlet class to the specified destination, which is another servlet class, `ReceivingServlet`. The `ReceivingServlet` servlet class gives a reply in the form of another SOAP message.

Listing 19.37 shows the code for the `SendingServlet` servlet class (you can find the `SendingServlet.java` file in the `code/JavaEE/Chapter19/saaj` folder in the CD):

Listing 19.37: Showing the Code for the SendingServlet.java File

```
import java.io.*;
import java.net.URL;
import java.util.Properties;
import javax.activation.DataHandler;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.xml.soap.*;
public class SendingServlet extends HttpServlet {
 String dest = null;
 private SOAPConnection con;
 public void init(ServletConfig servletConfig) throws ServletException {
        super.init( servletConfig );
        try {
        SOAPConnectionFactory scf = SOAPConnectionFactory.newInstance();
        con = scf.createConnection();
        } catch(Exception e) {
                System.out.println(  "Unable to open a SOAPConnection"+ e);
        }
 }
 public void doGet(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException {
        String SerResp ="<html><H4>";
        try {
                MessageFactory msgfact = MessageFactory.newInstance();
                SOAPMessage msg = msgfact.createMessage();
                SOAPPart sp = msg.getSOAPPart();
                SOAPEnvelope env = sp.getEnvelope();
                SOAPHeader head = env.getHeader();
                SOAPBody body = env.getBody();
                SOAPBodyElement sbe
                = body.addBodyElement(env.createName("GetLastTradePrice",
                "ztrade",
                "http://wombat.ztrade.com"));
                sbe.addChildElement(env.createName("symbol",
                "ztrade",
                "http://wombat.ztrade.com")).addTextNode("SUNW");
                StringBuffer sbUrl=new StringBuffer();
                sbUrl.append(req.getScheme()).append("://").
                append(req.getServerName());
                sbUrl.append( ":" ).append( req.getServerPort() ).append(
                req.getContextPath() );
                String reqBase=sbUrl.toString();
                URL url = new URL(reqBase + "/index.html");
                AttachmentPart ap = msg.createAttachmentPart(new DataHandler(url));
                ap.setContentType("text/html");
                msg.addAttachmentPart(ap);
                if(dest==null) {
                        dest=reqBase + "/ReceivingServlet";
```

```
                }
                URL urlEP = new URL(dest);
                SerResp += "Please see sentmsg.txt file which contains recently sent
                message and ";
                File f1 = new File("D:\\saaj\\sentmsg.txt");
                FileOutputStream sentFile = new FileOutputStream(f1);
                msg.writeTo(sentFile);
                sentFile.close();
                SOAPMessage reply = con.call(msg, urlEP);

                if (reply != null) {
                File f2 = new File("D:\\saaj\\replymsg.txt");
                FileOutputStream replyFile = new FileOutputStream(f2);
                reply.writeTo(replyFile);
                replyFile.close();
                SerResp += " see received reply in replymsg.txt file.</H4></html>";
                } else {
                        System.out.println("No reply");
                        SerResp += " no reply was received. </H4></html>";
                }
        } catch(Throwable t) {
        System.out.println("Error in constructing or sending message "+t);
                SerResp += " There was an error " +
                "in constructing or sending message. </H4></html>";
        }
        try {
                OutputStream os = resp.getOutputStream();
                os.write(SerResp.getBytes());
                os.flush();
                os.close();
        } catch (IOException e) {
        System.out.println("Error in outputting servlet response "+e);
        }
 }
}
```

In Listing 19.37, the `doGet()` method creates an instance of `SOAPMessage` from message factory. Empty `SOAPPart`, `SOAPEnvelope`, and `SOAPBody` parts are received in sequence. The SOAP body element, `GetLastTradePrize`, is added to the SOAP body. The request URL to send this message is prepared. An attachment part is created from a URL and added to the SOAP message. The URL for recipient of this message is created. The entire SOAP message is written to the `sentmsg.txt` file. The SOAP message is sent to the recipient by using the `call()` method of the `con` object.

Listing 19.38 shows the code for the `ReceivingServlet.java` file (you can find the `ReceivingServlet.java` file in the `code/JavaEE/Chapter19/saaj` folder in the CD):

Listing 19.38: Showing the Code for the ReceivingServlet.java File

```
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPMessage;
import com.sun.xml.messaging.soap.server.SAAJServlet;
public class ReceivingServlet extends SAAJServlet
{
        public SOAPMessage onMessage(SOAPMessage message) {
        System.out.println("The onMessage() method called in the
                        receiving servlet");
        try {
                System.out.println("The message is as follows: ");
                message.writeTo(System.out);
                SOAPMessage messg = msgFactory.createMessage();
                SOAPEnvelope envlp = messg.getSOAPPart().getEnvelope();
```

```
                envlp.getBody()
                .addChildElement(envlp.createName("MsgResponse"))
                .addTextNode("This is a response");
                return messg;
        } catch(Exception excp) {
        System.out.println("Error in processing or replying to a message"+
                excp);
        return null;
        }
    }
}
```

The onMessage() method of Listing 19.38 is executed when it receives a SOAP message.

The `index.html` file acts as a home page of this application. It contains a hyperlink, `here`, and a simple text description about the application. Listing 19.39 shows the code for the `index.html` file (you can find the `index.html` file in the `code/JavaEE/Chapter19/saaj` folder in the CD):

Listing 19.39: Showing the Code for the index.html File

```
<html>
  <body>
        This is a simple example of a SAAJ message exchange.
        <p> Click <a href="SendingServlet">here</a> to send the message
  </body>
</html>
```

Clicking the `here` hyperlink sends a request to the `SendingServlet` servlet, class which sends a message to the `ReceivingServlet` servlet class.

Each servlet-based Web service needs the `web.xml` Deployment Descriptor for deployment. Listing 19.40 shows the `web.xml` Deployment Descriptor of this application (you can find the `web.xml` file in the `code/JavaEE/Chapter19/saaj` folder in the CD):

Listing 19.40: Showing the Code for the web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet>
        <servlet-name>SendingServlet</servlet-name>
        <servlet-class>SendingServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet>
        <servlet-name>ReceivingServlet</servlet-name>
        <servlet-class>ReceivingServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
  </servlet>
  <servlet-mapping>
        <servlet-name>SendingServlet</servlet-name>
        <url-pattern>/SendingServlet</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
        <servlet-name>ReceivingServlet</servlet-name>
        <url-pattern>/ReceivingServlet</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
        <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Compile the `SendingServlet` and `ReceivingServlet` classes using the `javac` command. Create the `D:\saaj` folder whose directory structure is shown in Figure 19.24:

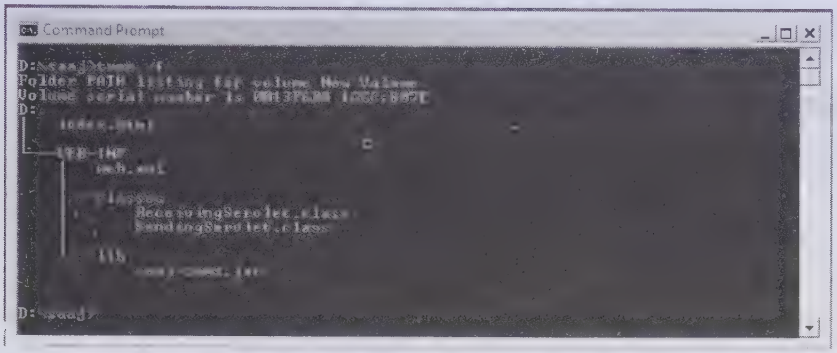


Figure 19.24: Showing the Directory Structure of the sibwithsei Web Service

Package the `D:\saaj` folder in a .WAR file, `saaj.war`. Copy this WAR file in the `C:\Program Files\glassfish\domains\domain1\autodeploy` folder. Start the GlassFish application server. Type `http://localhost:8080/saaj/` on the address bar of the browser and press the Enter key on the keyboard. You can see the `index.html` page, as shown in Figure19.25:

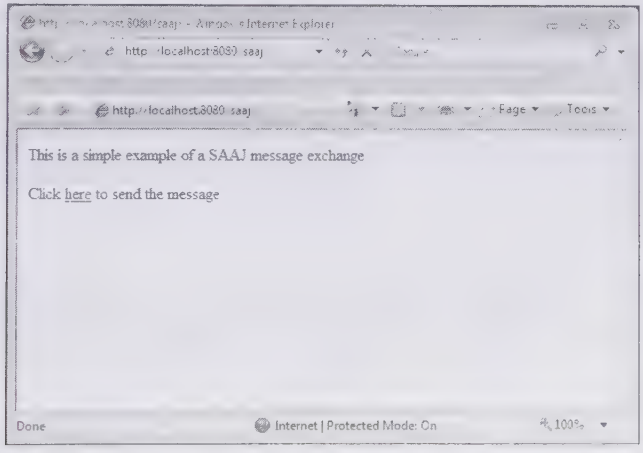


Figure 19.25: Showing the Home Page of the saaj Web Application

When you click the `here` hyperlink, you can see the response generated by the SendingServlet servlet class, as shown in Figure 19.26:

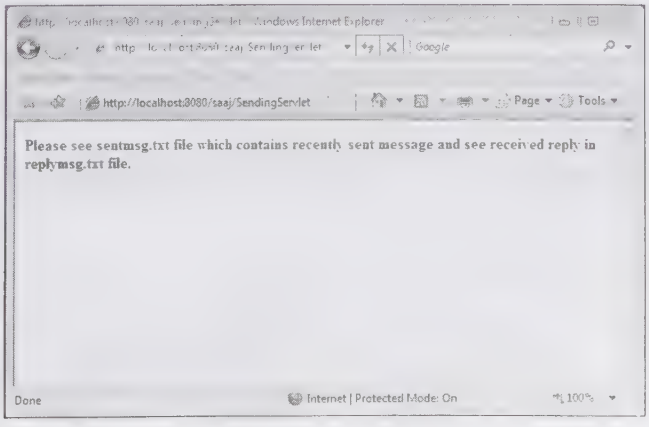


Figure 19.26: Response of the SendingServlet Servlet Class

Figure 19.26 indicates that SOAP message has been successfully sent. It also specifies the locations of the request and response SOAP messages.

Implementing the JAXR Specification

The section helps you in developing a JAXR client that can perform queries and modifications in the UDDI registry. In this application, the Glassfish V3 application server has been used, which provides a JAXR implementation. This server provides JAXR implementation in the form of a resource adapter.

To query and modify the entries in the UDDI registry, you first need to set up a connection.

Setting up a Connection

To set up a connection, you now need to create an instance of the connection factory. Some JAXR implementations provide one or more preconfigured connection factories. JAXR clients implemented as the Java EE component can access these factories using a resource injection. You need to specify the JNDI name of the connector resource (i.e., /JAXR) to access the preconfigured connection factory.

A JAXR client needs to specify the URLs of registries to be accessed. This is done by setting the properties of the `Properties` instance. JAXR specification defines some standard connection properties on a JAXR connection, as listed in Table 19.11:

Table 19.11: Standard JAXR Connection Properties

Property	Description	Data Type	Default value
javax.xml.registry.queryManagerURL	Determines the URL of the query manager service that is available within the target registry provider	String	None
javax.xml.registry.lifeCycleManagerURL	Determines the URL of the lifecycle manager service that is available within the target registry provider	String	Same as the value specified for queryManagerURL
javax.xml.registry.semanticEquivalences	Determines the semantic equivalence between any two concepts	String	None
javax.xml.registry.security.authenticationMethod	Indicates the provider about the authentication method that is to be applied for authentication with the registry provider	String	None
javax.xml.registry.uddi.maxRows	Indicates the maximum number of rows which a find operation can return.	Integer	None
javax.xml.registry.postalAddressScheme	Represents the ID of a ClassificationScheme that is to be used as a default postal address scheme	String	None

Let us look at connection properties that you can set in the Sun Java server, as specified in Table 19.12:

Table 19.12: Sun's JAXR Implementation Connection Properties

Property	Name	Data Type	Default Value
com.sun.xml.registry.http.proxyHost	Sets the HTTP proxy host that is used to access external registry	String	None
com.sun.xml.registry.http.proxyPort	Sets the HTTP proxy port that is used to access external registry, usually 8080	String	None
com.sun.xml.registry.https.proxyHost	Sets the HTTPS proxy host that is used to access external registry	String	Same as the value of HTTP proxy host

Table 19.12: Sun's JAXR Implementation Connection Properties

Property	Name	Data Type	Default Value
com.sun.xml.registry.https .proxyPort	Sets the HTTPS proxy port that is used to access external registry	String	Same as the value of HTTP proxy port
com.sun.xml.registry.http. proxyUserName	Sets the proxy host's user name for HTTP proxy authentication	String	None
com.sun.xml.registry.http. proxyPassword	Sets the proxy host's password for HTTP proxy authentication	String	None
com.sun.xml.registry.useC ache	Instructs the JAXR implementation to first search the registry objects in cache and then in the registry in case they are not found in the cache	Boolean (passed as String)	True
com.sun.xml.registry.user TaxonomyFilenames	Adds user-defined taxonomy structures to the JAXR provider	String	None

The following code snippet shows how to access a connection factory:

```
import javax.annotation.Resource;.*;
import javax.xml.registry.ConnectionFactory;
...
@Resource(mappedName="eis/JAXR")
  public ConnectionFactory connfact;
```

In a standalone JAXR client, use the following code snippet to access a connection factory:

```
import javax.xml.registry.ConnectionFactory;
 ...
ConnectionFactory connFact =ConnectionFactory.newInstance();
```

In order to set up a connection, a client needs to first create a set of properties to specify the details about the URLs of the registries to be accessed. The following code snippet shows how to set up the properties, by providing the URL referring to the query service for a hypothetical registry:

```
Properties propty = new Properties();
Propty.setProperty("javax.xml.registery.queryManagerURL",
                   "http://localhost:8080/RegisteryServer/");
...
```

If you are accessing an external registry, you also need to specify the proxy host and port for a network on which you are running your JAXR client program. The following code snippet shows how to set the connection properties on a JAXR connection for accessing external registries:

```
propty.setProperty("com.sun.xml.registry.http.proxyHost",
                   "yourhost.yourdomain");
propty.setProperty("com.sun.xml.registry.http.proxyPort",
                   "8080");
propty.setProperty("com.sun.xml.registry.https.proxyHost",
                   "yourhost.yourdomain");
propty.setProperty("com.sun.xml.registry.https.proxyPort",
                   "8080");
connFact.setProperties(propty);
Connection con = connFact.createConnection();
```

The last two lines in the preceding code snippet set the properties for the connection factory and create the connection.

Querying a Registry

After getting the connection instance, let us obtain a `RegistryService` object and its interfaces to perform queries on the registry, as shown in the following code snippet:

```
RegistryService <Registry service object name>= <connection
                 object>.getRegistryService();
BusinessQueryManager <object name>= <registry Service Object
```

```
                    name>.getBusinessQueryManager();
BusinessLifeCycleManager <object name>=<registry Service Object
                    name>.getBusinessLifeCycleManager();
```

You need to get only the `BusinessQueryManager` instance for performing simple queries on the registry. In case of complex queries, the `BusinessLifeCycleManager` instance is also required.

The `BusinessQueryManager` interface provides several find methods to search data on the basis of JAXR information model. Important methods used for search purpose are listed as follows:

- **findOrganizations**—Returns a `BulkResponse` object that represents a collection of organizations according to a specified criteria, such as a specified name pattern or a specific classification scheme.
- **findServices**—Returns a `BulkResponse` object that represents a collection of services which a particular organization offers.
- **findServiceBindings**—Returns a `BulkResponse` object that represents a set of service bindings supported by a particular service.

The `JAXRQuery` tool queries a registry on the basis of the name of organizations. There are three classification systems used for classifying registries:

- **NAICS**—Stands for North American Industry Classification System. You can visit the `http://www.census.gov/epcd/www/naics.html` link to get more information on this system.
- **UNSPSC**—Stands for Universal Standard Products and Services Classification. You can visit the `http://www.unspsc.org/` link to get more information on this system.
- **ISO**—Stands for International Organization for Standardization. This organization specifies a 3166 country codes classification system for accessing registries. You can visit the `http://www.iso.org/iso/country_codes` link to get more information on this system.

Finding Organizations by Name

Let us find organizations by name, by using the `findOrganizations()` method. The following code snippet shows the syntax of this method:

```
public BulkResponse findOrganizations(Collection findQualifiers,
                                      Collection namePatterns,
                                      Collection classifications,
                                      Collection specs,
                                      Collection extIds,
                                      Collection extLinks)
                               throws JAXRException
```

The `findOrganizations()` method returns a collection of organization objects after performing logical AND operation on the criteria specified by the parameters. Some parameters can take null values. Some of the parameters required by the `findOrganizations()` method are described as follows:

- **findQualifiers**—Represents a parameter that is a collection of the find qualifiers defined in the `FindQualifier` interface, which has several constants that affect the search of the `findOrganizations()` method. Some of the constants defined in this interface are EXACT_NAME_MATCH, SORT_BY_DATE_ASC and SORT_BY_NAME_ASC.
- **namePatterns**—Specifies a parameter that is a collection of either the String or the LocalizedString object. Each String or LocalizedString value contains a wildcard name matching the pattern used in the SQL-92 LIKE operator.

Finding Organizations by Classification

Let us now find the organizations by classification. To perform this task, you first need to create the `Classification` objects of different classification schemes. Then, you need to instantiate the `Collection` instance containing these `Classification` objects. The following code snippet shows the syntax to search all the organizations corresponding to the classification scheme designed by NAICS:

```
String uuid_naics =
"uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2";
ClassificationScheme <object name>=
  (ClassificationScheme) <business query manager object
```

```
    name>.getRegistryObject(uuid_naics,
    LifeCycleManager.CLASSIFICATION_SCHEME);
InternationalString <object name> = <business life cycle manager object
    name>.createInternationalString(
    <Enter Search String here in quotes>));
String <Variable name>= "Enter corresponding value of search string";
Classification <object name> =
    <business life cycle manager object name>.createClassification(<enter
    classification scheme object name>, <Enter international string object
    name, <Enter Search value>);
Collection<Classification><Collection of Classification object name>=
    new ArrayList<Classification>();
<Collection of Classification object names>.add(<Classification object name>);
BulkResponse <Object name>= <business query manager object
    name>.findOrganizations(null, null,
    <Collection of Classification object name>, null, null, null);
Collection <object name>= <BulkResponse Object name>.getCollection();
```

Let's now use classifications to search organizations whose services are based on some technical specifications. These technical specifications are in the form of WSDL documents. To perform this task, you need to use the `Concept` class that holds information for a single specification. After finding the specification concepts, we need to search for the organizations that use these concepts, as shown in following code snippet:

```
String <variable name> = "uddi-org:types";
ClassificationScheme <Object Name>=
    <business query manager object name>.findClassificationSchemeByName(null,
    <Enter variable name>);
Classification <object name>=
    <business life cycle manager object name>.createClassification(uddiOrgTypes,
    "wsdlSpec","wsdlSpec");
Collection<Classification><collection of classification object name> =
    new ArrayList<Classification>();
<collection of classification object name>.add(<Enter
    classification object name>);
// Find concepts
BulkResponse <Object name>=<business query manager object
    name>.findConcepts(null, null,
    <collection of classification object names>, null, null);
```

Manipulating Registry Objects

You must ensure that you have the required authorization before manipulating the registry objects. To perform submit, update, and delete operations, use the `BusinessLifeCycleManager` interface.

If you have the authorization, you need to send authorization credentials, such as user name and password, to the registry before manipulating the registry objects. The following code snippet shows how to send the authorization credentials:

```
String <user name variable> = <Enter value in Quotes>;
String <password variable> = <Enter value in Quotes>;
// Get authorization from the registry
PasswordAuthentication <object name> =
    new PasswordAuthentication(<enter password authentication name>,
    <password variable>.toCharArray());
HashSet<PasswordAuthentication><HashSet of PasswordAuthentication object name>=
    new HashSet<PasswordAuthentication>();
<HashSet of PasswordAuthentication object name>.add(<Enter PasswordAuthentication object
    name>);
con.setCredentials(<Enter HashSet  of PasswordAuthentication object name>);
```

In the preceding code snippet, con object is the Connection object. Let's now learn how to manipulate registry objects. You specifically learn about the following tasks for manipulating the registry objects:

- Add an organization to the registry
- Add classifications
- Add services and service bindings
- Publish the organization
- Delete entries from the registry

Adding an Organization to the Registry

The following code snippet creates an organization, adds it to the registry, and sets other details, such as its name, description, and primary contact:

```
// Create organization name and description
InternationalString <object name> =
  <business life cycle manager object name>.createInternationalString(<Enter
  name of service>);
Organization <object name> = <business life cycle manager object
  name>.createOrganization(<InternationalString object name>);
<InternationalString object name> = <business life cycle manager object
  name>.createInternationalString("Indo-Asian News Service (IANS) - formerly
  India Abroad News Service - was employedin 1986 to act as an
  information bridge between India and North America and chronicle their
  growing ethnic, business and cultural links.");
<Organization object name>.setDescription(<Enter InternationalString object
  name>);
User <User object name> = <business life cycle manager object
  name>.createUser();
PersonName <PersonName object name> = <business life cycle manager object
  name>.createPersonName(<Enter real name of person>);
<User object name>.setPersonName(<Enter PersonName object name>);
TelephoneNumber <Enter TelephoneNumber object name> = <business life cycle manager
  object name>.createTelephoneNumber();
<TelephoneNumber object name>.setNumber(<Enter phone number of the person>);
Collection<TelephoneNumber><Collection of TelephoneNumber object name> =
  new ArrayList<TelephoneNumber>();
<Collection of TelephoneNumber object names>.add(<Enter TelephoneNumber object
  name>);
<User object name>.setTelephoneNumbers(<Collection of TelephoneNumber object
  names>);
  EmailAddress <EmailAddress object name> =
  blcm.createEmailAddress(<enter actual email address of the person>);
Collection<EmailAddress><Collection of EmailAddress object name> =
  new ArrayList<EmailAddress>();
<Collection of EmailAddress object names>.add(<Enter EmailAddress object name>);
<User object name>.setEmailAddresses(<Collection of EmailAddress object names>);
<Organization object name>.setPrimaryContact(<Enter User object name>);
```

The preceding code snippet creates and adds an organization name and description in the registry. It uses the `InternationalizationString` object for setting the name and description of the organization.

Adding Classifications

Usually all organizations come under some classifications. Each classification is based on a classification scheme, which is defined by the JAXR specification. Let us associate a classification scheme to our organization, as shown in following code snippet:

```
// Set classification scheme to NAICS
ClassificationScheme <ClassificationScheme object> =
  <BusinessQueryManager object name>.findClassificationSchemeByName(null,
  "ntis-gov:naics:1997");
// Create and add classification
InternationalString <InternationalString object name> =
  <business life cycle manager object name>.createInternationalString(
  <enter search string in quotes>));
String <variable name>= <Enter value related to search string>;
Classification <classification object name> =
<business life cycle manager object name>.createClassification(<ClassificationScheme
  object name>,<Enter InternationalString object name>, <enter corresponding value of
  search string>);
Collection<Classification><collection of ClassificationScheme object name>
        =new ArrayList<Classification>();
<collection of ClassificationScheme object names>.add(<enter Classification object
  name>);
<Organization object name>.addClassifications(<collection of
        ClassificationScheme object name>);
```

The preceding code snippet uses the `findClassificationSchemeByName()` method to create the `ClassficationScheme` instance corresponding to the `NAICS` system.

Adding Services and Service Bindings to an Organization

We now need to add services and associated service bindings in the registry. Similar to an `Organization` object, a `Service` object has members, such as `Name`, `Description`, and `Key` objects. A `ServiceBinding` object also has `description`, access URI of service, and `link` that relates service binding with a technical specification.

Following code snippet shows how to add services and service bindings to an organization:

```
// Create services and service
Collection<Service><name of the collection of Service object name> = new
  ArrayList<Service>();
InternationalString <InternationalString object name> =
  <business life cycle manager object name>.createInternationalString(<Enter
  Your Service Name>);
Service <Service object name> = <business life cycle manager object
  name>.createService(<InternationalString object name>);
<InternationalString object name> = <business life cycle manager object
  name>.createInternationalString(<enter Your Service Description>);
<Service object name>.setDescription(<enter InternationalString object name>);
// Create service bindings
Collection<ServiceBinding><name of a Collection of ServiceBinding object name> =
  new ArrayList<ServiceBinding>();
ServiceBinding <ServiceBinding object name> = <business life cycle manager object
  name>.createServiceBinding();
<InternationalString object name> = <business life cycle manager object
  name>.createInternationalString(<enter Your Service Binding Description in quotes>);
<ServiceBinding object name>.setDescription(<Enter InternationalString object
  name>);
<ServiceBinding object name>.setValidateURI(false);
<ServiceBinding object name>.setAccessURI("http://ians.com:8080/ns/");
<name of a Collection of ServiceBinding object name>.add(<Enter ServiceBinding
  object name>);
<ServiceBinding object name>.addServiceBindings(<enter name of a Collection of
  ServiceBinding objects>);
<name of the collection of Service objects>.add(<enter ServiceObject name>);
<Organization object name>.addServices(<enter name of the collection of Service
  objects>);
```

The preceding code snippet creates a collection of `Service` objects. It also creates a `ServiceBinding` instance and adds it to the collection of the `ServiceBinding` objects.

Publishing an Organization

Our organization object is now ready to be published to a registry. The JAXR client uses the `saveOrganizations()` method to submit or update data of organizations to a registry. Publishing an organization implies that the data of the organization has been made public. After successful execution of the `saveOrganizations()` method, the registry assigns a unique key to the organization. Following code snippet shows how an organization can be published with a registry:

```
Collection<Organization><name of a collection of Organization object name > =
  new ArrayList<Organization>();
<name of a collection of Organization object name>.add(<enter Organization
      object name>);
BulkResponse<BulkResponse object name> = <business life cycle manager object
  name>.saveOrganizations(<name of a collection of Organization object names
  >);
Collection <name of a collection of Exception objects> = <enter BulkResponse
  object name>.getException();
if (<name of a collection of Exception object names> == null) {
  System.out.println("Organization saved to registry");
  Collection <collection of Key instances> = <enter BulkResponse object
      name>.getCollection();
  Iterator <Iterator Object name> = <collection of Key instances>.iterator();
  if (<Iterator Object name>.hasNext()) {
```

```
            Key <Key object name> = (Key)<Iterator Object name>.next();
            String <variable name> = <enter Key object name>.getId();
            System.out.println("Organization key is " + <variable name>);
        }
    }
```

The preceding code snippet publishes an organization to the registry by invoking the `saveOrganizations()` method.

Deleting Data from the Registry

You can also delete data published to a registry when the data becomes obsolete. You can delete organizations, services, service bindings, and concepts. To do this, the BusinessLifeCycleManager interface provides certain methods, such as `deleteOrganizations()`, `deleteServices()`, `deleteServiceBindings()`, and `deleteConcepts()`.

The following code snippet shows how an organization can be deleted from the registry:

```
String <variable name> = <Key object name>.getId();
System.out.println("Deleting organization with id " + <String variable name>);
Collection<Key><name of a collection of Key instances>= new ArrayList<Key>();
<name of a collection of Key instances>.add(<enter Key object name>);
BulkResponse<BulkResponse object name> = <business query life cycle manager
  object>.deleteOrganizations(<name of a collection of Key instances>);
Collection <name of a collection of Exception objects > = res.getException();
if (<collection of Exception object names> == null) {
  System.out.println("Organization deleted");
Collection <name of a collection of Key objects> = (<collection of Exception
  objects>.getCollection();
  Iterator <Iterator object> = <collection of Key objects>.iterator();
  Key <Key object name> = null;
  if (<Iterator object>.hasNext()) {
        <Key object name> = (Key) <Iterator object>.next();
        <variable name> = <Key object name>.getId();
        System.out.println("Organization key was " + <variable name>);
  }
}
```

The preceding code snippet uses the `deleteOrganizations()` method that takes a collection of keys to delete an organization from the registry.

Implementing the StAX Specification

You can use the StAX specification to read and write data in XML streams and documents. Let's learn about these in detail next.

Reading XML Streams

You can use the `XMLStreamReader` and `XMLEventReader` interfaces to read XML streams.

Using the XMLStreamReader Interface

You can use the `XMLStreamReader` interface of StAX `cursor` API to read XML documents or streams in a forward direction only, one element at a time. You can perform the following operations using the `XMLStreamReader` interface:

- Retrieve the value of an attribute
- Read XML data
- Determine whether or not an element contains data
- Access a collection of attributes using indexes
- Access a collection of namespaces using indexes
- Access name and contents of the current event

The following code snippet shows some of the methods used to retrieve information about the namespaces and attributes of an XML stream:

```
int getAttributeCount();
String getAttributeNamespace(int index);
```

```
String getAttributeName(int index);
String getAttributePrefix(int index);
String getAttributeType(int index);
String getAttributeValue(int index);
String getAttributeValue(String namespaceUri,
                         String localName);
boolean isAttributeSpecified(int index);
```

You can access the namespaces of XML streams by using the methods shown in the following code snippet:

```
int getNamespaceCount();
String getNamespacePrefix(int index);
String getNamespaceURI(int index);
```

You can access individual elements of an XML stream by instantiating an input factory, creating a reader, and iterating over elements using the methods of the `XMLStreamReader` interface, as shown in the following code snippet:

```
XMLInputFactory xifact = XMLInputFactory.newInstance();
XMLStreamReader xsread = xifact.createXMLStreamReader( ... );
while(xsread.hasNext()) {
   xsread.next();
}
```

Using the XMLEventReader Interface

The `XMLEventReader` interface of the StAX event iterator API maps events in an XML stream to the allocated event objects. It provides four methods, next(), nextEvent(), hasNext(), and peek(), to iterate XML streams. The following code snippet shows the syntax of these methods of the `XMLEventReader` interface:

```
package javax.xml.stream;
import java.util.Iterator;
public interface XMLEventReader extends Iterator {
   public Object next();
   public XMLEvent nextEvent() throws XMLStreamException;
   public boolean hasNext();
   public XMLEvent peek() throws XMLStreamException;
   ...
}
```

In the preceding code snippet, the `next()` method of the `XMLEventReader` interface returns the next event in the stream. The next typed `XMLEvent` is returned by the `nextEvent()` method. The `hasNext()` method checks whether or not any event is left for processing in the stream. The `peek()` method returns the current event.

The following code snippet shows how to read and print all the events of a stream:

```
while(streamObj.hasNext()) {
   XMLEvent evt = streamObj.nextEvent();
   system.out.print(evt);
}
```

Writing XML Streams

It has already been discussed that the StAX API can both read and write XML streams or XML documents. The interfaces of the `cursor` and `event` iterator APIs for reading XML streams are similar, but the interfaces for writing XML streams are significantly different. The interfaces for writing to the XML stream are given as follows:

- XMLStreamWriter
- XMLEventWriter

Let's explore these interfaces for writing XML stream next.

Using the XMLStreamWriter Interface

The `XMLStreamWriter` interface of the StAX cursor API is used either to create a new XML stream or to write to an existing XML stream. The implementation class of the `XMLStreamWriter` interface performs operations defined in the `XMLOutputFactory` class.

Let's consider an example that creates an instance of the XMLOutputFactory class and XMLStreamWriter interface, and finally writes the XML output, as shown in the following code snippet:

```
XMLOutputFactory xofact= XMLOutputFactory.newInstance();
XMLStreamWriter xswrite= xofact.createXMLStreamWriter( ... );
xswrite.writeStartDocument();
xswrite.setPrefix("y","http://y");
xswrite.setDefaultNamespace("http://y");
xswrite.writeStartElement("http://y","w");
xswrite.writeAttribute("x","abc");
xswrite.writeNamespace("y","http://y");
xswrite.writeDefaultNamespace("http://y");
xswrite.setPrefix("z","http://y");
xswrite.writeEmptyElement("http://y","z");
xswrite.writeAttribute("http://y","fname","lname");
xswrite.writeNamespace("z","http://y");
xswrite.writeCharacters("Character Content");
xswrite.writeEndElement();
xswrite.flush();
```

In the preceding code snippet, the `writeCharacters()` method escapes special characters, such as &, <, and ". The `setPrefix()` method binds the prefixes passed as an argument to it. The `setDefaultNamespace()` method sets the default namespace. The `writeStartElement()` method adds the `StartElement` event to the XML stream.

Using the XMLEventWriter Interface

The `XMLEventWriter` interface of the StAX event iterator API is also used either to create a new XML stream or to write to an existing XML stream. The following code snippet shows the methods of the `XMLEventWriter` interface:

```
public interface XMLEventWriter {
   public void flush() throws XMLStreamException;
   public void close() throws XMLStreamException;
   public void add(XMLEvent e) throws XMLStreamException;
   // ... other methods not shown.
}
```

In the preceding code snippet, the `add()` method is used to add stream events represented by the `XMLEvent` parameter. Note that you cannot modify the event after adding it to an event writer instance.

Reading an XML File using the Cursor API

Let's create an application that parses an XML file, `ProductDetails.xml`, using the cursor API.

Listing 19.41 shows the source code of the `ProductDetails.xml` file (you can find the `ProductDetails.xml` file in the `code/JavaEE/Chapter19/cursor` folder in the CD):

Listing 19.41: Showing the Code for the ProductDetails.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<ProductDetails xmlns="http://www.kogentindia.com">
  <Product>
        <Name>LG DVD Player</Name>
        <Category>Entertainment</Category>
        <LaunchDate>June, 2008</LaunchDate>
        <DESC>You can play DVD, CD in latest formats </DESC>
        <Price>2050</Price>
  </Product>
  <Product>
        <Name>Dish TV </Name>
        <Category>Education</Category>
        <LaunchDate>May,2008</LaunchDate>
        <DESC>You can see programs on various air channels</DESC>
        <Price>2250</Price>
  </Product>
</ProductDetails>
```

The `ProductDetails.xml` file contains two `Product` elements, where each element contains information about the `Name`, `Category`, `LaunchDate`, `DESC`, and `Price` of a product.

Listing 19.42 parses the `ProductDetails.xml` file using the cursor API (you can find the `ParseUsingCursor.java` file in the `code/JavaEE/Chapter19/cursor` folder in the CD):

Listing 19.42: Showing the Code for the ParseUsingCursor.java File

```
import java.io.FileInputStream;
import javax.xml.namespace.QName;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.events.XMLEvent;
public class ParseUsingCursor {
  public static void main(String[] args) throws Exception {
        int cnt = 0;
        String XMLFName = "ProductDetails.xml";
        cnt = Integer.parseInt(args[0]);
        XMLInputFactory xmlinfact = null;
        try {
                xmlinfact = XMLInputFactory.newInstance();
                xmlinfact.setProperty(
                XMLInputFactory.IS_REPLACING_ENTITY_REFERENCES,
                Boolean.TRUE);
                xmlinfact.setProperty(
                XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES,
                Boolean.FALSE);
                xmlinfact.setProperty(XMLInputFactory.IS_COALESCING,
                Boolean.FALSE);
        } catch (Exception ex) {
                ex.printStackTrace();
        }
        System.out.println("");
        System.out.println("FACTORY: " + xmlinfact);
        System.out.println("");
        try {
                for (int i = 0; i < cnt; i++) {
                XMLStreamReader xsr = xmlinfact.createXMLStreamReader(
                        XMLFName,new FileInputStream(XMLFName));
                int evntcat = xsr.getEventType();
                //printEventType(eventType);
                dispStartDocument(xsr);
                //check if there are more events in the input stream
                while (xsr.hasNext()) {
                        evntcat = xsr.next();
                        //printEventType(eventType);
                        //these functions prints the information about the
                        // particular event by calling relevant function
                        dispStartElement(xsr);
                        dispEndElement(xsr);
                        dispText(xsr);
                        dispPIData(xsr);
                        dispComment(xsr);
                }
        }
        } catch (XMLStreamException e) {
                System.out.println(e.getMessage());
                if (e.getNestedException() != null) {
                        e.getNestedException()
                        .printStackTrace();
                }
        } catch (Exception e) {
                e.printStackTrace();
        }
  }
  public static final String getEventTypeString(int evntcat) {
        switch (evntcat) {
        case XMLEvent.START_ELEMENT:
        return "START_ELEMENT";
        case XMLEvent.END_ELEMENT:
        return "END_ELEMENT";
        case XMLEvent.PROCESSING_INSTRUCTION:
```

```
        return "PROCESSING_INSTRUCTION";
        case XMLEvent.CHARACTERS:
        return "CHARACTERS";
        case XMLEvent.COMMENT:
        return "COMMENT";
        case XMLEvent.START_DOCUMENT:
        return "START_DOCUMENT";
        case XMLEvent.END_DOCUMENT:
        return "END_DOCUMENT";
        case XMLEvent.ENTITY_REFERENCE:
        return "ENTITY_REFERENCE";
        case XMLEvent.ATTRIBUTE:
        return "ATTRIBUTE";
        case XMLEvent.DTD:
        return "DTD";
        case XMLEvent.CDATA:
        return "CDATA";
        case XMLEvent.SPACE:
        return "SPACE";
    }
    return "UNKNOWN_EVENT_TYPE , " + evntcat;
    }
    private static void dispEventType(int evntcat) {
        System.out.println(
        "EVENT TYPE(" + evntcat + ") = "
        + getEventTypeString(evntcat));
    }
    private static void dispStartDocument(XMLStreamReader xsr) {
        if (xsr.START_DOCUMENT == xsr.getEventType()) {
                System.out.println(
                "<?xml version=\"" + xsr.getVersion() + "\""
                + " encoding=\"" + xsr.getCharacterEncodingScheme() + "\""
                + "?>");
        }
    }
    private static void dispComment(XMLStreamReader xsr) {
        if (xsr.getEventType() == xsr.COMMENT) {
                System.out.print("<!--" + xsr.getText() + "-->");
        }
    }
    private static void dispText(XMLStreamReader xsr) {
        if (xsr.hasText()) {
                System.out.print(xsr.getText());
        }
    }
    private static void dispPIData(XMLStreamReader xsr) {
        if (xsr.getEventType() == XMLEvent.PROCESSING_INSTRUCTION) {
                System.out.print(
                "<?" + xsr.getPITarget() + " " + xsr.getPIData() + "?>");
        }
    }
    private static void dispStartElement(XMLStreamReader xsr) {
        if (xsr.isStartElement()) {
                System.out.print("<" + xsr.getName().toString());
                dispAttributes(xsr);
                System.out.print(">");
        }
    }
    private static void dispEndElement(XMLStreamReader xsr) {
        if (xsr.isEndElement()) {
                System.out.print("</" + xsr.getName().toString() + ">");
        }
    }
    private static void dispAttributes(XMLStreamReader xsr) {
        int cnt = xsr.getAttributeCount();
        if (cnt > 0) {
                for (int i = 0; i < cnt; i++) {
                        System.out.print(" ");
                        System.out.print(xsr.getAttributeName(i).toString());
                        System.out.print("=");
```

```
                        System.out.print("\"");
                        System.out.print(xsr.getAttributeValue(i));
                        System.out.print("\"");
                }
        }
        cnt = xsr.getNamespaceCount();
        if (cnt > 0) {
                for (int i = 0; i < cnt; i++) {
                        System.out.print(" ");
                        System.out.print("xmlns");
                        if (xsr.getNamespacePrefix(i) != null) {
                        System.out.print(":" + xsr.getNamespacePrefix(i));
                        }
                        System.out.print("=");
                        System.out.print("\"");
                        System.out.print(xsr.getNamespaceURI(i));
                        System.out.print("\"");
                }
        }
    }
}
```

Create a folder named cursor in the D: drive and place the `ParseUsingCursor.java` and `ProductDetails.xml` files inside this folder. Compile and run the`ParseUsingCursor.java` class as shown in Figure 19.27:

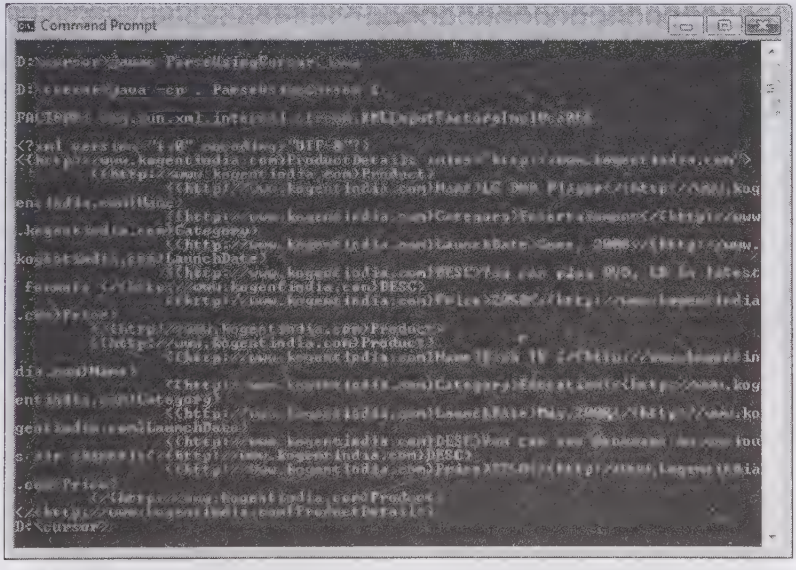


Figure 19.27: Showing the Compilation and Execution of the ParseUsingCursor.java File

Reading an XML File using the Event Iterator API

In this section, you learn to create a Java class, `ParseUsingEvent.java`, which parses the `ProductDetails.xml` file described in the previous section. This class creates an instance of the `XMLInputFactory` class. Listing 19.43 shows the source code of the`ParseUsingEvent.java` file (you can find the `ParseUsingEvent.java` file in the `code/JavaEE/Chapter19/EventIterator` folder in the CD):

Listing 19.43: Showing the Code for the ParseUsingEvent.java File

```
import java.io.FileInputStream;
import javax.xml.stream.*;
import javax.xml.stream.events.*;
import javax.xml.namespace.QName;
public class ParseUsingEvent {
    public static void main(String[] args) throws Exception {
```

```
        String XMLFName = "ProductDetails.xml";
        XMLInputFactory factory = XMLInputFactory.newInstance();
        System.out.println("FACTORY: " + factory);
        XMLEventReader xer = factory.createXMLEventReader(
        XMLFName,new FileInputStream(XMLFName));
        while (xer.hasNext()) {
                XMLEvent xet = xer.nextEvent();
                System.out.println(xet.toString());
        }
    }
    public static final String getEventTypeString(int eventcat) {
        switch (eventcat) {
                case XMLEvent.START_ELEMENT:
                return "START_ELEMENT";
                case XMLEvent.END_ELEMENT:
                return "END_ELEMENT";
                case XMLEvent.PROCESSING_INSTRUCTION:
                return "PROCESSING_INSTRUCTION";
                case XMLEvent.CHARACTERS:
                return "CHARACTERS";
                case XMLEvent.COMMENT:
                return "COMMENT";
                case XMLEvent.START_DOCUMENT:
                return "START_DOCUMENT";
                case XMLEvent.END_DOCUMENT:
                return "END_DOCUMENT";
                case XMLEvent.ENTITY_REFERENCE:
                return "ENTITY_REFERENCE";
                case XMLEvent.ATTRIBUTE:
                return "ATTRIBUTE";
                case XMLEvent.DTD:
                return "DTD";
                case XMLEvent.CDATA:
                return "CDATA";
                case XMLEvent.SPACE:
                return "SPACE";
        }
        return "UNKNOWN_EVENT_TYPE " + "," + eventcat;
    }
}
```

Create a folder named `EventIterator` in the D: drive and place the `ParseUsingEvent.java` and `ProductDetails.xml` files (described in the previous section) inside this folder. Compile and run the `ParseUsingEvent.java` class, as shown in Figure 19.28:

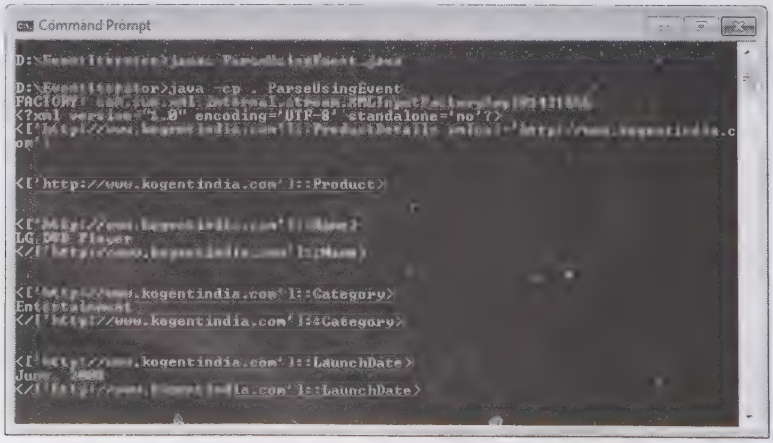


Figure 19.28: Showing the Output of the ParseUsingEvent Class

Writing an XML File using the Cursor API

In this section, you learn to create an XML file using the cursor API. This application that you are going to create in this section, has one Java class, `GenerateXMLUsingCusror`, which writes XML data to the output.xml file. Listing 19.44 shows the source code of the `GenerateXMLUsingCursor.java` file (you can find the `GenerateXMLUsingCursor.java` file in the `code/JavaEE/Chapter19/CursorWriter` folder in the CD):

Listing 19.44: Showing the Code for the GenerateXMLUsingCursor.java File

```
import java.io.File;
import java.io.FileOutputStream;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;
public class GenerateXMLUsingCursor {
   public static void main(String[] args) throws Exception
   {
         try
         {
         XMLOutputFactory xmlof = XMLOutputFactory.newInstance();
         XMLStreamWriter xmlsw = null;
         File file= new File("output.xml");
         xmlsw = xmlof.createXMLStreamWriter(new FileOutputStream(file));
         xmlsw.writeComment(
         "The namespace of price element is http://ecommerce.org/schema");
         xmlsw.writeStartDocument( );
         xmlsw.setPrefix("ecom", "http://ecommerce.org/schema");
         xmlsw.writeStartElement("http://ecommerce.org/schema", "x");
         xmlsw.writeNamespace("x", "http://ecommerce.org/schema");
         xmlsw.writeStartElement("http://ecommerce.org/schema", "price");
         xmlsw.writeAttribute("units", "Euro");
         xmlsw.writeCharacters("35.47");
         xmlsw.writeEndElement();
         xmlsw.writeEndElement();
         xmlsw.writeEndDocument();
         xmlsw.flush();
         xmlsw.close();
         } catch (Exception e) {
                System.err.println(
                "Exception occurred while running writer samples");
         }
   }
}
```

Create a folder named `CursorWriter` in the D: drive and place the `GenerateXMLUsingCursor.java` file inside this folder. Compile and run the `GenerateXMLUsingCusror.java` file, as shown in Figure 19.29:

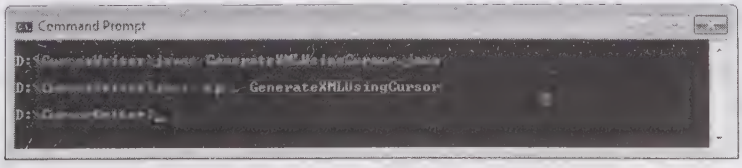


Figure 19.29: Showing the Output of the GenerateXMLUsingCursor Class

Running the `GenerateXMLUsingCursor` Java class creates the `output.xml` file shown in Listing 19.45:

Listing 19.45: Showing the output.xml File

```
<!--The namespace of price element is http://ecommerce.org/schema-->
<?xml version="1.0" ?>
<ecom:x xmlns:x="http://ecommerce.org/schema">
<x:price units="Euro">35.47</x:price>
</ecom:x>
```

Summary

In this chapter, you have learned about SOA and Java Web services. Section A has helped you to explore the basic concepts of SOA and JWSs. You have also learned about the role of WSDL, SOAP, and Java/XML mapping in SOA. In Section B, we have explored the different Web service specifications required to implement SOA. For example, the JAX-WS 2.2 specification provides a Java API that you can use to create Java Web services. The JAXB 2.2 specification provides the standards to bind Java classes to XML schema components. The WSEE 1.3 specification deals with the service architecture, packaging, and deployment of Web services. The WS-Metadata 2.2 specification provides the standards for deploying the JWSs. The SAAJ specification lays down the standards for transmitting and processing SOAP messages in compliance with the JAX-WS handlers and JAXR implementations. The JAXR specification helps access standard business registries over the Internet. Finally, we have learned that the StAX specification provides various APIs to control the parsing of XML documents. We also explored how to implement these Web service specifications in detail in Section C of the chapter.

Quick Revise

Q1. What is a Web service?

Ans. Web services present model by which tasks of e-business processes are distributed widely through Internet. This model is not restricted to specific business model. Web services are not graphical user interfaces but they can be used into software meant for user interaction. They describe their inputs and outputs in a manner that second party can predict its functionality, how to call it, and expected results. The Web services are reusable software components and let developers to reuse basic elements of code made by others. A simple example of Web service is auction engine, such as eBay. This website provides successful auction service. Nowadays, businesses that sell products wish to add auction process to own business model. For this auction process, these businesses or companies require to build the auction software from start or alternatively they can send the customers of products to an auction website, such as eBay. Using Web services, eBay leverage its auction process to other websites and applications for some fee. Businesses only need to subscribe to eBay's Web service and provide some lines of code to their applications to use Web service. Other uses of Web services are payroll management, credit scoring, shipping, business intelligence, and mapping services.

Q2. What is SOAP?

Ans. SOAP, i.e., Simple Object Access Protocol, is a protocol which allows applications to exchange information. It is not a language or platform specific protocol, and; thereby, allows communication between applications running on different platforms. SOAP is a text-based protocol and uses XML-based rule to allow applications interchange information over HTTP.

Q3. What is WSDL?

Ans. Web Services Description Language (WSDL) is an Interface Definition Language (IDL) on the basis of which SOA components can easily communicate with each other. It is a standard language to write guidelines for communicating with a component. Without this language, service providers must provide different documentations for communication with a component for different clients.

Q4. What is a relationship between SOA and Web Services?

Ans. Web Service technology is a best method to implement service-oriented architecture. Web services have self describing interfaces for clients and distributed into modularized services (which encapsulate business logic) that can be registered, searched, and called over the Internet. The modular services are loosely coupled which allows them to access by anyone at any location using any platform.

Q5. What is Java/XML binding?

Ans. The Java/XML binding are defined by using JAXB 2.2 annotations in Java classes. Each Java class maps to a unique XML schema component depending upon its annotations. You can implement type mappings in the Java/XML binding process in the following two ways:

- Begin with an existing Java application and takes help of schema generator to produce machine-generated XML schema. In this way, JAXB user develops Java application and use JAXB annotations or binding language inside the application to map it to a particular schema.

- Begin with an existing XML schema and takes help of a schema compiler to produce a machine-generated Java application. In this way, XML developer writes XML schema, creates a Java application from schema using the schema compiler, and customize it using annotations or binding language.

Q6. What is Java/XML type mapping?

Ans. Java/XML type mappings are implemented between existing Java application and existing XML schema definitions. This mapping is done by using user defined mappings between XML and Java. In this process, there is no generation of machine generated artifacts at runtime.

Q7. What is Java/WSDL mapping?

Ans. Java/WSDL mapping defines the binding between WSDL operations and Java methods. Initially, the SOAP message requests for a WSDL operation. Then, Java/WSDL mapping is used to invoke the associated Java method and map the SOAP message to the parameters of this method. Java/WSDL mapping is also used to map the return value of the method to the SOAP response.

Q8. What is runtime endpoint publishing?

Ans. JAX-WS supports publishing of Web service endpoints at runtime. The instance of the javax.xml.ws.EndPoint class used to assign the instance of the Web service implementation class to a URL. Dynamic publishing of end point is only supported by Java SE 6 while the Java EE 6 container does not support publishing of endpoint dynamically. The Java EE experts are trying to include this feature in future versions of Java EE.

Q9. Define Java API for XML- based Web Services (JAX-WS) Specification.

Ans. JAX-WS is a Java Web services specification that is used for deploying and invoking Web services. JAX-WS server side facilities helps us deploying a Web service partially and JAX-WS on client side helps us in building an SOA based client to consume or invoke a Web service.

Q10. Define Web Services Metadata (WS-Metadata) Specification.

Ans. The WS-Metadata specification provides annotations which are used to develop and deploy Web services on Java SE 6 and Java EE 6 platforms. This API makes development, deployment, and invocation of Web service easy.

Q11. Define Java for XML Binding (JAXB).

Ans. The JAXB 2.2 specification helps to bind XML instances with Java classes. This can be done in the following ways:

- Developers first create Java classes and then the JAXB 2.2 schema generator is used for generating the XML schema from these Java classes
- Developers first have XML schema which is used for generating Java classes using the JAXB 2.2 schema compiler

Q12. Define Streaming API for XML (StAX).

Ans. The StAX 1.0 specification is a joint effort of BEA and Sun Microsystems. The StAX API allows programmers to perform iterative and event based processing of XML documents, which are considered as filtered series of events.

Q13. Define Java API for XML Registries (JAXR).

Ans. The Java API for XML Registries (JAXR) API allows the different kinds of XML registries to be accessed in a uniform and standardized manner. XML registries provide an enabling infrastructure that aids in building, deployment, and discovery of Web services.

Q14. Define SOAP with Attachments API for Java (SAAJ).

Ans. You can build Web service applications directly using XML messages instead of using JAX-WS API. These applications work with SOAP messages using SOAP with Attachments API for Java (SAAJ). SAAJ specification helps in creating a SOAP message on sender side and in transmitting on receiver side.

Q15. What is schema generator?

Ans. Schema generator performs the reverse operation to that of the schema compiler. It maps a collection of schema-based program elements to a source schema by using Java annotations.

Q16. What is schema compiler?

Ans. Schema compiler binds a source schema to a collection of schema-based program elements by using the JAXB 2.2 binding language.

Q17. What is port component?

Ans. The WSEE specification defines port component as a component which is packaged and deployed on container to implement a Web service. In JAVA EE 1.4, WSEE 1.0 specification includes artifacts for deployment in a port component, such as SEI, Web services Deployment Descriptor (webservices.xml), and JAX-RPC mapping Deployment Descriptor. In Java EE 6, WSEE 1.3 specifications makes optional to include artifacts, such as WSDL document, SEI, and webservices.xml. If developer defines the webservices.xml Deployment Descriptor, then descriptor overrides the deployment information specified in annotations.

Q18. Define servlet and EJB endpoints.

Ans. Servlet endpoint: Refers to the Service implementation bean which when implemented as a servlet class is known as servlet endpoint.

EJB Endpoint: Implies that the WSEE 1.3 specification allows a stateless session bean to implement a Web service deployed in an EJB container. This stateless session bean is also known as EJB endpoint.

Q19. Define Service Implementation Bean (SIB).

Ans. WS-Metadata 2.2 specifies some conditions on Java class to deploy it as a Web service. The Java classes that satisfy these conditions are known as Service Implementation Beans.

Q20. Define SEI.

Ans. The SEI is key artifact in a port component that is deployed on application server.

Q21. What is a registry?

Ans. A registry can be defined as a shared resource among number of different business to make Business to Business (B2B) interaction possible in a flexible way and in the form of web-based service. In the Web service architecture, the registry plays a key role as it is used to publish, find, and utilize Web services.

Q22. What find methods BusinessQueryManager interface provides to search data from a JAXR registry?

Ans. The BusinessQueryManager interface provides several find methods to search data on the basis of JAXR information model. Important methods used for search purpose are given as follows:

- **findOrganizations** – Returns a BulkResponse object that is a collection of organizations having a specified name pattern a specific category
- **findServices** – Returns a BulkResponse object that is a collection of services which a particular organization offers
- **findServiceBindings** – Returns a BulkResponse object that is a set of service bindings supported by a particular service

Q23. What are the various ways of deploying Java EE 6Web services?

Ans. The different deployment methods provided by WSEE specification are given as follows:

- Deployment using a servlet endpoint
- Deployment using an EJB endpoint
- Deployment without Deployment Descriptors
- Deployment with Deployment Descriptor

20

Working with Struts 2

Struts 2 is an open source framework for creating Java Web applications; or in other words, it is released to the public domain free of charge. The Struts 2 Framework is designed for creating Web applications based on MVC architecture. As Struts 2 is based on the MVC architecture; it separates the business logic code, page design code, and navigational code into three different components called model, view, and controller. The introduction of these three components helps developers to easily maintain large Web applications. The Struts 2 Framework includes a library of mark-up tags, which are used to create dynamic data. These mark–up tags interact with the validation mechanism of Struts 2 to ensure that the output is correct. The tag library can be used with JavaServer Pages (JSP), Velocity, FreeMarker, JavaServer Pages Standard Tag Library (JSTL), and Asynchronous JavaScript and XML (AJAX) technology. The Struts framework was introduced to support the development of a Web application with its set of Application Programming Interfaces (APIs). These APIs provide a specific architecture and a mechanism to reuse the model, view, and controller parts of a Web application.

This chapter describes the key features of Struts 2, such as interceptors, results, XWork Validation framework support, integration with Object-Graph Navigation Language (OGNL), and implementation of Inversion of Control (IoC), which make this framework stand apart from other frameworks.

Introducing Struts 2

Struts1 was the first version of the Struts Framework. With the changing technology, the need for new enhancements and changes was felt in the original design of Struts 1. These changes include development of new and lightweight MVC-based frameworks, such as Spring, Stripes, Tapestry, and so on, to enable rapid development of Web applications. Therefore, it became necessary to modify the Struts Framework. Two new frameworks, Shale and Struts Ti were introduced in response to the growing need for rapid development of Web applications. Then in March 2002, the WebWork framework was released. WebWork includes new ideas, concepts and functionality with the original Struts code. In December 2005, WebWorks and the Struts Ti merged to develop Struts 2.

Struts 2 has features similar to Struts 1, but the Struts 2 Framework has some architectural differences as compared to Struts 1. In general, the Struts 2 Framework implements the MVC 2 architecture by centralizing the control using a front controller strategy similar to Struts 1. However, the basic code of components and their configuration is quite different in Struts 2.

Explaining MVC 2 Design Pattern for Struts 2

The Struts 2 Framework follows the MVC framework, where model, view, and controller components are represented by Action, Result, and FilterDispatcher classes, respectively. Figure 20.1 shows the implementation of the MVC pattern by Struts 2 components:

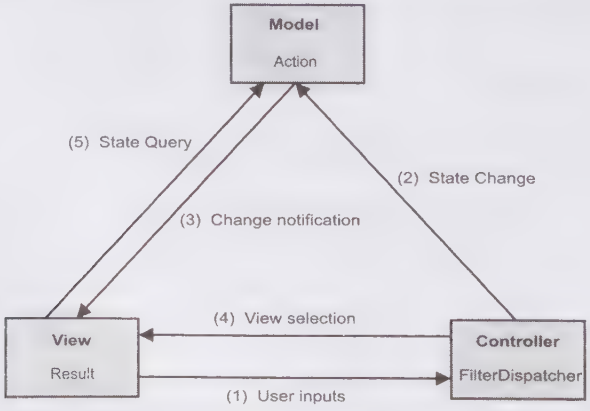


Figure 20.1: Implementing the MVC Pattern in Struts 2

According to Figure 20.1, the following are the steps for the work flow in Struts 2:

1. The user sends a request through a user interface provided by the view, which further passes this request to the controller, represented by `FilterDispatcher` class in Struts 2.

2. The controller servlet filter receives the input request coming from the user through the interface provided by the view, instantiates an object of the suitable action class, and executes different methods over this object.
3. If the state of model is changed, all the associated views are notified about the changes.
4. Next, the controller selects the new view to be displayed according to the result code returned by the action class.
5. The view presents the user interface. The view queries about the state of the model to show the current data, which is retrieved from the action class.

The Need for Struts 2

Struts 2 is created with the intention to streamline the entire development cycle of Web applications, which includes building, deploying, and maintaining these Web applications. Each Web application development framework has its own architecture; and the way the components are designed and configured using these frameworks is also different. What makes Struts 2 a better option amongst the other frameworks can be explained with the help of the features provided by Struts 2. These features are categorized into build supporting features, deployment supporting features, and maintenance supporting features.

Build Supporting Features

The build supporting features of Struts 2 are:

- The stylesheet-driven form tags: Decrease coding effort and reduce the requirement for input validation.
- Smart checkboxes: Provide the capability to select or clear checkboxes in a form with a single click.
- Support for AJAX tags: Help design interactive Web applications.
- Integration with Spring application framework: Provides support to integrate applications created by using Struts 2 with the Spring application framework. Integration with Spring application framework provides more control on struts actions and allows applying Aspect Oriented Programming (AOP) technique rather than object-oriented code.
- Support for action chaining and file downloading: Specifies that results obtained after processing of a request can be processed further for action chaining and file downloading.
- Use of JavaBeans for form inputs: Helps in placing binary and string properties directly on action class with the help of JavaBeans.
- Support for controller and model: Provides the controller and model to handle the role of interfaces in Struts 2, as interfaces are not used in the Struts 2 Framework.

Deployment Supporting Features

The deployment supporting features of Struts 2 are:

- Allows framework extension, automatic configuration, and use of plugins to enhance the capabilities of the framework
- Helps in debugging a Struts 2 application by ensuring that errors are reported precisely by indicating the location and line of an error
- Maintenance Supporting Features
- The maintenance supporting features of Struts 2 are:
- Help test Struts actions directly, without using the conventional HTTP objects to debug the application
- Allows easy customization of controller to handle the requests per action, since a new class is invoked for every new request
- Provides built-in debugging tools to report problems; therefore, less time is wasted in manual testing
- Supports JSP, FreeMarker, and Velocity tags that encapsulate the business logic

Processing Request in Struts 2

Similar to Struts 1, Struts 2 implements the front controller approach with the MVC 2 pattern, which means that there is a single controller component here. All requests are mapped to this front controller called

`org.apache.struts2.dispatcher.FilterDispatcher class`. The main work of this controller is to map user requests to the appropriate action classes. Unlike Struts 1, where actions are used to hold the business logic and the model part is represented by JavaBeans, in Struts 2, both the business logic and the model are implemented as action components. In addition to JSP pages, view can be implemented by using other presentation layer technologies, such as the Velocity template, in Struts 2.

The request flow in Struts 2 Web application framework is shown in Figure 20.2:

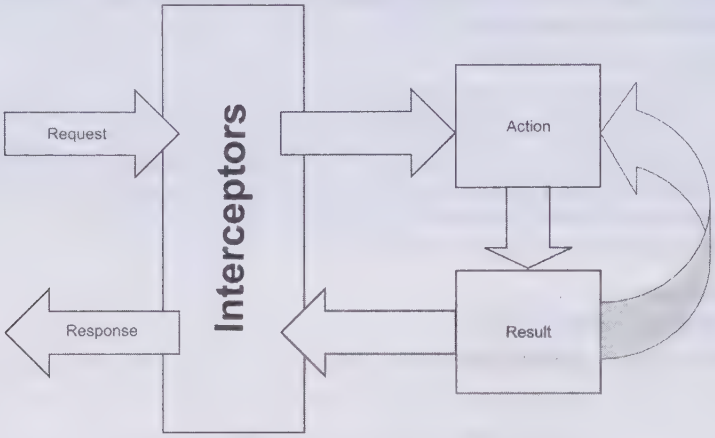


Figure 20.2: Processing Request in Struts 2

As evident from Figure 20.2, Struts 2 processes a request in the following steps:

1. **Request received**—Refers to the step in which the framework matches the request received from the user with the configuration file, struts-config.xml, to invoke the required interceptors, action class, and result class.
2. **Pre-processing by Interceptors**—Refers to the step in which the request passes through a series of interceptors. These interceptors perform certain tasks, such as initialization required to process a request.
3. **Invoke the Action class method**—Refers to the step in which a new instance of an action class is created and the method providing the logic for handling the request is invoked. Note that in Struts 2, a method to be invoked by an action class is specified in the configuration file.
4. **Invoke Result class**—Refers to the step in which the `Result` class determines the mapped class (in struts.xml) on the basis of the obtained result. Then, a new instance of this return class is created and invoked.
5. **Processing by Interceptors**—Refers to the step in which the response is passed through the interceptors in reverse order to perform any clean up or additional processing.
6. **Responding user**—Refers to the step to display the processed response back to the user by the servlet engine.

Exploring Relation between WebWork 2 and Struts 2

Struts 2 framework is dependent on the WebWork2 framework, and includes features of both the WebWork2 and Struts 1 frameworks. Struts 2 inherits certain features from the WebWork2 framework, such as interceptors, results, and so on. Several files and packages have been included from WebWork2 in the Struts 2 Framework.

In addition, some features of WebWork2 have been removed from Struts 2; while some new features have been added in the Struts 2 Framework.

Some features of WebWork2 included in Struts 2 (with some modifications) are as follows:

- ConfigurationManager is not a static factory in Struts 2; instead, an instance is created through Dispatcher.
- The tooltip library used by the xhtml theme has been replaced by Dojo's tooltip component.
- Tiles integration is available as a plug-in in Struts 2.

- Use of wildcards in action mappings is allowed.
- The MessageStoreInterceptor class has been introduced in Struts 2 so that field errors or action errors can be stored and retrieved through a session. Action's messages are also stored and retrieved through the session.

The features that have been removed from WebWork2 to develop Struts 2 are:

- AroundInterceptor—Refers to the class that has been removed in WebWork2. If you are extending the AroundInterceptor Class in your application, you need to import the AroundIntercepter class into the source code and modify it to serve as the base class, or rewrite the Interceptor.
- oldSyntax—Refers to the oldSyntax attribute that has been removed from WebWork2.
- Rich text editor tag—Refers to the tag that has been removed and replaced by the Dojo's rich text editor.
- Default method—Refers to the doDefault method, which is not supported in Struts 2.
- Inversion of Control Framework—Refers to the framework that has been deprecated in WebWork 2.2 and removed in Struts 2. The Struts 2 Framework provides a Spring plugin to implement IoC. This is implemented by using the ObjectFactory factory.

In addition, certain features of WebWork2, inherited by Struts 2, have been renamed in Struts 2, as listed in Table 20.1:

Table 20.1: Comparing Struts 2 and WebWork2

Basis Of Comparison	WebWork2	Struts 2
Action Support Class	com.opensymphony.xwork.*	com.opensymphony.xwork2.*
Interface	com.opensymphony.webwork.*	org.apache.struts2.*
Configuration File	xwork.xml	struts.xml
Properties File	webwork.properties	struts.properties
RequestDispatcher class	DispatcherUtil	Dispatcher
Configuration Settings class	com.opensymphony.webwork.config.Configuration	org.apache.struts2.config.Settings

Let's now discuss the implementation of MVC 2 architecture by Struts 2 framework.

Describing Struts 2 Architecture

Struts 2 contains various framework components, such as built-in classes, servlets, and Struts tags. All the framework components work together in a standard manner. The architecture of Struts 2-based application defines the relationship and interaction between these framework components.

Components of a Struts 2-Based Application

The architecture of a Struts 2-based application has been shown in Figure 20.3:

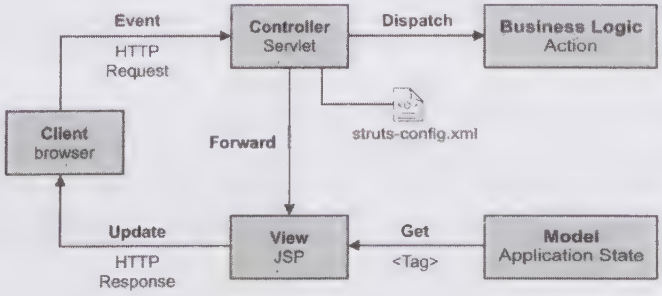


Figure 20.3: Displaying the Struts 2 Architecture

If we compare Figure 20.3 with Figure 20.1, we notice that a new group of components, called Business Logic, has been created. This can be seen as the separation of business logic from the model layer of the traditional

MVC architecture. The business logic component allows the model layer to store only the state of a Web application. The components created in the new group are known as action classes.

Let's discuss different Struts 2 components one by one next.

Controller

The controller receives the requests from users and decides where to send the request. The main aim of the controller is to map the request Uniform Resource Identifier (URI) to an action class using the mappings provided in the Struts-config.xml file. There is a single Controller Servlet and all requests go through this Servlet, which is provided by the framework itself.

The controller component available in Struts 2 framework is the `ActionServlet` class that represents the controller in the MVC design pattern. In addition, `ActionServlet` class implements both the front controller and the singleton pattern. The front controller pattern allows a centralized access point for presentation-tier request handling, and the singleton pattern provides a single instance of an object.

You should note that the `ActionServlet` instance selects the appropriate action class that needs to perform the requested business logic and then invokes the `action` class. The action classes do not produce the next page of the user directly; rather it is the duty of the `RequestDispatcher` class to forward the control to an appropriate JSP page. Generally, the Servlet engine uses the `RequestDispatcher.forward()` method of the Servlet API to perform this task.

The struts-config.xml File

The `struts-config.xml` file contains the transformation and configuration information for the Struts application. It provides information regarding various Struts resources, such as action, classes, and interceptors. The relation between user requests, the action class to be invoked, the ActionForm class to be used, and the next possible views are defined in the struts-config.xml file using appropriate mappings.

Action Classes

Action classes implement the business logic; or, in other words, they behave as wrappers around the business logic and interact with the model of the application. They are basically responsible for executing the business service according to user requests.

Model

In the MVC architecture, model represents the data objects that are defined with the help of JavaBeans. In Struts 2, the model is represented by org.apache.struts.action.ActionForm class, which provides the methods to get and set data fields with methods to validate the data.

View

The view represents a JSP or HTML page, which encapsulates the presentation semantics. A view does not include business logic.

Exploring Struts 2 Configuration Files

For configuring a Web application, Struts 2 loads a set of configuration files. The files used to configure an application in Struts 2 are:

- web.xml
- struts.xml
- struts.properties
- struts-default.xml

The preceding configuration files are dynamically reloaded by Struts 2. Dynamic loading of configuration files helps in reconfiguring the action mapping while developing Struts 2 application.

Let's now discuss the Struts 2 configuration files in detail.

The web.xml File

As Struts 2 applications include Servlet programs, it is necessary to keep the mapping functions inside the web.xml file. The web.xml file must reside in the WEB-INF folder of the Web application. The web.xml file is a deployment descriptor file, which represents the core of the Web application. The web.xml file defines the front controller, i.e. the `FilterDispatcher` class, which is a servlet filter class used to initialize the Struts 2 Framework and manage all the incoming requests.

The FilterDispatcher class can contain initialization parameters to process any additional configuration files that are loaded by the Struts 2 Framework. These key initialization parameters are processed with the help of the <filter> element of the web.xml file. The following are key initialization parameters of the `FilterDispatcher` class:

- **config** – Loads a comma-delimited list of XML configuration files
- **actionPackages** – Provides a comma-delimited list of Java packages to locate the required action classes
- **configProviders** – Loads a comma-delimited list of Java classes used for configuring the action classes

The following code snippet shows how to process the key initialization parameters of the FilterDispatcher class with the help of the<filter> and <filter-mapping> elements of the web.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

<filter>
  <filter-name>struts</filter-name>
  <filter-class>
  org.apache.struts2.dispatcher.FilterDispatcher
  </filter-class>
</filter>
  <filter-mapping>
  <filter-name>struts</filter-name>
  <url-pattern>/*</url-pattern>
  </filter-mapping>

  <!-- ... -->
</web-app>
```

In the preceding code snippet, the <url-pattern>, which is set to /*, ensures that all requests are handled by the FilterDispatcher class. The `FilterDispatcher` class further handles the client requests and uses other configuration details given in different configuration files.

The struts.xml File

The struts.xml file is a default file for the Struts 2 Framework, also known as the core configuration file for this framework. The struts.xml file must be stored in the WEB-INF/classes folder of the Web application. This file contains various configuration details for framework specific components, such as actions, results, and interceptors.

If required, you can also insert other configuration files into the struts.xml file. The following code snippet shows how to load different configuration files in the struts.xml file:

```
<struts>
  <include file="struts-default.xml"/>
  <include file="config-browser.xml"/>
  <package name="default" extends="struts-default">
        ..........
  </package>
  <include file="other.xml"/>
</struts>
```

In the preceding code snippet, the first include statement instructs the Struts Framework to load the struts-default.xml file, which is found in the struts2-core-2.0.6.jar file. The struts-default.xml file defines the default

bundled results, interceptors, and interceptor stacks. The files included in the struts.xml file are loaded in the order of their mappings.

The following code snippet shows an implementation of some of the important elements of the struts.xml file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.1//EN"
"http://struts.apache.org/dtds/struts-2.1.dtd">
<struts>
  <include file="struts-default.xml"/>
  <package name="default" extends="struts-default">
        <result-types>
              <result-type name="result1" class="SomeResultClass1"/>
              <result-type name="result2" class="SomeResultClass2"/>
        </result-types>
        <interceptors>
        <interceptor name="interceptor1" class="SomeInterceptorClass1"/>
        <interceptor name="interceptor2" class="SomeInterceptorClass2"/>
        <interceptor-stack name="SomeStack1">
              <interceptor-ref name="somename1" />
              <interceptor-ref name="somename2" />
        </interceptor-stack>
        </interceptors>
        <default-interceptor-ref name="defaultStack1"/>
        <global-results>
              <result name="resultname" type="resulttype1">/XXX</result>
        </global-results>
        <action name="action" class="SomeActionClass">
              <result name="success">some_page.jsp</result>
        </action>
  </package>
</struts>
```

In the preceding code snippet, the components of Struts 2 are configured in the `struts.xml` file. The various components or elements are configured under the opening and closing tags of the `<struts>` element. The default configuration file, `struts-default.xml`, has also been included in the struts.xml file. Only the elements usually configured in the `struts.xml` file are currently shown in the file. These elements include `<result-types>`, `<interceptors>`, `<interceptorstack>`, `<global-results>`, and `<action>`.

The struts.properties File

A Struts 2 application uses a number of properties, which can be changed according to the requirement of a user. These properties are specified in the `struts.properties` file, which is located in the `WEB-INF/classes` folder of the Web application. This property file is similar to other resource bundle files with the `.properties` extension. The property file contains key-value pairs, representing various properties and their values.

The `struts.properties` file can be used to override the default values of the keys set in the `default.properties` file. For example, setting the `struts.devMode` property to true helps in debugging Struts-based applications. Table 20.2 describes different properties, which are defined in the `default.properties` file and can be overridden in the `struts.properties` file:

Table 20.2: Properties of the default.properties File

Properties	Description
`struts.configuration = org.apache.struts2.config. DefaultConfiguration`	Defines the configuration class required to configure Struts, and retrieves the configuration parameter.
`struts.locale=en_US struts.i18n.encoding=UTF-8`	Sets the default locale and encoding scheme in the Struts application.
`struts.objectFactory=spring`	Specifies that the default object factory can be overridden.

Table 20.2: Properties of the default.properties File

Properties	Description
`struts.objectFactory.spring.autoWire = name`	Defines the autowiring logic when using the SpringObjectFactory class. The autowire property can be set to four options, such as name, type, auto, and constructor. The default value is name.
`struts.objectFactory.spring.useClassCache = true`	Specifies the Struts-Spring integration. If the class instance is cached, it should be set to true. By default, the value of this property is true.
`struts.objectTypeDeterminer = tiger`	Specifies that the default object type, determiner, can be overridden.
`struts.objectTypeDeterminer = notiger`	Specifies that the notiger value is used to disable tiger support.
`struts.enable.SlashesInActionNames = false`	Allows slashes in your action names. You need to set a boolean value for the `SlashesInActionNames` property. If false, action names cannot have slashes and will be accessible through any directory prefix. The value true is useful when you use wildcards and store the value in the URLs.
`struts.tag.altSyntax= true`	Uses alternative syntax containing %{} in most places used to evaluate the expression for String attributes of tags.
`struts.devMode = false`	Sets development mode by setting a boolean value for this property. When the true value is set, Struts 2 provides additional logging and debug information, which can speed up the process of Web application development.
`struts.configuration.xml.reload=false`	Reloads the struts.xml configuration file, when the file is changed.
`struts.url.http.port = 80` `struts.url.https.port =443`	Allows you to build URLs.
`struts.url.includeParams=get`	Sets the three possible options for this property: none, get or, all.
`struts.custom.i18n.resources=testmessages, testmessages2`	Loads the custom default resource bundles.
`struts.xslt.nocache=false`	Allows you to use stylesheet caching, by configuring the XSLTResult class and setting the value of this property to true.
`struts.configuration.files=struts-default.xml, struts-plugin.xml,struts.xml`	Shows a list of configuration files that are automatically loaded by Struts.
`struts.mapper.alwaysSelectFullNamespace=false`	Defines whether or not to always select the namespace before the last slash provided in the namespace.

Let's now discuss Struts 2 applications that do not need XML files for configuration related matters.

Explaining Zero Configuration Applications

Zero configuration Struts 2 applications are Web applications that do not use additional files, such as XML and properties files, to configure different components of the Web application. Instead, annotations are used to provide details about the action class to be executed, the result to be used, validations, and type conversions.

Struts 2 support annotations, which provide information regarding configuring zero configuration Struts 2 applications. Let's discuss annotations in the next section.

Exploring Struts 2 Annotations

Annotations provide information to configure Zero configuration Struts 2 applications. In case of Java, annotation can be defined as a code segment that describes the Java classes, methods, and fields. Annotations are implemented as an interface and can be packaged, compiled, and imported similar to any other classes. Most of the time, the user wants to give more information to a particular piece of code. This is where annotation plays an important role. One of the benefits of the new annotation specification is that annotation enables you to create quite complex structures of Java classes or interfaces, while maintaining type safety.

Annotation in Struts 2 is generally divided into four different types:

- **Action annotation**—Represents the annotations used while creating an action class. There are different annotations available to configure namespace and results for a given action. Action classes need not be configured in the struts.xml file for their associated mapping. This results in zero configuration for actions. There are different annotations, which help in defining namespace for the action, extending a class from the parent package, and associating the results with the actions. Generally, these annotations are collectively known as action annotations. There are four types of action annotations: Namespace annotation, ParentPackage annotation, Result annotation, and Results annotation.
- **Interceptor annotation**—Represents the annotations used to configure different methods of the action class. This ensures that an action class method is invoked before or after the execute() method; consequently intercepting the execution path. There are three types of interceptor annotations: @After annotation, @Before annotation, and @BeforeResult annotation.
- **Validation annotation**—Represents the validation annotations that help in implementing validation rules on various validation fields, without mapping them in the XML files. Different validation annotations are provided for corresponding validation rules. We can use different validation annotation in methods to validate the data being set. Similar to other annotations, these annotations have their own set of attributes. The most frequently used validation annotations are @ConversionErrorFieldValidator, @DateRangeFieldValidator, @DoubleRangeFieldValidator, @EmailValidator, @IntRangeFieldValidator, @RequiredFieldValidator, @RequiredStringValidator, and @StringLengthFieldValidator.
- **Type conversion annotation**—Represents the type conversion annotation that provides the mechanisms to avoid the use of any ClassNameconversion.properties file. There are six kinds of type conversion annotations: @Conversion annotation, @CreateIfNull annotation, @Element annotation, @Key annotation, @KeyProperty annotation, and @TypeConversion annotation.

We learned about the rich set of annotations available in Struts 2, which can be used with the code to reduce the declaration in configuration files. Let's now discuss mechanism to implement actions in Struts 2.

Understanding Actions in Struts 2

Struts 2 is a front controller based framework, i.e., there is a single controller that handles all the requests and executes an appropriate action in response to those requests. Specific classes, known as *action classes,* are executed to process the response to requests. We can have a number of action classes that embed different business logics for a particular user action. The flow of execution of an action class involves a sequential invocation of different methods. This sequence of method invocation is handled by different interceptors, which simply pre-process the request before executing the main logic embedded in the action class.

Struts 2 has introduced a simpler implementation for the action class, which can be created by implementing some interfaces, extending some classes, or even using Plain Old Java Objects (POJOs). In other words, creating and configuring action classes have become more flexible and simpler in Struts 2.

Let's now discuss the mechanism of creating, configuring, and using action classes in Struts 2.

Action Classes

The action classes are configured to be executed when a particular request for these actions is submitted. An action class contains the execute() method as the default entry point where the execution of the action class starts. The next process to be performed depends on the result returned by the action class, which may include

rendering the next JSP page or invoking another action. The following code snippet provides the execute() method of action classes in Struts 2:

```
//execute() method of Struts 2 action classes
public String execute() throws Exception {
  //some logic implementation
}
```

In the preceding code snippet, the execute() method returns a String value from a set of standard values that are defined to be returned by the execute() method. These standard values are success, input, error, none, and login, which are returned by the execute() method of Struts 2 action classes. When the String value returned by the execute() method of an action class is matched with a result configured for that action class in struts.xml file, the JSP page set for that result is executed.

In Struts 1, an action class can be created by extending the org.apache.struts.action.Action class only, while in Struts 2 there are different ways to create an action class. Struts 2 provides flexibility in creating a new action class by implementing the com.opensymphony.xwork2.Action interface or by extending the com.opensymphony.xwork2.ActionSupport class. In addition, Struts 2 supports pure Java classes to be treated as actions classes, known as POJO actions, which enable us to create action classes without implementing any interface and extending any class. The only convention to be followed is the availability of the execute() method in the action classes.

We can create different action classes for different use cases. For every request, an action mapping is searched and the associated action class methods are invoked. Struts 2 actions are not singletons; in other words, for each request, a different instance of the action class is created. Therefore, Struts 2 action need not be thread safe, as in case of Struts 1 actions. So, how can we access different objects to work with, such as request, response, and session? The solution to this problem is Dependency Injection (DI) pattern.

The Dependency Injection pattern used in Struts 2 is interface injection. This means that we can implement various interfaces that provide different methods to the action class. These implemented methods should be invoked in the predefined order, which is assured by the use of appropriate Interceptors configured for the action.

Let's now discuss the various interfaces and classes used to create, configure, and execute the action class. Some of these interfaces and classes are:

- The Action interface
- The ActionSupport class
- The ActionMapping class
- The DefaultAction class
- The ActionContext class

The Action Interface

Though we can create action classes without implementing any interface and extending any class, the Struts 2 Framework provides an interface that can be used to create action classes to simplify the code in the action classes. The com.opensymphony.xwork2.Action interface provides the `execute()` method to execute the action class. In addition, the Action interface provides common results (success and error) as string constants, which are described in Table 20.3:

Table 20.3: String Constants of the Action Interface

Field Name	Description
`static String ERROR`	Specifies that the execution of action has failed
`static String INPUT`	Specifies that the action class requires more input to be executed
`static String LOGIN`	Specifies that the execution of the action class needs a logged in user
`static String NONE`	Specifies that the execution of the action class has been successful, but no view is provided

Table 20.3: String Constants of the Action Interface	
Field Name	**Description**
`static String SUCCESS`	Specifies that the execution of action was successful

The `Action` interface provides the execute() method, which all action classes implementing this interface must override. The following code snippet shows how to create a sample action class by using the `Action` interface:

```
import com.opensymphony.xwork2.Action;
public class SomeAction implements Action{
  public String execute() throws Exception {
        return SUCCESS;
  }
}
```

In the preceding code snippet, an action class called `SomeAction` implements the `Action` interface. The `Action` interface provides the `execute()` method to the `SomeAction` class. The `execute()` method returns `SUCCESS` as a result type for this action class.

The ActionSupport Class

The `ActionSupport` class provides basic implementation for common actions, such as validation. The ActionSupport class implements various interfaces, such as `Action`, `LocaleProvider`, `TextProvider`, `Validateable`, `ValidationAware`, and `Serializable`. We can create the action classes by extending the `ActionSupport` class and overriding the required methods.

We will now briefly discuss some of these interfaces.

The Validateable Interface

The `com.opensymphony.xwork2.Validateable` interface provides a `validate()` method to validate an action class. You must override the `validate()` method in the action class to implement your logic to validate the data.

The LocaleProvider Interface

The `com.opensymphony.xwork2.LocaleProvider` interface provides a getLocale() method to specify the locale to be used for getting localized messages. This Locale is used in an action class to override the default locale, whenever needed. A locale refers to an object representing a particular geographical, political, or cultural region.

The ValidationAware Interface

The `com.opensymphony.xwork2.ValidationAware` interface provides methods to save and retrieve class-level actions and field-level error messages. Collections classes are used to store/retrieve action level error messages, and Maps classes are used to store/retrieve field-level error messages. Table 20.4 describes the methods of the `ValidationAware` interface:

Table 20.4: Methods of the ValidationAware Interface	
Methods	**Description**
`void addActionError(String anErrorMessage)`	Adds an action-level error message to the current action
`void addActionMessage(String aMessage)`	Adds an action-level message to the current action
`void addFieldError(String fieldName, String errorMessage)`	Adds an error message for the specified field
`Collection getActionErrors()`	Retrieves the collection of action-level error messages for this action
`Collection getActionMessages()`	Retrieves the collection of action-level messages for this action
`boolean hasErrors()`	Verifies whether there are any errors (action level or field level) set or not

Table 20.4: Methods of the ValidationAware Interface	
Methods	**Description**
`boolean hasFieldErrors()`	Verifies whether there are any field errors associated with this action
`void setActionErrors (Collection errorMessages)`	Sets the collection of action-level String error messages
`void setActionMessages (Collection messages)`	Sets the collection of action-level String messages instead of errors
`void setFieldErrors(Map errorMap)`	Sets the field error map of fieldname (String) to collection of String error messages

The TextProvider Interface

The `com.opensymphony.xwork2.TextProvider` interface provides methods for getting localized message texts. The `TextProvider` interface is used to access the text messages in the resource bundle. Table 20.5 describes various methods of the `TextProvider` interface:

Table 20.5: Methods of the TextProvider Interface	
Methods	**Description**
`String getText(String key)`	Retrieves a message on the basis of a message key. However, if the message is not found, the null value is returned.
`String getText(String key, List args)`	Retrieves a message on the basis of the message key using the specified `args` provided in the form of a list, as defined in `MessageFormat`. However, if no message is found, the null value is returned.
`String getText(String key, String defaultValue)`	Retrieves a message on the basis of the message key. However, if the message is not found, the supplied default value is returned.
`String getText(String key, String[] args)`	Retrieves a message on the basis of a key and using the args supplied in the form of String type array, as defined in MessageFormat. However, if no message is found, the null value is returned.
`String getText(String key, String defaultValue, List args)`	Retrieves a message on the basis of a key and using the supplied args, as defined in MessageFormat. However, if the message is not found, a supplied default value is returned.
`String getText(String key, String defaultValue, List args, ValueStack stack)`	Retrieves a message on the basis of a key using the supplied args provided in the form of a list, as defined in MessageFormat and the specified value stack. However, if the message is not found, the supplied default value is returned.
`String getText(String key, String defaultValue, String obj)`	Retrieves a message on the basis of a key using the supplied obj, as defined in MessageFormat. However, if the message is not found, the supplied default value is returned.
`String getText(String key, String defaultValue, String[] args)`	Retrieves a message on the basis of a key using the supplied args in the form of the String type array, as defined in MessageFormat. However, if the message is not found, a supplied default value is returned.
`String getText(String key, String defaultValue, String[] args, ValueStack stack)`	Retrieves a message on the basis of a key using the supplied args provided in the form of String type array, as defined in MessageFormat, and value stack. However, if the message is not found, a supplied default value is returned.
`ResourceBundle getTexts()`	Retrieves the resource bundle related to the implementing class (usually an action).

Table 20.5: Methods of the TextProvider Interface	
Methods	**Description**
`ResourceBundle getTexts(String bundleName)`	Retrieves the named bundle, such as com/kogent/demo

The `com.opensymphony.xwork2.ActionSupport` class provides default definitions of the methods of all the interfaces implemented by it. Therefore, we can create the action classes by extending the `ActionSupport` class and use these methods in our action classes. The following code snippet shows an action class, `LoginAction`, which extends the `ActionSupport` class:

```
package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;
public class LoginAction extends ActionSupport {
  private String username;
  private String password;
  public String getUsername() {
        return username;
  }
  public void setUsername(String username) {
        this.username = username;
  }
  public String getPassword() {
        return password;
  }
  public void setPassword(String password) {
        this.password = password;
  }
  public String execute() throws Exception {
        if(username.equals(password))
        return SUCCESS;
        else{
                this.addActionError(getText("app.invalid"));
                return ERROR;
        }
  }
  public void validate() {
        if ( (username == null ) || (username.length() == 0) ) {
        this.addFieldError("username", getText("app.username.blank"));
        }
        if ( (password == null ) || (password.length() == 0) ) {
        this.addFieldError("password", getText("app.password.blank"));
        }
  }
}
```

In the preceding code snippet, the action class called `LoginAction` is created by implementing the `ActionSupport` class.

The ActionMapping Class

The `ActionMapping` class is used to provide configuration associated with an action name and action class. It specifies a mapping between HTTP requests and action invocation requests, and vice-versa. For a given HttpServletRequest, the `org.apache.struts2.dispatcher.mapper.ActionMapper` interface may return null if no action invocation request matches; otherwise it may return an instance of the `org.apache.struts2.dispatcher.mapper` type. The `ActionMapping` class describes an action invocation for the Web application.

Table 20.6 describes various fields available with the `ActionMapping` class:

Table 20.6: Fields of the ActionMapping Class	
Field Name	**Description**
`private String method`	Specifies the name of the method

Table 20.6: Fields of the ActionMapping Class	
Field Name	**Description**
`private String name`	Specifies the name of the action
`private String namespace`	Specifies the namespace of the action
`private Map params`	Specifies the optional set of parameters
`private Result result`	Specifies the result of the action

Let's now discuss the methods of the ActionMapping class, as described in Table 20.7:

Table 20.7: Methods of the ActionMapping Class	
Method	**Description**
`String getMethod()`	Returns the method name
`String getName()`	Returns the name of the action
`String getNamespace()`	Returns the namespace for the action
`Map getParams()`	Returns the Map containing name/value pairs for action parameters
`Result getResult()`	Returns the result
`void setMethod(String method)`	Sets the method name
`void setName(String name)`	Sets the action name
`void setNamespace(String namespace)`	Sets the namespace name
`void setParams(Map params)`	Sets a Map of parameters
`void setResult(Result result)`	Sets the result

The action mappings are the basic units of work in the Struts 2 Framework, which map an identifier to a handler class. When a request matches with the name of the action, the specified mapping is used by the Struts Framework to determine how to process the request. Action mapping provides a set of result types, a set of exception handlers, and an Interceptor stack. In a Web application, every resource should be referred to a Uniform Resource Identifier (URI), which includes HTML pages, custom actions, and JSP pages. The Struts framework provides this facility by using action mapping, which is defined in the struts.xml configuration file. A sample action mapping is shown in the following code snippet:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.1//EN"
"http://struts.apache.org/dtds/struts-2.1.dtd ">
<struts>
  <include file="struts-default.xml"/>
  <package name="default" extends="struts-default">
        <action name="login" class="com.kogent.action.LoginAction">
                <interceptor-ref name="exception"/>
                <interceptor-ref name="params"/>
                <interceptor-ref name="workflow"/>
                <result name="success">/login_success.jsp</result>
                <result name="error">/login.jsp</result>
                <result name="input">/login.jsp</result>
        </action>
  </package>
</struts>
```

The Default Action

The default action class is used when an action is requested but is not configured in the struts.xml. In such a case, a default action class can be invoked, which performs an action to prevent error pages from being displayed if a

request does not map with the appropriate action. In other words, if no action matches in the struts configuration file, the default action class is used to display JSP result of an application.

The following code snippet shows a default action configured in a struts configuration file:

```
<package name="my-default" extends="struts-default">
  <default-action-ref name="remove">
  <action name="remove" >
        <result>/index.jsp</result>
  </action>
</package>
```

In the preceding code snippet, a default action called remove is configured to generate a view in case Struts 2 is unable to locate the requested action class. In the preceding code snippet, the remove action returns a JSP page called index.jsp as a default view to the user. Let's now discuss the methods of an action class.

The main method in any action class is its String execute() method, which is executed by default to accomplish some defined business logics. Struts 2 action classes provide flexibility of defining methods and invoking them directly by using action configuration. Therefore, for different requests, we can execute different methods of the same action class, provided these methods have a signature similar to the execute() method. This eliminates the need for creating different action classes for different actions.

Let's suppose we have two different action classes for adding and editing an employee. We can group similar type of actions into a single action class. The execute() methods of both these classes operate on a similar set of data and require similar set of validation rules to be followed. We can create a single action class (UpdateEmployeeAction), and two execute() methods of the two action classes can be converted into two different methods of this new class. These methods can be String addEmployee() and String editEmployee(). The following code snippet shows an action class using the `methodName()` method:

```
package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;
public class UpdateEmployeeAction extends ActionSupport {
  public String execute() throws Exception {
        //some logic
        return SUCCESS;
  }
  public String addEmployee() throws Exception {
        //Logic fo add new employee
        return SUCCESS;
  }
  public String editEmployee() throws Exception {
        //Logic to edit employee
        return SUCCESS;
  }
}
```

In the preceding code snippet, the addEmployee() and editEmployee() methods provide logic for adding and editing employees' details in a single class called `UpdateEmployeeAction`. Keeping these methods in a single class eliminates the need to create two separate classes for adding and editing employee details.

Let's now explore how to configure these action methods in the struts.xml file, as shown in the following code snippet:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.1//EN"
"http://struts.apache.org/dtds/struts-2.1.dtd ">
<struts>
  <include file="struts-default.xml"/>
  <package name="default" extends="struts-default">
        <action name="add"
              class="com.kogent.action.UpdateEmployeeAction"
              method="addEmployee">
              <result name="success">/add_success.jsp</result>
              <result name="error">/add.jsp</result>
              <result name="input">/add.jsp</result>
        </action>
```

```
        <action name="edit"
                class="com.kogent.action.UpdateEmployeeAction"
                method="editEmployee">
                <result name="success">/edit_success.jsp</result>
                <result name="error">/edit.jsp</result>
                <result name="input">/edit.jsp</result>
        </action>
    </package>
</struts
```

The preceding code snippet adds and edits employee details by mapping the addEmployee() and editEmployee() methods in the struts.xml file.

The **ActionContext** Class

The ActionContext class provides objects required to execute an action class. The ActionContext class provides objects such as request, response, session, parameters locale, and so on. We can obtain the reference of this class by using its own static getContext() method, as shown in the following syntax:

```
ActionContext context = ActionContext.getContext();
```

As the execute() method of Struts 2 action classes does not take any argument, we need a mechanism to access objects, such as request, response, and session. The `ActionContext` class helps obtain these objects in the action class. The following code snippet shows the use of the `ActionContext` class:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ActionContext;
import org.apache.struts2.ServletActionContext;
public class SomeAction extends ActionSupport {
  public String execute() throws Exception {
        ActionContext acx=ActionContext.getContext();
        HttpServletRequest request=
        (HttpServletRequest)acx.get(ServletActionContext.HTTP_REQUEST);
        request.setAttribute("name", "John");
        HttpSession session=
        (HttpSession)acx.get(ServletActionContext.SESSION);
        session.setAttribute("user", "kogent");
        return SUCCESS;
  }
}
```

After obtaining the reference of the `ActionContext` class using `getContext()` method, the `getContext()` method returns the value stored in the current `ActionContext` class by looking up the value for the key passed as an argument to it.

Table 20.8 describes the fields of the `ActionContext` class:

Table 20.8: Fields of the ActionContext Class

Field Name	Description
`static String ACTION_INVOCATION`	Represents a constant for the invocation context of an action
`static String ACTION_NAME`	Represents a constant for the name of the action being executed
`(package private) static ThreadLocal actionContext`	Represents a static thread local variable used by different threads independently
`static String APPLICATION`	Represents a constant for an action application context
`(package private) Map context`	Represents a map type of field to be used to store key/value pairs for a given context
`static String CONVERSION_ERRORS`	Represents a constant for the map of type conversion errors
`static String LOCALE`	Represents a constant for an action locale
`static String PARAMETERS`	Represents a constant for an action parameters

Table 20.8: Fields of the ActionContext Class	
Field Name	**Description**
`static String SESSION`	Represents a constant for an action session
`static String TYPE_CONVERTER`	Represents a constant for an action type converter
`static String VALUE_STACK`	Represents a constant for the OGNL value stack

Let's now discuss the methods of the `ActionContext` class, as specified in Table 20.9:

Table 20.9: Methods of the ActionContext Class	
Method	**Description**
`Object get(Object key)`	Performs a lookup by using the specified key and returns a value that is stored in the current `ActionContext` class
`ActionInvocation getActionInvocation()`	Retrieves the action invocation (the execution state)
`Map getApplication()`	Returns a map of the `ServletContext` class
`static ActionContext getContext()`	Returns the `ActionContext` class associated with the current thread
`Map getContextMap()`	Returns the context map object
`Map getConversionErrors()`	Returns the map of conversion errors occurred during execution of an action
`Locale getLocale()`	Returns the locale of the current action
`String getName()`	Gets the name of the current action
`Map getParameters()`	Returns a map of the `HttpServletRequest` parameters
`Map getSession()`	Returns a map of `HttpSession` values
`ValueStack getValueStack()`	Returns the OGNL value stack
`void put(Object key, Object value)`	Stores a specified value against a specified key in the current `ActionContext`
`void setActionInvocation(ActionInvocation actionInvocation)`	Sets the action invocation (the execution state)
`void setApplication(Map application)`	Sets the application context of the action
`static void setContext(ActionContext context)`	Sets the action context for the current thread
`void setContextMap(Map contextMap)`	Sets the action's context map
`void setConversionErrors(Map conversionErrors)`	Sets conversion errors occurred during the execution of the action
`void setLocale(Locale locale)`	Sets the Locale for the current action
`void setName(String name)`	Sets the name of the current Action in the `ActionContext class`
`void setParameters(Map parameters)`	Sets the action parameters
`void setSession(Map session)`	Sets a map of action session values
`void setValueStack(ValueStack stack)`	Sets the OGNL Value Stack

Another class that stores Web-specific context information for actions is `ServletActionContext` class, which is a subclass of `ActionContext`. In addition to extending fields and methods from ActionContext, the

ServletActionContext class also implements the `org.apache.struts2.StrutsStatics` interface. Table 20.10 describes the fields of the `ServletActionContext` class:

Table 20.10: Fields of the ServletActionContext Class

Field Name	Description
`static String ACTION_MAPPING`	Refers to the static final String type field having value struts.actionMapping
`private static long serialVersionUID`	Represents the serial version id of the `ServletActionContext` class and its value is -666854718275106687L
`static String STRUTS_VALUESTACK_KEY`	Represents the key for the associated value stack and its value is struts.valueStack

Let's now discuss the methods of the `ServletActionContext` class in Table 20.11:

Table 20.11: Methods of the ServletActionContext Class

Method	Description
`static ActionContext getActionContext (HttpServletRequest req)`	Returns the current action context
`static ActionMapping getActionMapping()`	Returns the action mapping for the current context
`static PageContext getPageContext()`	Returns the HTTP page context
`static HttpServletRequest getRequest()`	Returns the HTTP Servlet request object
`static HttpServletResponse getResponse()`	Returns the HTTP servlet response object
`static ServletContext getServletContext()`	Returns the Servlet context
`static ValueStack getValueStack (HttpServletRequest req)`	Returns the current value stack for this request
`static void setRequest (HttpServletRequest request)`	Sets the HTTP Servlet request object
`static void setResponse (HttpServletResponse response)`	Sets the HTTP Servlet response object
`static void setServletContext (ServletContext servletContext)`	Sets the current Servlet context object

Let's now discuss about POJO to create action classes.

POJO as Action

POJOs are Java objects that neither implement an interface nor extend a Java class. A POJO object is a pure Java object and does not depend on other APIs. The creation of POJO objects is simple; and therefore, better to design, debug, and test. Struts 2 provides the facility to use POJO as an action class. The methods of POJO action must agree with the contracts defined for them in the Struts 2 specification; for example the signature of the execute() method (). The following code snippet shows a POJO as an action class:

```
public class SomeAction{
public String execute() throws Exception {
return "success";
```

The importance of POJO as action is that we do not need to use extra objects in the Struts Framework. It is faster, simpler, and easier to develop. POJO organizes and manages the business logic, database communication, transactions, and database concurrency.

Implementing Actions in Struts 2

Let's create an application to learn how to implement actions in Struts 2. The application created in this subsection adds some users in the application context and the list of available users is maintained throughout the

running application. We can also edit and delete user information. To do this, we create JSP files and action classes, along with modifications in the configuration files, followed by the deployment and testing of our Struts 2 based Web application. Perform the following steps to create the Struts2App Web application:

- Setting Struts 2 environment
- Creating Home page
- Creating action class and JSP file
- Creating UserAction as ActionForm
- Creating Action class and JSP file for editing
- Exploring the directory structure of the application
- Packaging, Deploying, and Running the application

Now, let's study each of them in detail.

Setting Struts 2 Environment

Before getting started with developing Struts applications, you need to first prepare the development environment. This includes getting Struts 2 APIs in the form of JARs. The JAR files provide support to Web container for managing the Web application with the components of Struts framework.

To develop a Struts 2 based Web application, you need to install Struts 2 APIs. You can download the latest version of Struts 2 APIs from the Apache Web site (`http://struts.apache.org/2.x/`). The latest available release of Struts 2 is Struts 2.1.8. Download the binary distribution of Struts 2.1.8 on your computer system.

After you have installed the Struts 2.1.8 distribution, you can develop Web application using Struts 2 APIs, which are available in the form of JARs. Extract the Struts 2 archive (struts-2.1.8.1-all.zip) and save it on your local disk, say `D:\Struts2`. A directory named struts-2.1.8.1 will be created in `D:\Struts2`.

Other important subdirectories that can be found in the struts-2.1.8.1 directory are as follows:

- **D:\Struts2\struts-2.1.8.1\apps**—Contains .war files for sample Struts 2 applications.
- **D:\Struts2\struts-2.1.8.1\lib**—Contains all Struts 2 JAR files that are required for Struts 2 based Web applications. These JAR files contain Strut 2 API in the form of interfaces, classes, and some default configuration files.

Creating Home Page

Let's create a home page in the `Struts2App` application, index.jsp. This page simply contains some hyperlinks to run different JSP pages and actions.

Let's create a client view, `index.jsp` page, which acts as the home page of our application. Listing 20.1 shows the index.jsp page (you can find the index.jsp file on the CD in the code\JavaEE\Chapter20\Struts2App folder):

Listing 20.1: Displaying the Code of the `index.jsp` File

```
<%@ page contentType="text/html;charset=UTF-8" language="java"%>
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
  <head>
        <title>Struts 2 Actions<title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
  </head>
  <body>
        <center>
              <h2>
                    Struts 2 Actions
              </h2>
              <br>
              <br>
              Welcome
              <s:property value="#session.user" default="Guest" />!
              <s:if test="#session.user!=null">
                    <s:url id="logout" action="logout" />
                    | <s:a href="%{logout}">Logout</s:a> |
```

```
                </s:if>
                <br>
                <table cellspacing="5" width="180">
                        <tr bgcolor="#f0edd9" height="25" align="center">
                        <td>
                        <s:url id="hello" action="hello" />
                        <s:a href="%{hello}">Hello Action</s:a>
                        </td>
                        </tr>
                        <tr bgcolor="#f0edd9" height="25" align="center">
                        <td>
                        <s:a href="add_user.jsp">Add User</s:a>
                        </td>
                        </tr>
                        <tr bgcolor="#f0edd9" height="25" align="center">
                        <td>
                        <s:a href="user.jsp">View Users</s:a>
                        </td>
                        </tr>
                        <tr bgcolor="#f0edd9" height="25" align="center">

                        </tr>
                </table>
        </center>
    </body>
</html>
```

In Listing 20.1, the hyperlinks, such as Add User, View User, and Login, map to different JSP pages that are yet to be created. To make these hyperlinks work, we need to create different JSP pages, action classes, and other helping Java classes in the following sections. Save the index.jsp page in the root directory of the application.

Creating Action class and JSP file

Let's design some action classes for the `Struts2App` application. The first action class to be created is HelloAction. This action class is created by implementing the `com.opensymphony.xwork2.Action` interface. The action class can use string constants from the `Action` interface, such as SUCCESS and ERROR. Perform the following steps to create and configure the `HelloAction` class and also create the `hello.jsp` file:

- Creating the `HelloAction` class
- Configuring the `HelloAction` class
- Creating the `hello.jsp` file

Creating the HelloAction class

Let's create the HelloAction class that sets a string message field with the `Hello From Struts` value. Listing 20.2 shows the `HelloAction` class (you can find the HelloAction.java file on the CD in the `code\JavaEE\Chapter20\Struts2App\src\com\kogent\action` folder):

Listing 20.2: Displaying the Code of the HelloAction.java File

```
package com.kogent.action;

import com.opensymphony.xwork2.Action;

public class HelloAction implements Action {

  String message;

  public String getMessage() {
        return message;
  }

  public void setMessage(String message) {
       this.message = message;
  }
```

```
    public String execute() throws Exception {
        setMessage("Hello From Struts!");
        return SUCCESS;
    }
}
```

Compile the HelloAction.java file and place HelloAction.class file in the WEB-INF/classes/com/kogent/action folder, as shown in the directory structure (Figure 20.4). All the properties defined in an action are pushed into its value stack. The value of these properties can be accessed in the next JSP page to be based on the result of String returned from this action class. The action class must have getter methods for all properties to access their values.

Configuring the HelloAction Class

All action classes need to be configured in Struts configuration file struts.xml, which provides action mapping for the corresponding action class. Listing 20.3 shows the struts.xml file; in which HelloAction class is configured (you can find this file on CD in the code\JavaEE\Chapter20\StrutsApp\WEB-INF\ folder):

Listing 20.3: Displaying the Code of the struts.xml File

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd ">
<struts>
   <include file="struts-default.xml"/>
   <package name="my-default" extends="struts-default">
         <action name="hello" class="com.kogent.action.HelloAction" >
               <result name="success">hello.jsp</result>
         </action>
   </package>
</struts>
```

The action mapping for the `HelloAction` action class is provided in `struts.xml`. We can configure the list of Interceptors and results for a particular action. The request path containing hello.action invokes the `HelloAction` action class, and executes the `execute()` method.

Creating the hello.jsp file

The `HelloAction` class returns the String, such as success or failure. If the String returned is success, then the action class consequently invokes the hello page. The `org.apache.struts2.dispatcher.mapper.ActionMapper` interface provides the mapping between HTTP requests and action invocation requests. If the action invocation request matches, this returns an object `org.apache.struts2.dispatcher.mapper.ActionMapping`; otherwise, it returns null. The object of the ActionMapping class holds action mapping information that is used to invoke a Struts action.

Listing 20.4 shows the code of the hello.jsp file (you can find this file on CD in the `code\JavaEE\Chapter20\Struts2App` folder):

Listing 20.4: Displaying the Code of the hello.jsp Page

```
<%@ page contentType="text/html;charset=UTF-8" language="java"%>
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
   <head>
         <title>Struts 2 Actions</title>
         <link rel="stylesheet" href="mystyle.css" type="text/css" />
   </head>
   <body>
         <center>
               <h2>
               <s:property value="message" default="Struts Actions" />
               </h2>
               Welcome <s:property value="#session.user" default="Guest" />!
               <br><br>
               <s:if test="#session.user!=null">
                     <s:url id="logout" action="logout" />
                     | <s:a href="%{logout}">Logout</s:a> |
```

```
                </s:if>
                <br><br>| <s:a href="index.jsp">Back</s:a> |
            </center>
        </body>
    </html>
```

The hello.jsp simply displays a text message from the value stack available.

The <s:property value="message" default="Struts action" /> tag invokes the `getMessage()` method and obtains the message set by the `HelloAction` action. If not set, the default message is printed. Another `<s:property/>` tag displays the value of the user attribute from the session. If there is no user attributes in the session scope, the default `Guest` string is displayed.

Creating UserAction as ActionForm

Struts 2 eliminates the requirement of creating a separate ActionForm class, which is invoked when the user enters the input data. We can use the action fields as input properties by providing their setter and getter methods. For example, for a request parameter named `username`, we can declare a field `username` of String type in the action class with two methods—`void setUsername(String)` and `String getUsername()`. The action class field will automatically be populated with the data sent as the request parameter. This becomes possible only when the ParameterInterceptor is configured as one of the Interceptors for the action. Let's now design some JSPs and an action class to show how it works. Let's perform the following broad-level steps:

- Creating UserAction class
- Creating add_user.jsp file
- Creating user.jsp file

Now, let's discuss each of them in detail.

Creating UserAction Class

Let's create the UserAction action class, which extends the ActionSupport class. Listing 20.5 shows the code of the UserAction action class (you can find the UserAction.java file on CD in the `code\JavaEE\Chapter 20\Struts2App\src\com\kogent\action` folder):

Listing 20.5: Displaying the Code of the UserAction.java File

```
package com.kogent.action;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map;
import org.apache.struts2.interceptor.ApplicationAware;
import com.kogent.User;
import com.opensymphony.xwork2.ActionSupport;
public class UserAction extends ActionSupport implements ApplicationAware{
   String username;
   String password;
   String city;
   String email;
   String type;
   Map application;
   public void setApplication(Map application) {
         this.application=application;
   }
   public String getUsername() {
         return username;
   }
   public void setUsername(String username) {
         this.username = username;
   }
   public String getPassword() {
         return password;
   }
   public void setPassword(String password) {
         this.password = password;
   }
```

```
public String getCity() {
	return city;
}
public void setCity(String city) {
	this.city = city;
}
public String getEmail() {
	return email;
}
public void setEmail(String email) {
	this.email = email;
}
public String getType() {
	return type;
}
public void setType(String type) {
	this.type = type;
}
public String execute() throws Exception {
	ArrayList users=(ArrayList)application.get("users");
	if(users==null){
		users=new ArrayList();
	}
	if(getUser(username)==null){
		users.add(buildUser());
		application.put("users", users);
	}else{
		return ERROR;
	}
	return SUCCESS;
}
public User buildUser(){
	User user=new User();
	user.setUsername(username);
	user.setPassword(password);
	user.setCity(city);
	user.setEmail(email);
	user.setType(type);
	return user;
}
public User getUser(String username){
	User user=new User();
	boolean found=false;
	ArrayList users=(ArrayList)application.get("users");
	if(users!=null){
		Iterator it=users.iterator();
		while(it.hasNext()){
			user=(User)it.next();
			if(username.equals(user.getUsername())){
				found=true;
				break;
			}
		}
		if(found){
			return user;
		}
	}
	return null;
}
public void validate() {
if ( (username == null ) || (username.length() == 0) ) {
	this.addFieldError("username", getText("app.username.blank"));
}
if ( (password == null ) || (password.length() == 0) ) {
	this.addFieldError("password", getText("app.password.blank"));
```

```
        }
        if ( (email == null ) || (email.length() == 0) ) {
              this.addFieldError("email", getText("app.email.blank"));
        }
        }
    }
```

The `UserAction` class extends the `com.opensymphony.xwork2.ActionSupport` class and implements the `org.apache.struts2.interceptor.ApplicationAware` interface. The `ActionSupport` class itself implements different interfaces, such as Action, LocaleProvider, TextProvider, Validateable, and ValidationAware, and provides default implementations for the methods from these interfaces, which we can use in our action class.

The extending of the `ActionSupport` class enables you to use methods such as `addFieldError()`, `addActionError()`, and `getText()`. The implementation of the `ApplicationAware` interface uses an instance of the Application Map class, which can be used to store attributes within the application scope. The objects stored in the Application Map class are available in the whole application.

In the `UserAction` action class, the `execute()` method provides the implementation of the business logic to add a new user into an ArrayList, which is maintained in the application scope. The `getUser()` method searches for the user with the given username and returns an object of the User class. This class is a simple JavaBean that is used to group single user information. The `users` ArrayList in application scope basically contains the objects of the `User` class.

Listing 20.6 shows the code of the User.java file (you can find this file on CD in the `code\JavaEE\Chapter20\Struts2App\src\com\kogent` folder):

Listing 20.6: Displaying the Code of the User.java File

```
package com.kogent;
public class User {
   String username;
   String password;
   String city;
   String email;
   String type;
   public String getUsername() {
         return username;
   }
   public void setUsername(String username) {
         this.username = username;
   }
   public String getPassword() {
         return password;
   }
   public void setPassword(String password) {
         this.password = password;
   }
   public String getCity() {
         return city;
   }
   public void setCity(String city) {
         this.city = city;
   }
   public String getEmail() {
         return email;
   }
   public void setEmail(String email) {
         this.email = email;
   }
   public String getType() {
         return type;
   }
   public void setType(String type) {
         this.type = type;
   }
}
```

Another method of the `UserAction` action class is `buildUser()` that returns an object of the `User` class to be added into the `ArrayList` after populating it with the values of corresponding input properties. The `execute()` method makes sure that no two User objects with the same username are added into the application scoped ArrayList.

Now, add the action mapping shown in the following code snippet for the UserAction action class in the struts.xml file:

```
<struts>
  <include file="struts-default.xml"/>
  <package name="my-default" extends="struts-default">
  <action name="hello" . . . .>
        . . . .
  </action>
        <action name="adduser" class="com.kogent.action.UserAction" >
        <result name="input">add_user.jsp</result>
        <result name="error">add_user.jsp</result>
        <result name="success">user.jsp</result>
  </action>
  </package>
</struts>
```

Creating the add_user.jsp File

The two JSP files, add_user.jsp and user.jsp, are configured based on the results provided by the action class. The add_user.jsp page provides a form with five input fields with names matching the input properties defined in the UserAction class.

Listing 20.7 shows the code of the add_user.jsp page (you can find the add_user.jsp file on CD in the `code\JavaEE\Chapter20\Struts2App` folder):

Listing 20.7: Displaying the Code of the add_user.jsp File

```
<%@ page contentType="text/html;charset=UTF-8" language="java"%>
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
  <head>
        <title>Struts 2 Actions
        </title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
  </head>
  <body>
        <center>
              Adding User!
              <br><br>
              <table bgcolor="#f0edd9">
              <tr><td>
              <s:form action="adduser">
              <s:textfield name="username" label="User Name" />
              <s:password name="password" label="Password" size="15" />
              <s:textfield name="city" label="City" />
              <s:textfield name="email" label="E-Mail" />
              <s:select label="Type" name="type" headerKey="1" headerValue="Select
              Type" list="#@java.util.HashMap@{'Admin':'Admin', 'Client':'Client'}"
              ></s:select>
              <s:submit value="Add User" />
              </s:form>
              </td></tr>
              </table>
              <br>| <s:a href="index.jsp">Back</s:a> |
        </center>
  </body>
</html>
```

Listing 20.7 provides the action attribute of <s:form/> having the value adduser. This action attribute will use UserAction class for the processing of data entered into this form. We can click on "Add User" hyperlink, created in index.jsp, to display the add_user.jsp page.

Creating the user.jsp File

After entering the user information in the add_user.jsp page, click the Add User button to add a new user. If the username entered is unique, a new User object is added into an ArrayList, which is added into application scope. After the successful addition of a new user, the user.jsp page is displayed, which lists all the users added.

Listing 20.8 shows the code for user.jsp page (you can find the user.jsp file on CD in the `code\JavaEE\Chapter20\Struts2Action folder`):

Listing 20.8: Displaying the Code of the user.jsp File

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
  <head>
        <title>Struts 2 Actions</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
  </head>
  <body>
        <s:set name="users" value="#application.users"/>
        <table cellspacing="5" cellpadding="2" width="600">
              <tr bgcolor="#f0edd9">
                    <td>User Name</td>
                    <td>Password</td>
                    <td>City</td>
                    <td>Email</td>
                    <td>Type</td>
                    <td colspan="2"> </td>
              </tr>
              <s:iterator id="user" value="users">
              <tr>
                    <td><s:property value="username"/></td>
                    <td><s:property value="password"/></td>
                    <td><s:property value="city"/></td>
                    <td><s:property value="email"/></td>
                    <td><s:property value="type"/></td>
                    <td>
                    <s:url id="url" action="getuser">
                    <s:param name="username"><s:property value="username"/></s:param>
                    </s:url>
                    <s:a href="%{url}">Edit</s:a>
                    </td>
                    <td>
                    <s:url id="url" action="delete">
                    <s:param name="username"><s:property value="username"/></s:param>
                    </s:url>
                    <s:a href="%{url}">Delete</s:a>
                    </td>
              </tr>
              </s:iterator>
        </table>
<br>| <s:a href="index.jsp">Home</s:a> | <s:a href="add_user.jsp">Add More</s:a>
  </body>
</html>
```

Listing 20.8 uses the <s:iterator/> tag to iterate over the ArrayList and displays the values of all fields set for each user.

The user.jsp file contains two hyperlinks, Edit and Delete, rendered in front of each user record. These hyperlinks are not functional yet; therefore, we need to create action classes and JSP pages to make these two hyperlinks perform the specified functions.

Creating Action Class and JSP file for Editing User Data

Let's now create the `GetUserAction` class, which is invoked when the client clicks the `Edit` hyperlink of the index.jsp page. The `GetUserAction` class executes the `edit_user.jsp` page. Perform the following steps to create and configure the `GetUserAction` class as well as create the `edit_user.jsp` file:

- Creating the GetUserAction class
- Configuring the GetUserAction class
- Creating the edit_user.jsp file
- Configuring Action Class methods

Creating the GetUserAction Class

The `GetUserAction` class implements the `Action`, `ServletRequestAware`, and `ApplicationAware` interfaces. In the GetUserAction class, the `setServletRequest(HttpServletRequest)` method of the `ServletRequestAware` interface is used to set the `HttpServletRequest` object to be used in the action.

The `execute()` method of the GetUserAction action class simply obtains a User object having username matching with the username passed as the request parameter to this action. The `HttpServletRequest` object is used to get this username parameter as well as to save the obtained `User` object into the request scope.

Listing 20.9 shows the code of the GetUserAction action class (you can find the GetUserAction.java file on CD in the `code\JavaEE\Chapter20\Struts2App\src\com\kogent\action` folder):

Listing 20.9: Displaying the Code of the GetUserAction.java File

```
package com.kogent.action;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;

import com.opensymphony.xwork2.Action;
import org.apache.struts2.interceptor.ApplicationAware;
import org.apache.struts2.interceptor.ServletRequestAware;
import com.kogent.User;
public class GetUserAction implements
  Action, ServletRequestAware, ApplicationAware {
  HttpServletRequest request;
  Map application;
  public void setServletRequest(HttpServletRequest request) {
        this.request = request;
  }
  public void setApplication(Map application) {
        this.application = application;
  }
  public String execute() throws Exception {
        String username=request.getParameter("username");
        ArrayList users=(ArrayList)application.get("users");
        User user;
        if(users!=null){
              Iterator it=users.iterator();
              while(it.hasNext()){
                    user=(User)it.next();
                    if(username.equals(user.getUsername())){
                          request.setAttribute("user", user);
                          break;
                    }
              }
        }
        return SUCCESS;
  }
}
```

Configuring the GetUserAction class

Configure the `GetUserAction` class by adding new action mapping into the struts.xml file. The following code snippet provides the new action mapping for the `GetUserAction` class:

```
<action name="getuser" class="com.kogent.action.GetUserAction" >
  <result name="success">edit_user.jsp</result>
</action>
```

In the preceding code snippet, if the result obtained from the GetUserAction class is successful, the edit_user.jsp page is invoked.

Creating the edit_user.jsp file

The Edit hyperlink invokes the GetUserAction class and passes a request parameter named username having different values for different user records. The execution of the GetUserAction class gives edit_user.jsp. Listing 20.10 shows the code of the edit_user.jsp page (you can find the edit_user.jsp file on CD in the code\JavaEE\Chapter20\Struts2App folder):

Listing 20.10: Displaying the Code of the edit_user.jsp File

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="/struts-tags" prefix="s" %>
<html>
  <head>
        <title>Struts 2 Actions</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
  </head>
  <body>
        <center>
                Editing User!<br><br>
                <table bgcolor="#f0edd9">
                <tr>
                        <td>
                        <s:form action="edit">
                        <s:textfield name="username" label="User Name" readonly="true">
                        <s:param name="value"><s:property
                        value="#request.user.username"/></s:param>
                        </s:textfield>
                        <s:textfield name="password" label="Password" size="15">
                        <s:param name="value"><s:property
                        value="#request.user.password"/></s:param>
                        </s:textfield>
                        <s:textfield name="city" label="City">
                        <s:param name="value"><s:property
                        value="#request.user.city"/></s:param>
                        </s:textfield>
                        <s:textfield name="email" label="E-Mail">
                        <s:param name="value"><s:property
                        value="#request.user.email"/></s:param>
                        </s:textfield>
                        <s:select label="Type" name="type" headerKey="1"
                        headerValue="Select Type" list="#@java.util.HashMap@{'Admin':
                        'Admin', 'Client':'Client'}" ></s:select>
                        <s:param name="value"><s:property
                        value="#request.user.type"/></s:param>
                                <s:submit value="Edit"/>
                                </s:form>
                        </td>
                </tr>
                </table>
                <br>| <s:a href="index.jsp">Home</s:a> |
        </center>
  </body>
</html>
```

The edit_user JSP page creates a form similar to add_user.jsp. However, the fields in the edit_user JSP page are populated with the values of different fields of the User object. This User object is set into request scope by the GetUserAction class, according to the username passed as the request parameter when the Edit hyperlink is clicked.

Configuring Action Class Methods

To edit the user record, a client needs to click the Edit hyperlink. However, prior to that, the developer needs to decide which action will be performed to handle this edit action form. The edit _user.jsp page is similar to the add_user.jsp page; therefore, we need the same set of input properties to be set as that of the UserAction action class. Similarly, the validation logic for these fields will be same as that implemented in the validate() method of UserAction. Therefore, when all these things are similar and can be reused, there is no need for creating a new action class here.

Struts 2 allows configuration of any method, having signature String methodName(), which can work similar to the execute() method of action class.

Add a new method into your existing UserAction class. This method contains the logic to edit the users. The logic is to replace the existing User object in the ArrayList with the new `User` object created with new data. The following code snippet shows the code of the edit() method of the UserAction class:

```
public class UserAction extends ActionSupport implements ApplicationAware{
String username;
String password;
String city;
String email;
String type;
Map application;
public void setApplication(Map application) {
  this.application=application;
}
//Setter and getter methods..
. . .
. . .
public String execute() throws Exception {
  . . . . .
  return SUCCESS;
  }
  //New method added.
  public String edit() throws Exception{
        ArrayList users=(ArrayList)application.get("users");
        User user=null;
        int index=0;
        Iterator it=users.iterator();
        while(it.hasNext()){
                user=(User)it.next();
                if(user.getUsername().equals(username)){
                        break;
                }
                index++;
        }
        User newuser=buildUser();
        users.set(index, newuser);
        application.put("users", users);
        return SUCCESS;
  }


  public User buildUser(){
  }
  public User getUser(String username){
  }
}
```

Recompile the `UserAction` action class and place the `UserAction.class` file in the `WEB-INF\classes\com\kogent\action` folder. The `<s:form/>` tag used in `edit_user.jsp` page is set with the value edit. Add a new action mapping with the name edit, which invokes the `edit()` method of the `UserAction` class to edit the user information. This is done by appending the code for mapping the edit() method in the struts.xml file, as shown in the following code snippet:

```
<action name="edit" class="com.kogent.action.UserAction" method="edit" >
  <result name="input">edit_user.jsp</result>
  <result name="success">user.jsp</result>
</action>
```

Now, the client can click on the Edit hyperlink, which invokes the `edit()` method of the `UserAction` class.

Similarly, to delete the user record, we need to click the Delete hyperlink. The action generated by the Delete hyperlink is handled by the `deleteUser()` method, which needs to be appended to the existing code of the

UserAction.java file. The `deleteUser()` method removes the matching user object from the ArrayList object. The following code snippet shows the deleteUser() method of the UserAction class:

```
public class UserAction extends ActionSupport implements ApplicationAware{
String username;
String password;
String city;
String email;
String type;
Map application;
public void setApplication(Map application) {
this.application=application;
}
//Setter and getter methods..
. . .
. . .
public String execute() throws Exception {
. . . . .
return SUCCESS;
}
//New method added.
public String deleteUser() throws Exception{

        ArrayList users=(ArrayList)application.get("users");
        User user=null;
        int index=0;
        Iterator it=users.iterator();
        while(it.hasNext()){
                user=(User)it.next();
                if(user.getUsername().equals(username)){
                        break;
                }

                index++;
        }
        users.remove(index);
        application.put("users", users);
        return SUCCESS;
  }

}
```

Add a new action mapping with the name delete, which invokes the deleteUser() method of the UserAction class to delete the user information. This is done by appending the code for mapping the deleteUser() method in the struts.xml file, as shown in the following code snippet:

```
<action name="delete" class="com.kogent.action.UserAction" method="deleteUser" >
  <interceptor-ref name="basicStack"/>
  <result name="input">edit_user.jsp</result>
  <result name="success">user.jsp</result>
</action>
```

The mapping shown in the preceding code snippet invokes the `deleteUser()` method of the `UserAction` action class, instead of executing its execute() method, as the method configured in the preceding code snippet is `deleteUser`. Then the validate() method is executed, as username is the only request parameter passed to the action.

You now need to configure the Web application according to the components within the directory structure. To do this, you must make modifications in the web.xml file, as shown in Listing 20.11 (you can find the web.xml file on the CD in the `code/JavaEE/Chapter20/Struts2App/WEB-INF` folder):

Listing 20.11: Displaying the Code of the web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <display-name>Struts 2 Actions</display-name>
  <filter>
        <filter-name>struts2</filter-name>
```

```
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher
      </filter-class>
      </filter>
      <filter-mapping>
            <filter-name>struts2</filter-name>
            <url-pattern>/*</url-pattern>
      </filter-mapping>
      <welcome-file-list>
            <welcome-file>index.jsp</welcome-file>
      </welcome-file-list>
    </web-app>
```

Let's now explore the directory structure of the application.

Creating Directory Structure

We now need to create a sample Web application called Struts2App, according to the directory structure shown in Figure 20.4:

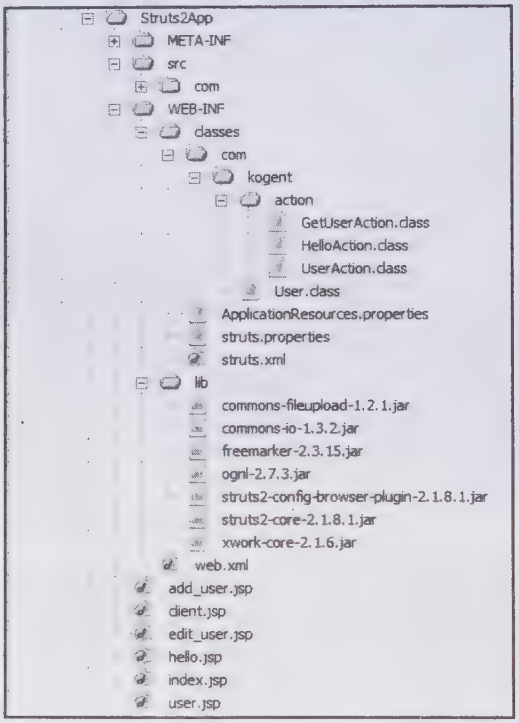


Figure 20.4: Displaying the Directory Structure of Struts2App

Table 20.12 describes the directory structure of the `Struts2App` Web application, and a list of files that should exist in this directory:

Table 20.12: The Directory Structure of Struts2App

Directory	Contains
`/Struts2App`	Serves as the root directory of the Web application. All the JSP and HTML files are stored in this directory. You can also put JSP and HTML pages in separate folders under this directory.
`/Struts2App/WEB-INF`	Contains all resources used in the application. The web.xml file, which contains the configuration/deployment details of the Web application, must exist in this directory.

Table 20.12: The Directory Structure of Struts2App	
Directory	**Contains**
`/Struts2App/WEB-INF/classes`	Contains Java class file, Struts 2 action class files, and other utility classes. For developing this application, the struts.xml, struts.properties, ApplicationResources.properties and .class files (used in the application) must exist in this directory.
`/Struts2App/WEB-INF/lib`	Contains JAR files to which the components of the Web application are dependent.
`/Struts2App/WEB-INF/src`	Contains the source code, which is used to develop the Web application.

Copy the following Struts 2 JARs from struts-2.1.8.1\lib for this application into your `D:\code\JavaEE\Chapter20\Struts2App\WEB-INF\lib` directory:

- commons-fileupload-1.2.1.jar
- commons-io-1.3.2.jar
- freemarker-2.3.15.jar
- ognl-2.7.3.jar
- struts2-config-browser-plugin-2.1.8.1.jar
- struts2-core-2.1.8.1.jar
- xwork-core-2.1.6.jar

Make sure that all JSPs, class files, and JARs are placed according to the directory structure shown in Figure 20.4.

Packaging, Deploying, and Running the Application

We will now create Struts2App.war and deploy this Web application on the Glassfish V3 application server. You should ensure that the index.jsp page is configured as the welcome page in the web.xml file of the Struts2App application. Browse `http://localhost:8080/Struts2App` URL to run the application, as shown in Figure 20.5:

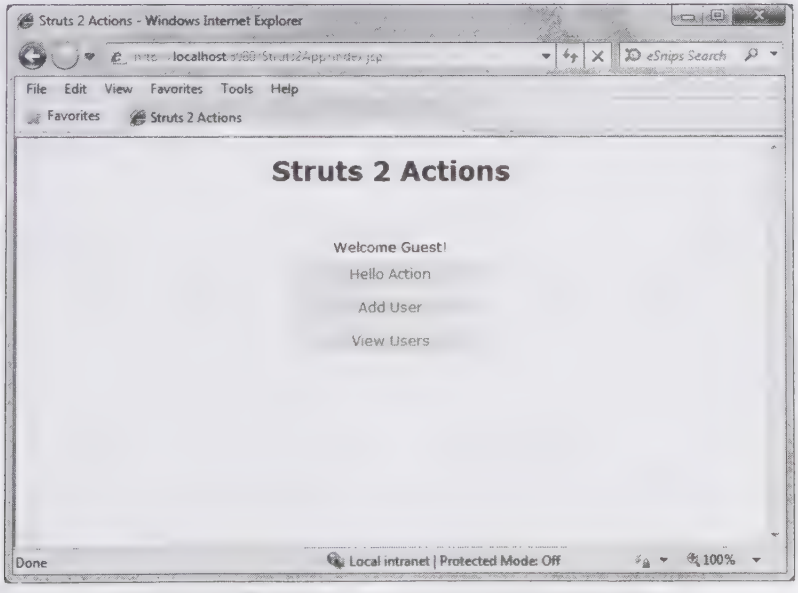


Figure 20.5: Displaying the Output of the index.jsp Page

In Figure 20.5, the hyperlink Hello Action creates a request for `hello.action` and invokes the action named hello configured in the struts.xml. As a result, the hello.jsp page appears, which has been configured to display the result for this action. The output of the hello.jsp page is shown in Figure 20.6:

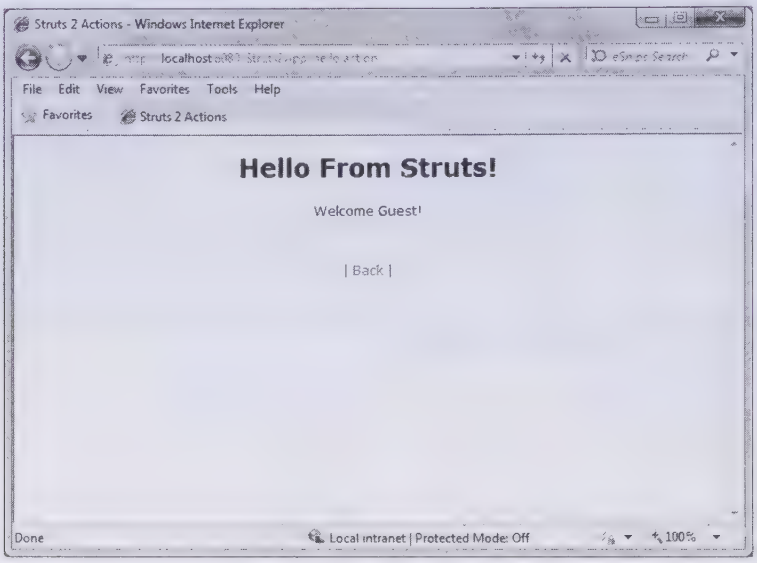


Figure 20.6: Showing the hello.jsp Page with a Message

Now, click the `Back` link to go to the `index.jsp` page. You should note that the hello.jsp page is displayed as hello.action depending upon the mapping of the action done in the web.xml file of the application. Then, click the `Add User` link, as shown in index.jsp (Figure 20.5). As a result, the `add_user.jsp` page appears. The output of `add_user.jsp` page is shown in Figure 20.7:

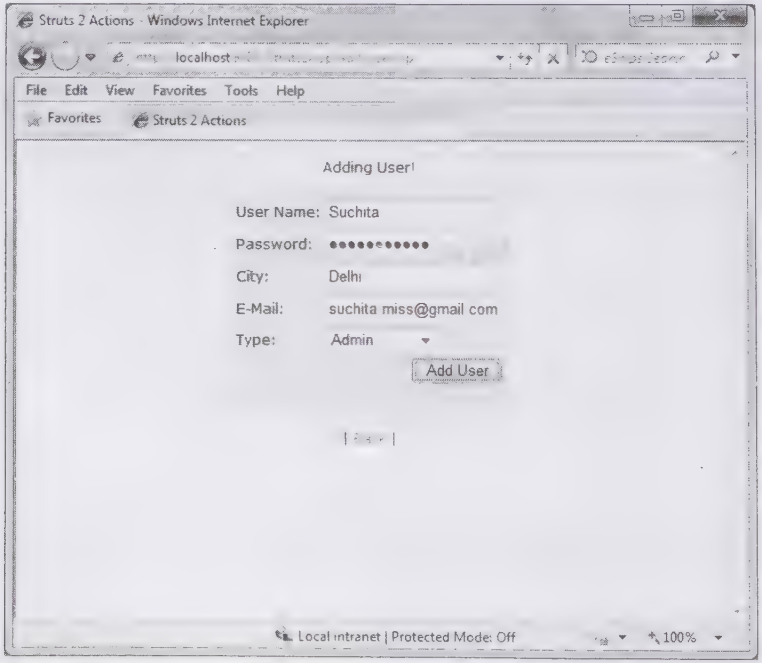


Figure 20.7: Showing the Output of the add_user.jsp Page with User Details

Enter the required data into the fields shown in Figure 20.7 and click the Add User button to add a new user. In our case, we have also added the Pallavi user. If the username entered is unique, a new User object is added into the ArrayList of the application scope. The output is shown in Figure 20.8:

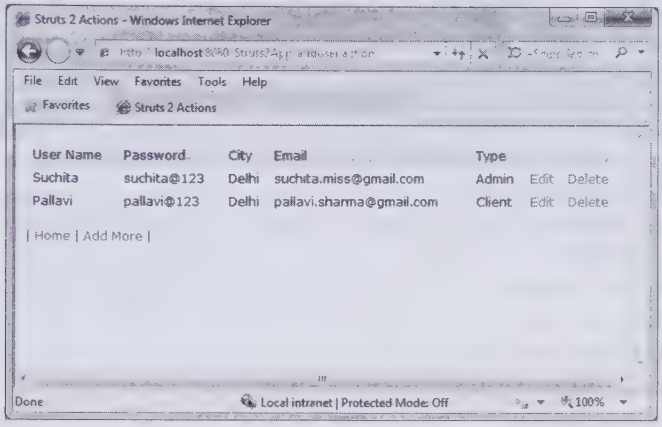


Figure 20.8: Displaying the Output of the user.jsp Page with a List of Users Added

In Figure 20.8, each record has two hyperlinks, `Edit` and `Delete`. When you click the `Edit` hyperlink, the `edit_user.jsp` page is displayed. The `Edit` hyperlink invokes the `GetUserAction` class and passes the username request parameter containing user records. The successful execution of the `GetUserAction` class invokes the `edit_user.jsp` page, as shown in Figure 20.9:

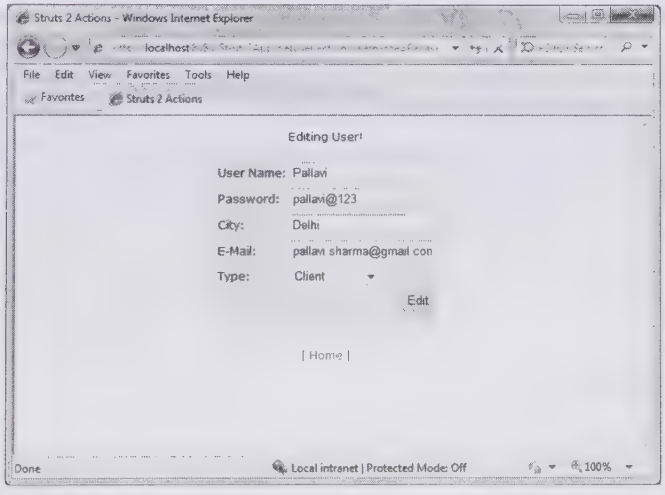


Figure 20.9: Displaying the Output of the edit_user.jsp Page

Similarly, to delete the user information, click the Delete hyperlink shown in Figure 20.8.

Let's now explore the concept of dependency injection and inversion of control in the Struts2 framework.

Dependency Injection and Inversion of Control

DI and IoC are programming design patterns used to reduce coupling in programs. When you use DI, you do not need to create objects, as DI only describes how to create objects, and the objects are automatically created. This task is accomplished using a factory known as `ObjectFactory` Interface. The `ObjectFactory` interface allows you to create objects of specific types.

The IoC provides a way to inject logic into the client code rather than writing it.

There are two ways to implement IoC in Struts:

- **With instantiation** – Allows you to instantiate a given action object with the resource object as a constructor parameter.

- **Using an enabler interface**— Allows a resource to be passed to the specified action object after the object is instantiated. In this implementation, the action class implements an interface with some methods, such as `setResources(ResourceObject r)`.

To use HTTP-specific objects in Struts action classes, Struts 2 uses DI and IoC. While using these techniques, the `Aware` interfaces are injected in an action class. These interfaces are called `Aware` interfaces because the interface names always end with aware. These interfaces have methods to set specific resources into the implementing class and to make the resource available.

Struts 2 action classes implement Injection interface, which is a form of DI pattern. The Common aware interfaces that Struts 2 supports are as follows:

- The ApplicationAware interface
- The ParameterAware interface
- The ServletRequestAware interface
- The ServletResponseAware interface
- The SessionAware interface

The ApplicationAware Interface

The `org.apache.struts2.interceptor.ApplicationAware` interface is used to expose a method to an action class. This method sets an Application Map object in this action class. This Map object contains different objects that are stored in the application scope. The key/value pairs added into this Map can now be accessed similar to other attributes of the application scope. The `ApplicationAware` interface has one method, `setApplication()`, which sets the map of application properties in implementing class.

The code for action that implements the `ApplicationAware` interface is provided in Listing 20.12:

Listing 20.12: Displaying the Code for an Action Implementing the ApplicationAware Interface

```
package com.kogent.action;

import java.util.Map;
import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ApplicationAware;

public class SomeAction extends ActionSupport implements ApplicationAware {

  Map app;

  public void setApplication(Map app) {
        this.app=app;
  }
  public String execute() throws Exception {
        app.put("company", "Kogent Solutions Inc.");
        return SUCCESS;
  }
}
```

The Interceptor places the Map application into the SomeAction action class before it is executed.

All key/value pairs stored in Application Map can be accessed from any action or JSP page. The Struts 2 tags support the display of different application scope objects using a corresponding key. For example, the single key/value pair stored in Application Map app, shown in Listing 20.12, can be accessed in a JSP page, as shown in the following code snippet:

```
<s:property value="#application.name"/>
or
<s:property value="#application['name']"/>
```

The ParameterAware Interface

The `org.apache.struts2.interceptor.ParameterAware` interface is used when an action class handles input parameters. All the input parameters, with their name/value pairs, are set in a parameter map. When the

actions require the HTTP request parameter map, this interface is implemented within an action. Another common use of this interface is to use parameters to internally instantiate data objects.

The `setParameter(Map map)method of the ParameterAware interface` sets the map of input parameters in an action class. An action that implements the `ParameterAware` interface is shown in Listing 20.13:

Listing 20.13: Displaying the Code for an Action Implementing the ParameterAware Interface

```
package com.kogent.action;

import java.util.Map;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ParameterAware;

public class SomeAction extends ActionSupport implements ParameterAware {
  Map params;

    public void setParameters(Map params) {
          this.params=params; }
    public String execute() throws Exception {
          return SUCCESS;
    }
}
```

In Listing 20.13, params is a Map containing all input parameters set as the key/value pairs. The action class can use this map to process input parameters or to propagate them to another object. These input parameters can also be displayed on JSP using the following syntax, which displays two input parameters, name and city:

```
<s:property value="#parameters.name"/>
<s:property value="#parameters.city"/>
or
<s:property value="#parameters['name']"/>
<s:property value="#parameters['city']"/>
```

The ServletRequestAware Interface

The HttpServletRequest object is not available in the action class by default. However, this can be injected into the action class by implementing the org.apache.struts2.interceptor.ServletRequestAware interface. The exposed method setServletRequest(HttpServletRequest request) sets the HttpServletRequest object in the action class. This allows an action to use the HttpServletRequest object in a Servlet environment. Listing 20.14 shows an action that implements the ServletRequestAware interface:

Listing 20.14: Displaying the Code for an Action Implementing the ServletRequestAware Interface

```
package com.kogent.action;

import java.util.Map;

import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ServletRequestAware;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class SomeAction extends ActionSupport implements ServletRequestAware{
  HttpServletRequest request;
  public void setServletRequest(HttpServletRequest request) {
        this.request=request;   }
  public String execute() throws Exception {
        request.setAttribute("userid", 101);
        HttpSession session=request.getSession();
        return SUCCESS;
  }
}
```

The ServletResponseAware Interface

The org.apache.struts2.interceptor.ServletResponseAware interface is similar to the ServletRequestAware interface. This interface is used to inject the HttpServletResponse object into an action. This interface has the setServletResponse(HttpServletResponse response) method, which sets the HttpServletResponse object. The following code snippet shows the code for an action that implements the ServletResponseAware interface and gets an HttpServletResponse object:

```
package com.kogent.action;
import java.util.Map;
import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.ServletResponseAware;
import javax.servlet.http.HttpServletResponse;
public class SomeAction extends ActionSupport implements ServletResponseAware {
  HttpServletResponse response;
  public void setServletResponse(HttpServletResponse response) {
        this.response=response;
  }
  public String execute() throws Exception {
        response.setContentType("text/html");
        int buffer=response.getBufferSize();
        . . . .
        . . . .
        return SUCCESS;
  }
}
```

The SessionAware Interface

The `org.apache.struts2.interceptor.SessionAware` interface is used to handle client sessions within an action class. The `SessionAware` interface provides the `setSession()` method to set the map of session attributes while implementing an action class. The action class implementing the `SessionAware` interface can access session attributes in the form of a Session map. We can add, remove, and obtain different session attributes by manipulating this map. Listing 20.15 shows an action class implementing the `SessionAware` interface:

Listing 20.15: Displaying the Code for an Action Implementing the SessionAware Interface

```
package com.kogent.action;
import java.util.Map;
import com.opensymphony.xwork2.ActionSupport;
import org.apache.struts2.interceptor.SessionAware;
public class SomeAction extends ActionSupport implements SessionAware {
  Map session;
  public void setSession(Map session) {
        this.session=session;
  }
  public String execute() throws Exception {
        session.put("user", "John");
        return SUCCESS;
  }
}
```

The single key/value pair stored in session Map session, shown in Listing 20.15, can be accessed in a JSP page, as shown in the following code snippet:

```
<s:property value="#session.user"/>
or
<s:property value="#session['user']"/>
```

Preprocessing with Interceptors

The Interceptor class allows you to define the piece of code that can be executed before or after the execution of an action class. Interceptors also have the ability to prevent an action from being executed. Interceptors provide developers a way to encapsulate common functionality in a reusable form, which can then be applied to one or more action classes. These common functionalities may include validation of input data, pre-processing of file

upload, protection from double submit, and sometimes pre invocation of controls with some data before a Web page is displayed. The Interceptor helps in modularizing common code into reusable classes.

The Struts 2 framework provides a set of Interceptors that can be used to provide the required functionalities to an action class. Interceptor or a stack of Interceptors is configured and executed before the action is executed, to provide all pre-processing of the request. Similarly, these configured interceptors are again executed after the execution of action to provide additional processing, if any. Let's now discuss the basic concept of interceptors, their configuration, and implementation in the Struts 2 Framework.

What are Interceptors?

Interceptor is a class that contains business logic implementation to provide a specific functionality. Interceptors provide additional processing information before the execution of an action class.

Every interceptor is pluggable, and the required interceptor or interceptor stack can be configured on a per-action basis. The separation of core functionality code in the form of interceptors eliminates the chances of implementing an extra code and references in an action class. The purpose of using interceptors is to allow greater control over controller layer and separate some common logic that applies to multiple actions.

An Interceptor class can be created by implementing the `com.opensymphony.xwork2.interceptor.Interceptor` interface or by extending `com.opensymphony.xwork2.interceptor.AbstractInterceptor` class. An intercept(ActionInvocation inv) method is associated with an Interceptor class to perform some processing on the request before and/or after rest of the processing of the request by the `ActionInvocation` interface, which stores the state of an action.

Interceptors as RequestProcessor

In Struts 2, interceptors have been replaced with the `RequestProcessor` interface, which was used in Struts 1 for processing the requests. Similar to `RequestProcessor`, interceptors provide all basic pre-processing required before executing an action class. Similar to the `RequestProcessor` class, we now have Interceptors that contain common logic to be applied to a number of actions. Only a small percentage of Struts 2 applications need to define their custom Interceptors for specific functionality.

How to Configure Interceptors?

Interceptors to be used in an application must be declared in the `struts.xml` file. Interceptor classes can be defined using a name-class pair specified in the Struts configuration file. The other approach is to include another `.xml` file containing the configuration of interceptors into the `struts.xml` file. All the interceptors, which are required to pre-process the request for an action class, should be defined in the action mapping provided for that specified action class in the struts.xml file.

For example, let's assume that we have two custom interceptor classes, `Interceptor1_class_name.class`, and `Interceptor2_class_name.class`. These two Interceptor classes are defined in the `struts.xml` file using the `<interceptor>` element, as shown in the following code snippet:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.1//EN"
"http://struts.apache.org/dtds/struts-2.1.dtd ">
<struts>
   <include file="struts-default.xml"/>
   <package name="default" extends="struts-default">
   <interceptors>
         <interceptor name="interceptor1" class="Interceptor1_class_name"/>
         <interceptor name="interceptor2" class="Interceptor2_class_name "/>
   </interceptors>
   <action name="login" class="LoginAction">
         <interceptor-ref name="interceptor1"/>
         <interceptor-ref name="interceptor2"/>

         <result name="input">login.jsp</result>
         <result name="success">success.jsp</result>
   </action>
```

```
    </package>
</struts>
```

In the preceding code snippet, interceptors, such as Interceptor1_class_name and Interceptor2_class_name, are configured in the struts.xml file with the help of the <interceptor> element. The <interceptor> element uses an attribute called name to provide a simple name for an interceptor class specified in the `class` attribute. In the preceding code snippet, the `name` attribute is set to values, such as interceptor1 and interceptor2, representing interceptor classes, such as Interceptor1_class_name and Interceptor2_class_name, respectively.

Further, a list of interceptor classes used to intercept the action request is declared in action mapping using the `<interceptor-ref name=". . ."> element. The interceptors are executed in the same order in which they are declared in action mapping. The corresponding interceptor classes ar executed to provide all pre-processing before the execution of an action class for which the interceptors are defined.

Stacking of Interceptors

Stacking of interceptors refers to the grouping of a set of interceptors. Stacking of interceptors is important as we may need to apply the same set of interceptors for any number of action classes. Therefore, instead of writing the whole list of interceptors in every action mapping repeatedly, we can just write the name of the interceptor stack. In most of the Struts 2 applications, we do not need to create our custom Interceptors as all the required pre-processing is provided by the default Interceptor stack, which is declared in the `struts-default.xml` file. An Interceptor stack is a set of Interceptors configured in a specific order. Therefore, the default Interceptor stack can be taken as the `RequestProcessor` interface in Struts 2. The action mapping provided in the following code snippet uses a single `<interceptor-ref>` element to declare interceptor stack, instead of using separate `<interceptor-ref>` elements for each interceptor class:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.1//EN"
"http://struts.apache.org/dtds/struts-2.1.dtd ">
<struts>
  <include file="struts-default.xml"/>
  <package name="default" extends="struts-default">
  <interceptors>
        <interceptor name="interceptor1" class="Interceptor1_class_name"/>
        <interceptor name="interceptor2" class="Interceptor2_class_name "/>
              <interceptor-stack name="custom_stack">
              <interceptor-ref name="interceptor1"/>
              <interceptor-ref name="interceptor2"/>
        </interceptor-stack>
  </interceptors>
  <action name="login" class="LoginAction">
        <interceptor-ref name="custom_stack"/>
        <result name="input">login.jsp</result>
        <result name="success">success.jsp</result>
  </action>
  </package>
</struts>
```

In preceding code snippet, we have defined two interceptors, and an interceptor stack (`custom_stack`) containing these two interceptors. Interceptor stack is defined using the `<interceptor-stack>` element.

The name attribute of `<interceptor-ref>` can take the name of an interceptor or an interceptor stack.

Bundled Interceptors

Bundled interceptors are a set of pre-defined interceptors, which are used in a Web application to provide required processing before and after an action class is executed. These pre-defined interceptors are known as bundled interceptors since they are bundled inside the Struts 2 Framework and are defined in the `struts-default.xml` file. Each of these pre-defined interceptors is designed to implement logic for a specific function, which is common to almost all the Web applications. Interceptors can be seen as a reusable component, which is commonly used in different application and different `action` invocations in an application. To use these bundled interceptors in our application, we can extend the `struts-default` package or define these bundled

interceptors in our package with name-class pair specified in the `<interceptor>` element. The noteworthy bundled interceptors available in the Struts 2 Framework are described in Table 20.13:

Table 20.13: Noteworthy Bundled Interceptors of the Struts 2 Framework

Name of the Interceptor	Description
alias	Converts the parameters that are named differently but work similar to each other between requests
chain	Provides the properties of the previous action for the current action
checkbox	Allows you to use a hidden field to add a checkbox automatically to detect the unsubmitted checkboxes
conversionError	Allows you to add the errors related to conversion of values from the ActionContext to the field defined for an action
createSession	Allows you to create an HttpSession automatically to facilitate the working of the interceptors that require the HttpSession
Exception	Allows you to map an execption to a specific result
execAndWait	Allows an action to be executed in the background and then forwards the request of a user to the page displaying the progress of the background process
fileUpload	Provides support of uploading a file at the specified target
i18n	Holds the details of the locale selected for the session of a user
model-driven	Pushes the result of the getModel () method to the Value Stack
prepare	Invokes the prepare() method
roles	Verifies the JAAS role of a user, and if the role is correct, the required action is executed
scope	Stores the state of an action in either session or application scope
scoped-model-driven	Retrieves the models from a scope and sets it on the action invoking the setModel() method
servletConfig	Allows you to acess the Maps representing HttpServletRequest and HttpSeervletResponse instances
staticParams	Sets the parameters defined in the struts.xml file to the action

In addition to these bundled interceptors, Struts 2 framework also allows you to create customized interceptors. Let's now discuss about writing our own interceptors, or customized interceptors.

Writing Interceptors

Struts 2 framework provides the flexibility to create our own interceptor classes that provide additional logic in a reusable form, which can be applied to one or more action classes.

When customized interceptor classes are created, they are declared in the struts.xml file of the Web application. Further, we can define a default interceptor stack, including the newly designed interceptor, to use it with the action classes.

The main method in every interceptor is its `intercept (ActionInvocation invocation)` method, which takes a reference of the `com.opensymphony.xwork2.ActionInvocation` interface. The `ActionInvocation` interface represents the execution state of the action. The `ActionInvocation interface` holds the instances of all the interceptors and action classes.

The processing of `ActionInvocation` is performed in steps, and every step is invoked by the `ActionInvocation.invoke()` method. First, the `invoke()method` is called on ActionInvocation by the ActionProxy factory and then, each Interceptor calls the invoke() method to execute the next Interceptor.

Interceptors can also quit the execution of `ActionInvocation` and return a code piece, such as `Action.SUCCESS`, or it may choose to do some processing before or after delegating the rest of the processing by using `ActionInvocation.invoke()`.

The action class is executed only after all the interceptors defined for an action class are executed.

An interceptor class can access various resources by invoking methods on the `ActionInvocation` object, which is passed as an argument to its `intercept()` method. The methods of ActionInvocation are listed in Table 20.14:

Table 20.14: Methods of ActionInvocation Interface

Method	Description
Object getAction()	Returns the action associated with the ActionInvocation interface.
ActionContext getInvocationContext()	Returns the instance of ActionContext class associated with the ActionInvocation interface.
ActionProxy getProxy()	Returns the ActionProxy holding the ActionInvocation interface.
ValueStack getStack()	Returns an object of ValueStack, which represents the value stack of the action.
String invoke()	Invokes the next step in the processing of the ActionInvocation interface. This may be the execution of next Interceptor or the action itself.
Result getResult()	Returns a non-chain result from a chain of ActionChainResults, if the object of the ActionInvocation interface has been executed earlier.
String getResultCode()	Returns the code returned from the current ActionInvocation.

The `ActionContext` returned by `getInvocationContext()` method can further provide details for `HttpServletRequest` and `HttpSession`. The following code snippet shows the sample `intercept()` method of an interceptor:

```
package com.kogent.interceptors;
public class SimpleInterceptor extends AbstractInterceptor
{
        public String intercept (ActionInvocation invocation) throws Exception {
        final ActionContext context = invocation.getInvocationContext();
        HttpServletRequest request = (HttpServletRequest) context.get(HTTP_REQUEST);
        HttpSession session = request.getSession (true);
        Object user = session.getAttribute (USER_HANDLE);
        if (user == null) {
                String loginAttempt = request.getParameter(LOGIN_ATTEMPT);
                if (! StringUtils.isBlank (loginAttempt) ) {
                if (some condition ) {
                        return "success";
                }
                else {
                        Object action = invocation.getAction ();
                        if (action instanceof ValidationAware) {
                        ((ValidationAware) action).addActionError("Username
                        or password incorrect.");
                        }
                }
        }
        // Either the login attempt failed or the user hasn't tried to login yet,
        // and we need to send the login form.
        return "login";
        } else {
                return invocation.invoke ();
        }
  }
}
```

After creating an interceptor, the mapping of this custom interceptor is provided in the struts.xml to configure this custom interceptor so that it can be used in the interceptor stacks defined in the action mappings provided in the struts.xml file. The sample configuration of a custom interceptor is shown in the following code snippet:

```
<struts>
   <package name="my-default" extends="struts-default">
      <interceptors>
<interceptor name="login" class=" com.kogent.interceptors.SimpleInterceptor " />
      </interceptors>
   </package>
</struts>
```

In the preceding code snippet, a custom interceptor, called the `SimpleInterceptor` class, is declared in the struts.xml file to intercept action classes. In the preceding code snippet, the `class` attribute of the `<interceptor>` element is used to specify a path of an interceptor class. The `name` attribute of the `<interceptor>` element specifies a name to the current interceptor class.

Let's now move on to discuss Object Graph Navigation Language (OGNL) support in Struts 2.

OGNL Support in Struts 2

OGNL is an expression language used to manipulate and retrieve different properties of Java objects. OGNL has its own syntax, which is very simple in structure; therefore, it is easy to learn and use and also makes the code more readable. OGNL acts as an expression language for the GUI elements to model objects. You can utilize OGNL for multiple uses. The uses of OGNL are as follows:

- Provides a `TypeConverter` mechanism that is used to convert values of a given data type to another data type. For example, it converts `String` type to numeric type.
- Serves as a data source language used to create a map between table columns and a Swing `TableModel`.
- Works as a binding language between Web components and the underlying model objects.
- Serves as a replacement to other property getting languages used by Jakarta Commons BeanUtils package or JSTL's expression languages, as it is more expressive in terms of getting and setting the objects properties.

OGNL syntax provides a high-level abstraction for navigating object graphs by specifying paths through JavaBeans properties, collection indices, and so on. The following code segment shows how to find the difference between the two Java code segments and the corresponding OGNL expression to access a session scoped object (say, student) and obtain value of a property (say name). In Web frameworks, expression languages have similar goals to eliminate the repetitive code. For example, without an Expression Langugae (EL), getting a Student object from the session and then displaying its name on the Web page requires a few lines of Java code in a JSP, as can be seen in the following code snippet:

```
//Using Java Code - First way
<%
Student student = (Student) session.get("student");
String name = student.getName();
%>
<%= name %>
//Using Java Code - Second way
<%= ((Student) session.get("student")).getName() %>
//OGNL expression - Third Way
#session.student.name
```

In the preceding code snippet, we have used three ways to get the name of a student from a session. First two ways use Java code in JSP page. It is to be noted that the second way has reduced the code to a single line, but the code now looks complex and difficult to read. The third way shows the use of OGNL expression to get the name of a student from a session.

Syntax of OGNL

The most commonly used unit in OGNL expression language is the navigation chain, which is also termed as chain. The chain consists of the following parts:

- Property names, such as `name` and `text`
- Method calls, such as the `hashCode()`
- Array indices, such as `listeners [0]`

All OGNL expressions are evaluated in the context of the current object. Chains work by inheriting the output of the previously executed link and present the same as the current object for the next link. A chain can be extended upto any limit. For example, consider the following chain:

```
name.toCharArray() [0].numericValue.toString()
```

The preceding expression is evaluated in the following manner:

- First, it extracts the `name` property of the initial or root object. This root object is provided to the OGNL through the OGNL context.
- Then, it calls the toCharArray() method of the resulting string.
- After calling the toCharArray()method, the first character at 0 index is extracted from the resulting array.
- It then gets the numericValue property from the first character. The character is represented as a Character object, and numeric value can be retrieved by using the getNumericValue() method of the Character class.
- Finally, the String is returned after calling the toString() method on the resulting Integer object.

Using OGNL in Struts 2

With the help of a standard naming context, the Struts 2 Framework evaluates the OGNL expression. The topmost object dealing with OGNL is a Map, usually called as a context map or context. You should note that the properties of the root object in the OGNL expressions can be referenced without a special marker notion.

In addition, the Struts 2 Framework maps the value stack to the OGNL root object and OGNL context to the `ActionContext class`. Apart from mapping the value stack, the Struts 2 Framework also places other objects in the `ActionContext` class.

The `Action` instance is always pushed onto the value stack as the Action is on the stack, and the stack is the OGNL root. To access other objects in the `ActionContext class`, we must use the # notation so that the object is looked for in the `ActionContext class`. The following code shows how to reference an `Action` property:

```
<s:property value="state"/>
```

The other non-root objects in the `ActionContext` class can be referenced using the # notation as follows:

```
<s:property value="#session.mySesPropKey"/> or
<s:property value="#session["mySesPropKey"]"/> or
<s:property value="#request["mySesPropKey"]/>
```

The following are the examples of using the `select` tag:

- **Syntax for list** (`{e1, e2, e3}`)— Creates a list that contains the String `n1, n2,` as well as `n3` and selects `n2` as the default value, as shown in the following code snippet:

```
<s:select label="label1" name="name1"
list="{'n1','n2','n3'}" value="%{'n2'}" />
```

- **Syntax for map** (`#{key1:value1,key2:value2}`)— Creates a map used to configure the String `name` to the String `namevalue` and `desg` to the string `desgvalue`:

```
<s:select label="label2" name="name2" list="#{
'name':'namevalue', 'desg':'desgvalue'}" />
```

The following code snippet shows the implementation of `in` and `not in` to determine if an element exists in a Collection or not:

```
<s:if test="'name' in {'name','desg'}">
  muhahaha
</s:if>
<s:else>
  boo
</s:else>

<s:if test="'name' not in {'name','desg'}">
  muhahaha
```

```
    </s:if>
    <s:else>
      boo
    </s:else>
```

The following wildcards are used within the collection to select a subset of a collection (called projection):

- ? – Refers to the wildcard depicting that all elements should match the selected logic
- ^ – Refers to the wildcard depicting that only the first element should match the selected logic
- $ – Refers to the wildcard depicting that only the last element should match the selected logic

For example, the following snippet shows how to obtain a subset of just female relatives from the object person:

```
person.relatives.{? #this.gender == 'female'}
```

You should note that OGNL also provides support for the basic lambda expression syntax to write simple functions. The following code snippet shows that the nth element in Fibonacci series is returned:

```
fib: if n==0 return 0; elseif n==1 return 1; else return fib(n-2)+fib(n-1);
//Output of different method calls with argument 0, 1 and 11
fib(0)= 0

fib(1)= 1
fib(11)= 89
```

The following code shows the usage of lambda expression for the same fib method in Struts 2 Framework:

```
    <s:property
    value="#fib =:[#this==0 ? 0 : #this==1 ? 1 : #fib(#this-2)+#fib(#this-1)],
#fib(11)" />
```

In the preceding code snippet, the code within the brackets is the lambda expression. The # notation is used to hold the argument to the expression, which initially starts at 11.

Implementing Struts 2 Tags

The 's' prefix has been used in JSP pages to implement Struts 2 tags. The tags in Struts 2 can be divided into two categories:

- Generic tags
- UI tags

Generic Tags

Generic tags are used for controlling the flow of data and data extraction from the value stack or other locations. In other words, the Generic tags are developed to control the flow of execution in the page, and obtain and display data.

Similar to other tags, Generic tags are also contained in a tag library, which is a collection of predefined tags. We have been using Struts 2 tag library in our previous applications, and designed a number of JSP pages using them. Each tag has a specific work and many attributes. You can use these tags in your JSP page by simply importing a tag library in your JSP page, as follows:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
```

The <%@ %> directive identifies the URI defined in the previously listed `<taglib>` element and states that all the tags should be prefixed with string 's'. In Struts 1, there are many tag libraries; but in Struts 2, only one tag library, struts-tags, is specified. The Struts 2 tags are grouped into different categories. The Generic tags are one of the most important tags of the Struts 2 tag library. The Generic tags control the execution flow of the rendered pages. The Generic tags are specified in the `/struts tags` tag library. There are two types of Generic tags:

- Control tags
- Data tags

Control Tags

Control tags are used to control the behavior of data on a page. Control tags, as the name suggests, are designed to control the path of execution with the help of some decision making constructs, such as `if`, `else`, and `elseif` tags. Another controlling structure is looping constructs, which is supported by the `iterator` tag. You can use the `iterator` tag with the `append` and `merge` tags to loop over data and the `if\else\elseif` tag to

make decisions, you can develop more sophisticated and dynamic Web pages. Table 20.15 lists the control tags of the Struts 2 Framework:

Table 20.15: Control Tags of the Struts 2 Framework

Tag	Description
`if`	Allows you to implement the conditional flow of execution
`elseIf`	Allows you to extend the decision making to a number of conditions, and is used along with the if tag
`else`	Allows you to provide alternate execution path if all conditions given in the if tag and the elseif tag(s) fail
`append`	Allows you to append iteratable lists together to form an appended iterator
`generator`	Generates an iterator on the basis of a given value
`iterator`	Allows you to iterate over the given value
`merge`	Allows you to merge iteratable lists together to form a merged iterator
`sort`	Allows you to sort a given List
`subset`	Accepts an iterator and provides output for its subset

Data Tags

Data tags are used for creating and manipulating data. These tags allow you to either get data out of the value stack or place variables and objects in the value stack. Before discussing data tags, let's explain the meaning of the value stack, which is just a stack of objects. Initially, the stack contains input properties of the action executed. This is why when we write value="name", it calls the `getName()` method of the action class. New objects are placed onto the stack when the `<s:iterator>` or `<s:property>` tags are used. The data tags available in Struts 2 are listed in Table 20.16:

Table 20.16: Data Tags Available in Struts 2

Tag	Description
`a`	Helps to create an hyperlink, and is similar to HTML <a href = " " /> tag.
`action`	Helps to call actions directly from a page.
`bean`	Helps to create an instance of a class, which is in agreement with JavaBean specification.
`date`	Helps to format and display date objects in different ways.
`debug`	Helps to create a link, which can be clicked to view all value stack contents with different items available in the stack context. This helps in debugging.
`i18n`	Helps to get a resource bundle and place it on value stack, so that in addition to the bundle associated with action, other resource bundles could be used.
`include`	Includes output of some Servlet or JSP page.
`param`	Helps to define parameters to other tags.
`push`	Pushes the value stack.
`set`	Helps to assign a value to a variable. It is similar to setting a variable with the given value.
`text`	Helps to render an internationalized message from the resource bundle.
`url`	Helps to create a URL.
`property`	Helps to get the property of a value and returns the object from the top of the stack, by default.

Struts 2 comes with another set of tags, which can be used to design a JSP page with quality components for the required user interface, for example a form with some input fields and a submit button.

After discussing Generic tags, let's now start exploring another category of Struts tags, UI Tags.

UI Tags

The UI tags provide a way to create user interface (UI) components on the screen. The UI tags are basically used to design the Web page with high quality UI components that may include a form with other individual components contained in it, such as text box, combo box, submit buttons, radio buttons, and checkboxes. There is a tag associated with each UI component that is used to create and place these components on the page. These UI components can be customized by setting different values for different attributes of the corresponding tags.

The UI tags provide the logic behind the flow of an execution. These tags involve the use of data in an application and focus on how the data is entered by the user and accepted by the application. This data can be retrieved from the actions, stack values, or from the data tags available in the Struts 2 Framework. The UI tags represent the tags and generate the reusable HTML format. The UI tags work on the basis of templates. Templates again group together to form the themes, which are used to perform the actual operation. The UI tags are classified into the following two categories:

- Form tags
- Non-form tags

Form Tags

Form tags allow the display of data in a simple and reusable format. These tags also include the HTML representation, which provides the feature of reusability. These tags are used to create a basic HTML form and other form components. Table 20.17 lists the form tags of the Struts 2 Framework:

Table 20.17: Form Tags of the Struts 2 Framework	
form	checkboxlist
file	Token
password	Label
textarea	Hidden
checkbox	doubleselect
select	combobox
radio	Submit
head	DateTimePicker
optiontransferselect	optgroup
reset	textfield
updownselect	

These tags are described in the coming headings, with all their syntaxes and the attributes that control the behaviors of the UI components rendered using these tags.

Non-Form Tags

We have seen a number of form UI tags used to create various components, which are used in a simple form created in a Web page. There is another category of UI tags, known as the Non-form UI tags, which are used to display the output texts, such as action-level messages, action-level errors, and field-level errors. These tags also create some other UI components, such as tabbed panel and table.

Non-form tags are grouped under certain templates. The templates are again combined to form a theme. The themes available for Non-form tags are the same as that of the Form tags. All Non-form tags must extend one of these themes to draw the output of the page. Table 20.18 lists the Non-form tags:

Table 20.18: Non-form Tags of the Struts 2 Framework

actionerror	Div	tabbedpanel
actionmessage	FieldError	tree
component	Table	treenode

These Non-form tags are used for creating UI components, such as tabs, div, tables, and trees. In addition, these tags include tags to display action errors, action messages, and field errors. The Non-form tags are used to display the output to the user without using the forms. It does not take the input from a form tag to generate the output.

Controlling Results in Struts 2

Action classes are used for the processing of user action, which has been requested with the data sent as input parameters. Interceptors provided for all required pre-processing are executed before the execution of the action class. The processing of request must consequently provide some result, which is to be sent as the response back to the user. After the execution of action class, we always get a string as result code. This result code is mapped to a specific source to be rendered as view to the user, which may be a JSP page, or an HTML page but it is not limited to these two things. There are different classes implementing `com.opensymphony.xwork2.Result` interface, which are responsible for rendering output to the user. Struts 2 provides a set of result classes that needs to process different types of results, such as rendering JSP or HTML page, generating output for the users that are using freemarker or velocity templates or sometimes invoking other actions to get results.

Now, let's explore the result classes along with their need, implementation, and configuration details.

What is Result?

Struts 2, similar to Struts 1, is based on MVC architecture and the results in Struts 2 are purely associated with the view part of the MVC implementation. The term result is used to refer to the output to be displayed to the user as a consequence of the action performed for its request. All user requests are processed by some action class, and each user request needs some response in return. Result in Struts 2 can be defined as something that can be obtained with the help of the result code returned by the action and the result class, which renders the associated source (JSP or HTML) as the next view to the user.

Action classes can return one or many types of result codes (success or error), and the results generated and returned to the user for these different result codes need not be of the same type. The different JSP or HTML pages are returned to generate view in response to results codes returned by the action classes.

The supported result classes are configured in the Struts configuration file so that they can be used by different actions. The association between action and the possible results is also mapped through the result configuration provided with each action mapping in the `struts.xml` file.

Types of Results

The different JSP or HTML pages that are returned to generate view in response to result returned by action classes are not same. For example, the result `success` may render a JSP or HTML page and another result `error` may need to send a HTTP header to the user. Struts 2 comes bundled with several inbuilt result types. The Struts 2 Framework provides several implementation of the `com.opensymphony.xwork2.Result` interface. These implementations can be directly used in your application. Although you can make your own Result class and use it, generally these inbuilt implementations are sufficient for developing the View portion of the Web application. We can define a group of supported `<result-type>` under `<result-types>` elements in `struts.xml`. We can alternately include the `struts-default.xml` file and extend the struts-default package to make all the `<result-type>` definitions available in the `struts.xml file`.

The result types, which are configured in the `struts-default.xml` file, are given in the following code snippet:

```
<package name="struts-default">
  <result-types>
        <result-type name="chain"
        class="com.opensymphony.xwork2.ActionChainResult"/>
        <result-type name="dispatcher"
        class="org.apache.struts2.dispatcher.ServletDispatcherResult"
        default="true"/>
        <result-type name="freemarker"
        class="org.apache.struts2.views.freemarker.FreemarkerResult"/>
        <result-type name="httpheader"
        class="org.apache.struts2.dispatcher.HttpHeaderResult"/>
        <result-type name="redirect"
        class="org.apache.struts2.dispatcher.ServletRedirectResult"/>
        <result-type name="redirect-action"
        class="org.apache.struts2.dispatcher.ServletActionRedirectResult"/>
        <result-type name="stream"
        class="org.apache.struts2.dispatcher.StreamResult"/>
        <result-type name="velocity"
        class="org.apache.struts2.dispatcher.VelocityResult"/>
        <result-type name="xslt"
        class= "org.apache.struts2.views.xslt.XSLTResult"/>
        <result-type name="plaintext"
        class="org.apache.struts2.dispatcher.PlainTextResult" />
  </result-types>
</package>
```

Struts 2 provides various result types that are configured in the `struts-default.xml` file with their associated result classes. The result types are described in Table 20.19:

Table 20.19: Result Types Available in the Struts 2 Framework

Result name	Description
`Chain`	Helps to implement action chaining
`Dispatcher`	Helps to integrate Web resource, such as JSP
`Redirect`	Helps to redirect the browser to a new URL
`Velocity`	Helps to integrate Velocity
`FreeMarker`	Helps to integrate FreeMarker
`HttpHeader`	Helps to control special HTTP behaviors
`Redirect-Action`	Helps to redirect the request to another action mapping
`Stream`	Helps to stream an InputStream back to the browser, usually for tasks such as file downloading
`XSL`	Helps to integrate the XML/XSLT in the output
`PlainText`	Helps to display the raw content of a particular page, such as HTML and JSP

You can also create additional result types and plug them in your Web application, by implementing the `com.opensymphony.xwork2.Result` interface. You can make result types for different operations, such as generating e-mails, generating images, and so on. Out of these result types, certain result types, such as chain, dispatcher, and redirect, are used frequently as compared to other Result types.

Configuring Results

The method of the action class that processes the user request returns a string as a result code. The String value could be `success`, `error`, `input`, and so on. The string result values returned by an action class are matched with the result elements configured in the struts.xml file for that action class. The action mapping defines the set of possible results, which describes the different possible outcomes. The Action interface has a defined standard

set of result tokens. The result tokens describe the names of the predefined result. These result names are mentioned as follows:

- String SUCCESS = "success";
- String NONE = "none";
- String ERROR = "error";
- String INPUT = "input";
- String LOGIN = "login";

The preceding mentioned names of results are predefined, but you can also add your own result name to match the specific cases in your Web application. For example, when the `execute()` method of your Action returns `success` string value, this string value is matched with the result element having `name=success` in `struts.xml`.

We need to configure all supported result types to be used in an application. In addition, the set of possible results to be rendered are also defined in the `struts.xml` file for different action mappings. Consequently, the basic elements used while configuring results, are `<result-types>`, `<result-type>`, and `<result>`.

Configuring Result Types

We have two approaches to configure result types for an action class. The first one is to define all the result types to be used in the application providing a set of `<result-type>` elements for each. This is always required if you want to configure your customized result types. The following code snippet shows the configuration of result types for an action class in the struts.xml file:

```
<struts>
  <package name="mypackage">
        <result-types>
        <result-type name="dispatcher"
        class=" org.apache.struts2.dispatcher.ServletDispatcherResult"
        default="true"/>
        <result-type name="redirect"
        class=" org.apache.struts2.dispatcher.ServletRedirectResult"/>
        </result-types>
        <interceptors>
        //Interceptors declaration.
        </interceptors>
        <action name="login" class="com.kogent.LoginAction">
        <result name="input">login.jsp</result>
        <result name="success" type="redirect">/secure/admin.jsp</result>
        </action>
  </package>
</struts>
```

In the preceding code snippet, results returned by the LoginAction class are configured along with some interceptors. In the preceding code snippet, the login.jsp page is set as a view if the LoginAction class returns `input` as a result type. Similarly, admin.jsp is set as a view in case LoginAction class returns `success`.

The second approach to configure the result types is to include `struts-default.xml` file in your copy of the `struts.xml` file. The `struts-default.xml` file defines a package named `mypackage` with all the available result types configured. The package defined in `struts.xml` can extend the `struts-default` package to make all result types available for all action mappings provided in our package. The following code snippet shows the second approach to configure result types for an action class:

```
<struts>
  <include file="struts-default.xml"/>
  <package name="mypackage" extends="struts-default">
        <action name="login" class="com.kogent.LoginAction">
              <result name="input">login.jsp</result>
              <result name="success" type=
                    "redirect">/secure/admin.jsp</result>
        </action>
  </package>
</struts>
```

In the preceding code snippet, a standard way of implementing all results types for action mapping is shown. We can use all kinds of result types directly without providing any `<result-type>` element for them.

After having this general discussion over results in Struts 2, let's learn how to implement validations in Struts 2.

Performing Validation in Struts 2

In this section, we discuss about validation framework of Struts 2 applications, which is based on XWork validation framework. It is a powerful addition to the Struts 2 Framework. This framework allows you to manage the validations in a separate configuration file, where they can be reviewed and modified without changing Java or JSP code, because localization and validations are tied to the business tier, not to the presentation tier. The XWork validation framework also provides several basic validators that perform validations. The XWork validation framework allows developers to add custom validators to support user-defined validation. You can also use the original Struts 2 `validate()` method in tandem with the Struts validators, if required. Struts validator also provides the localization feature that is similar to the validate() method. The validate() method allows you to share the standard message resource file with the main framework, providing an error free solution for the translators you are working with. Struts 2 validators provide validating input for meeting the requirements of complex applications.

Let's now discuss about the XWork validation framework, which provides validation in the Struts 2 based Web applications.

XWork Validation framework

When a client submits data through a form, it is necessary to check whether the data is valid and in proper format, as required. Validating form data is essential to prevent incorrect data from getting into your applications. Validations can also be performed on the database used in your Web application. Suppose, you perform validation on databases and the welcome page of your application is displayed before the client. This welcome page contains a simple login form, which requires an email address. The proper format of the required email address is defined on the database, which is not known to the client. Now, if the client enters the email address in the wrong format, an error message is displayed. Providing meaningful error messages is one of the keys to providing feedback to the user.

Validations can be performed in your action class, but this approach can make your code very complicated. The code for an action class becomes complicated as while implementing various validation mechanisms, the code for an action class becomes large and difficult to maintain. For example, the code of an action class becomes complicated if we also provide the validation logic for checking empty fields in an action class. Therefore, implementing validation checks declaratively is always preferred over hard coding of logic in an action class. These concerns led to the development of the `XWork Validation` framework, which is part of XWork and describes the validations to be performed on an action with the help of metadata provided in XML files.

Bundled Validators

A class that implements `com.opensymphony.xwork2.validator.Validator` interface can be defined as a validator class or simply a validator. Validators are called by the framework to validate an object by accessing its properties. We have a set of validators, which are already defined by the XWork Validation framework, and the associated classes are bundled in Struts 2 API. Therefore, these validators are also known as bundled validators. These bundled validators are defined in an XML file called `validators.xml`. The `validators.xml` file must be available in your classpath (`/WEB-INF/classes`). However, in case there is no custom validator used in the application, there is no need to put this file in the classpath. All the bundled validators are automatically registered with `ValidatorFactory`, when the `com/opensymphony/xwork2/validator/validators/default.xml` file containing the configuration for all bundled validators is loaded.

All the bundled validators should be configured in the validators.xml file if a custom validator is defined and placed in a classpath. When a `validators.xml` is detected in the classpath, the `com/opensymphony/xwork2/validator/validators/default.xml` file is not automatically loaded; it is only loaded when a custom `validators.xml` cannot be found in the classpath. The following code snippet shows the code of the validators.xml file that contains all the bundled validators:

```
<validators>
<validator name="required"
class="com.opensymphony.xwork2.validator.validators.RequiredFieldValidator"/>
<validator name="requiredstring"
class="com.opensymphony.xwork2.validator.validators.RequiredStringValidator"/>
<validator name="int"
class="com.opensymphony.xwork2.validator.validators.IntRangeFieldValidator"/>
<validator name="double"
class="com.opensymphony.xwork2.validator.validators.DoubleRangeFieldValidator"/>
<validator name="date"
class="com.opensymphony.xwork2.validator.validators.DateRangeFieldValidator"/>
<validator name="expression"
class="com.opensymphony.xwork2.validator.validators.ExpressionValidator"/>
<validator name="fieldexpression"
class="com.opensymphony.xwork2.validator.validators.FieldExpressionValidator"/>
<validator name="email"
class="com.opensymphony.xwork2.validator.validators.EmailValidator"/>
<validator name="url"
class="com.opensymphony.xwork2.validator.validators.URLValidator"/>
<validator name="visitor"
class="com.opensymphony.xwork2.validator.validators.VisitorFieldValidator"/>
<validator name="conversion"
class="com.opensymphony.xwork2.validator.validators
  .ConversionErrorFieldValidator"/>
<validator name="stringlength"
class="com.opensymphony.xwork2.validator.validators
  .StringLengthFieldValidator"/>
<validator name="regex"
class="com.opensymphony.xwork2.validator.validators.RegexFieldValidator"/>
</validators>
```

The preceding code snippet shows the bundled validators with their names and descriptions. All validators have their properties, such as `defaultMessage`, `messageKey`, and `shortCircuit`, configured using built-in attributes in the XML file. In addition, all `validators`, except the Expression validator, support a `fieldName` property that is normally set by the Field `validators` inside a `<field>` element. Table 20.20 describes bundled validators available in the Struts 2 Framework:

Table 20.20: Shows Bundled Validators

Validator	Description
`RequiredFieldValidator`	Checks whether your input field is not null.
`RequiredStringValidator`	Checks whether a String field does not contain null value or has a length greater than 0.
`IntRangeFieldValidator`	Checks whether an Integer value entered in the field is within the specified range.
`DoubleRangeFieldValidator`	Checks whether the double or double value is in the specified range.
`DateRangeFieldValidator`	Checks whether the date is within the specified range.
`ExpressionValidator`	Refers to a non-field validator that evaluates the Boolean value in the OGNL expression.
`FieldExpressionValidator`	Checks whether your input field contains OGNL expression and returns a Boolean value.
`EmailValidator`	Checks whether a given String field is in a valid Email address format.
`URLValidator`	Checks whether your input text field contains valid URL or not.
`VisitorFieldValidator`	Allows the validation to be run against the value of the field to which this validator is applied.

Table 20.20: Shows Bundled Validators

Validator	Description
ConversionErrorFieldValidator	Checks if any conversion error had occurred for this field. The ConversionErrorFieldValidator validator checks whether a type conversion error had occurred while setting the value on this field. It also uses the type-conversion framework to create the correct field error message to be added for this field.
StringLengthFieldValidator	Validates a string for the number of character. This validator checks and makes sure that the length of a String property's value is within a specified range.
RegexFieldValidator	Validates a String field using a regular expression. This validator checks the value of a field against the given regular expression.

Registering Validators

Validators must be registered with the `com.opensymphony.xwork2.validator.ValidatorFactory` class by using the `registerValidator` static method of the `ValidatorFactory` class. The signature for `registerValidator()` method is `public static void registerValidator (String valname, String classname)`. This method allows you to register the specified validator to the existing map of validators. In this method, the parameter `valname` is the name of the validator to be added and `classname` is the fully qualified classname of the validator.

You can register a validator simply by adding the `validators.xml` file in the root of the classpath (`/WEB-INF/classes`), which declares all the validators that you need to use. The following code snippet shows the code for the `validators.xml` file where all the bundled validators are registered:

```
< validators >
<validator name ="required"
class="com.opensymphony.xwork2.validator.validators.RequiredFieldValidator" />
<validator name ="requiredstring"
class="com.opensymphony.xwork2.validator.validators.RequiredStringValidator"/>
<validator name ="int"
class="com.opensymphony.xwork2.validator.validators.IntRangeFieldValidator" />
<validator name ="double"
class="com.opensymphony.xwork2.validator.validators.DoubleRangeFieldValidator"/>
<validator name ="date"
class="com.opensymphony.xwork2.validator.validators.DateRangeFieldValidator" />
<validator name ="expression"
class="com.opensymphony.xwork2.validator.validators.ExpressionValidator" />
<validator name ="fieldexpression"
class="com.opensymphony.xwork2.validator.validators.FieldExpressionValidator" />
<validator name ="email"
class="com.opensymphony.xwork2.validator.validators.EmailValidator" />
<validator name ="url"
class="com.opensymphony.xwork2.validator.validators.URLValidator" />
<validator name ="visitor"
class="com.opensymphony.xwork2.validator.validators.VisitorFieldValidator" />
<validator name ="conversion"
class="com.opensymphony.xwork2.validator.validators.
  ConversionErrorFieldValidator" />
<validator name ="stringlength"
class="com.opensymphony.xwork2.validator.validators
  .StringLengthFieldValidator"/>
<validator name ="regex"
class="com.opensymphony.xwork2.validator.validators.RegexFieldValidator" />
< / validators >
```

In the preceding code snippet, all the pre-defined validators or bundled validators are registered in the validation.xml file.

Defining Validation Rules

Struts 2 framework provides validation rules for action classes. There are two different ways to define validation rules to validate business logic defined in the action classes.

Validation rules for Struts 2 framework can be defined in the following ways:

- Per Action class
- Per Action alias

Per Action Class

We can define validation rules for a single action class by creating an XML file, named `ActionName-validation.xml`. This is known as Action-level validation. To create an Action-level validation, create a file, `ActionClass-validation.xml`, at the same location where an action class is located. For example, if the name of the action class is `MyAction`, the name of the validation XML file would be `MyAction-validation.xml`. The following code snippet shows the Action level validation:

```
< action name="myAlias" class=" action.level.validation.MyAction" >
< / action>
  <action name="myAnotherAlias" class=" action.level.validation.MyAction"
  method = "another" >
< / action>
```

In the preceding code snippet, both the actions, `myAlias` and `myAnotherAlias`, are validated according to the same validation configuration file named `MyAction-validation.xml` file.

Per Action Alias

We can also implement validation rules per action alias by creating a validation configuration file, named as `ActionClassName-alias-validation.xml`. This is known as Action Alias-level validation. To create an Action Alias-level validation, create a file `ActionClassName-alias-validation.xml` at the same location where the action class is located. For example, if the name of the action class is `MyAction` with an alias `myAlias`, the name of the validation XML file would be `MyAction-myAlias-validation.xml`. The following code snippet shows the code for the `MyAction-myAlias-validation.xml file:`

```
< action name="myAlias" class="action.level.validation.MyAction" >
  < / action>
<action name="myAnotherAlias" class="action.level.validation.MyAction"
method = "another">
< / action>
```

In the preceding code snippet, `Action` Alias-level validation allows the validation to be applied to all action classes with alias named `myAlias`. This validation mechanism is configured in the `MyAction-myAlias-validation.xml`. The `MyAction-myAlias-validation.xml` file will not validate `myAnotherAlias` action alias as it is not been defined in the validation configuration file representing its alias.

Custom Validators

In addition to using bundled Struts 2 validators, you can build your own custom validators for providing user-defined validation logic in your Web applications. The validation framework allows custom validators to be built and applied in the same declarative fashion as other bundled validators. Implementing a custom validator is as simple as creating a class that extends the `com.opensymphony.xwork2.validator.validators.ValidatorSupport` (for Global validator) or `com.opensymphony.xwork2.validator.validators.FieldValidatorSupport` (for `FieldValidator`). The following code snippet shows the code to build a custom validator named `MyStringLengthFieldValidator`:

```
package com.kogent.validators;
import com.opensymphony.xwork2.validator.ValidationException;
import com.opensymphony.xwork2.validator.validators.FieldValidatorSupport;
public class MyStringLengthFieldValidator extends FieldValidatorSupport{
  private int maxLength = -1;
  private int minLength = -1;
  private boolean dotrim = true;
```

```
    public void setMinLength ( int minLength )
    {
        this.minLength = minLength;
    }
    public void setMaxLength ( int maxLength )
    {
        this.maxLength = maxLength;
    }
    public void setTrim ( boolean trim )
    {
        dotrim = trim;
    }
    public int getMinLength ( )
    {
        return minLength;
    }
    public int getMaxLength ( )
    {
        return maxLength;
    }
    public boolean getTrim ( )
    {
        return dotrim;
    }
    public void validate ( Object obj ) throws ValidationException
    {
        String fieldName = getFieldName( ) ;
        String val = ( String ) getFieldValue( fieldName , obj )        ;
        if ( dotrim )
        {
            val = val.trim ( );
        }
        if (( minLength > -1 ) && ( val.length ( ) <  minLength ) )
        {
            addFieldError ( fieldName , obj );
        }
        else if (( maxLength > -1 ) && ( val.length ( ) >  maxLength ) )
        {
            addFieldError ( fieldName , obj );
        }
    }
}
```

In the preceding code snippet, a custom validator named `MyStringLengthFieldValidator class` is created by extending the `FieldValidatorSupport` class. The `MyStringLengthFieldValidator` class adds the properties `maxLength`, `minLength`, and `trim`, and uses them to check against the length of the String. The `getFieldName()`, `getFieldValue()`, and `addFieldError()` methods are implemented in the abstract base class. The `MyStringLengthFieldValidator class` needs to be registered with the validation framework by adding the code in the validation.xml file, as shown in the following code snippet:

```
<validator name="mystringlength"
class="com.kogent.validators.MyStringLengthFieldValidator"/ >
```

In the preceding code snippet, custom validator named `MyStringLengthFieldValidator` class is registered in the validation.xml file to validate any number of action classes available in a Web application.

When implementing custom validators and registering them using `validators.xml` file, all the required bundled validators should also be configured in the `validators.xml` file.

Short-circuiting Validators

Short-circuiting validators are those validators that cause the other validators to quit validation, if the first validator itself fails to validate. For example, if an email field is left blank, you do not need to check whether it is a valid email address. In order to implement this function, XWork 1.0.1 added a short-circuit property to the Validation framework. By using this validation, it is possible to short-circuit a stack of validators. The following code snippet shows the configuration for the short-circuit validation:

```
< validators >
< ! - - Field Validator Syntax for EmailValidator  - - >
  < field   name = "EnterEmail" >
        < field-validator type = "requiredstring" short-circuit = "true" >
              < message > You must enter a value for  Email Address
              < /message >
        < /field-validator >
        < field-validator type = "email" short-circuit = "true" >
              < message > Enter a valid Email Address < /message >
        < /field-validator >
  < /field >
< / validators >
```

The preceding code snippet shows that if the `Email` field is null or empty, the EmailValidator is not called because an attribute named short-circuit is set to `true` for the required string validator.

Validation Annotation

Validation annotations help in implementing validation rules on different fields without configuring them in some XML files. Different validation annotations are provided for corresponding validation rules. Let's now discuss each of these validation annotations.

***ConversionErrorFieldValidator** Annotation*

The `ConversionErrorFieldValidator` annotation checks whether there are any conversion errors for a field and corrects them. This validator must be applied at the method level. It has five parameters, namely message, key, fieldName, shortCircuit, and type. Among these five parameters, only the message and type are the required parameters; others are the optional. The following code snippet shows the implementation of the `ConversionErrorFieldValidator` annotation:

```
@ConversionErrorFieldValidator(message = "Default message",
key = "i18n.key", shortCircuit = true)
```

The DateRangeFieldValidator Annotation

The DateRangeFieldValidator annotation checks the information regarding the date field; whether the date field has a value within a specified range. The parameters of `DateRangeFieldValidator` annotation are listed in Table 20.21.

Table 20.21: Parameters of DateRangeFieldValidator Annotation

Parameters	Description
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
fieldName	Specifies the name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
type	Specifies Enum value from the ValidatorType element
min	Specifies a minimum date set for the field to be validated
max	Specifies a maximum date set for the field to be validated

Of these parameters, the `message` and `type` parameters are the necessary fields along with the `min` and `max` fields for the checking operations. All the other parameters are optional.

The implementation of the `DateRangeFieldValidator` annotation is shown in the following code snippet:

```
@DateRangeFieldValidator(message = "Default message",
  key = "i18n.key",
  shortCircuit = true,
  min = "2010/01/01", max = "2010/12/31")
```

The DoubleRangeFieldValidator Annotation

The DoubleRangeFieldValidator annotation checks whether or not a double field has a value within the given range. Therefore, for the DoubleRangeFieldValidator annotation also, you have to provide min and max properties; otherwise, no validation is performed. The parameters of DoubleRangeFieldValidator annotation are listed in Table 20.22.

Table 20.22: Parameters of DoubleRangeFieldValidator Annotation

Parameters	Description
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
fieldname	Specifies the name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
type	Specifies Enum value from the ValidatorType element
minInclusive	Specifies a minimum inclusive number for the field to be validated
maxInclusive	Specifies a maximum inclusive number for the field to be validated
minExclusive	Specifies a minimum exclusive number for the field to be validated
maxExclusive	Specifies a maximum exclusive number for the field to be validated

The implementation of the DoubleRangeFieldValidator annotation is shown in the following code:

```
@DoubleRangeFieldValidator(message = "Default message",
  key = "i18n.key",
  shortCircuit = true,
  minInclusive = "1.567", maxInclusive = "99.678")
```

The EmailValidator Annotation

The EmailValidator annotation checks whether the field contains a valid email address. It has the parameters similar to that of the ConversionErrorFieldValidator annotation.

The implementation of the EmailValidator annotation is shown in the following code snippet:

```
@EmailValidator(massage = "Default message",
  key = "i18n.key", shortCircuit = true)
```

The ExpressionValidator Annotation

The ExpressionValidator annotation is used to validate an expression. This annotation must be applied at the method level. The parameters of ExpressionValidator annotation are listed in Table 20.23.

Table 20.23: Parameters of ExpressionValidator Annotation

Parameters	Description
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
shortCircuit	Specifies if the current validator class should be used as shortCircuit
expression	Specifies an OGNL expression returning a boolean value

The implementation of the ExpressionValidator annotation is shown in the following code snippet:

```
@ExpressionValidator(message = "Default message",
  key = "i18n.key",
  shortCircuit = true, expression = "Sample OGNL Expression")
```

The FieldExpressionValidator Annotation

The `FieldExpressionValidator` annotation performs validation with the help of an OGNL expression. If the expression returns false at the time it is evaluated against the value stack, the error message gets added to the field. The parameters of `FieldExpressionValidator` annotation are listed in Table 20.24:

Table 20.24: Parameters of FieldExpressionValidator Annotation

Parameters	Description
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
fieldName	Specifies the name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
expression	Specifies an OGNL expression returning a boolean value

Of these parameters, only the message and type are the required fields. The others are optional. The implementation of the `FieldExpressionValidator` annotation is shown in the following code snippet:

```
@FieldExpressionValidator(message = "Default message",
   key = "i18n.key",
   shortCircuit = true,
   expression = "Sample OGNL expression" )
```

The IntRangeFieldValidator Annotation

The `IntRangeFieldValidator` annotation validates whether or not the numeric field has a value within a specified range. The parameters of `IntRangeFieldValidator` annotation are listed in Table 20.25:

Table 20.25: Parameters of IntRangeFieldValidator annotation

Parameters	Description
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
fieldName	Specifies the name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
type	Specifies Enum value from the ValidatorType element
min	Specifies a minimum date set for the field to be validated
max	Specifies a maximum date set for the field to be validated

When you are using this annotation, you have to provide `min` and `max` values in such a way that `0` can also be considered as a possible value.

The implementation of the `IntRangeFieldValidator` annotation is shown in the following code snippet:

```
@IntRangeFieldValidator(message = "Default message",
   key = "i18n.key",
   shortCircuit = true, min = "10", max = "50" )
```

The RegexFieldValidator Annotation

The `RegexFieldValidator` annotation uses a regular expression to validate a String field. It is also applied at the method level. The parameters of the `RegexFieldValidator` annotation are listed in Table 20.26.

Table 20.26: Parameters of the RegexFieldValidator Annotation

Parameters	Description
message	Specifies an error message for a field

Table 20.26: Parameters of the RegexFieldValidator Annotation	
Parameters	**Description**
key	Specifies i18n key from language specific properties file
fieldName	Specifies the name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
type	Specifies Enum value from the ValidatorType element
expression	Specifies an OGNL expression returning a boolean value

The implementation of the `RegexFieldValidator` annotation is shown in the following code snippet:

```
@RegexFieldValidator (key = "regex. Field", expression = "your regex")
```

The RequiredFieldValidator Annotation

The `RequiredFieldValidator` annotation checks whether or not a field is required, and is applied at the method level. The parameters of `RequiredFieldValidator` annotation are listed in Table 20.27:

Table 20.27: Parameters of the RequiredFieldValidator Annotation	
Parameters	**Description**
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
fieldName	Specifies the name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
type	Specifies Enum value from the ValidatorType element

The implementation of `@RequiredFieldValidator` annotation is shown in the following code snippet:

```
@RequiredFieldValidator (message = "Default message",
  key = "i18n.key", shortCircuit = true)
```

The RequiredStringValidator Annotation

The `RequiredStringValidator` annotation checks whether a string field is empty or not. In other words, it verifies whether the length of a string field is greater than zero (0). The parameters of the `RequiredStringValidator` annotation are listed in Table 20.28:

Table 20.28: Parameters of the RequiredStringValidator Annotation	
Parameters	**Description**
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
fieldName	Specifies the name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
type	Specifies Enum value from the ValidatorType element
trim	Specifies a boolean property that determines whether the specified String is trimmed before checking its length

The use of `@RequiredStringValidator` annotation is shown in the following code snippet:

```
@RequiredStringValidator (message = "Default message",
  key = "i18n.key",
  shortCircuit = true, trim = true)
```

The StringLengthFieldValidator Annotation

The `StringLengthFieldValidator` annotation checks the right length of the string field. To utilize this annotation, you need to set `minLength` and `maxLength`. The parameters of the `StringLengthFieldValidator` annotation are listed in Table 20.29:

Table 20.29: Parameters of StringLengthFieldValidator Annotation

Parameters	Description
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
fieldName	Specifies the name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
type	Specifies Enum value from the ValidatorType element
trim	Specifies a boolean property that determines whether the provided String is trimmed before checking its length
minLength	Specifies the minimum length the String must be
maxLength	Specifies the maximum length the String can be

The implementation of `@StringLengthFieldValidator` annotation is shown in the following code snippet:

```
@StringLengthFieldValidator (message = "Default message",
   key = "i18n.key",
   ShortCircuit = true,
   trim = true,
   minLength = "10", maxLength = "50")
```

The StringRegexValidator Annotation

The `StringRegexValidator` annotation validates the entered string against some configured regular expression. The parameters of the `StringRegexValidator` annotation are listed in Table 20.30:

Table 20.30: Parameters of StringRegexValidator Annotation

Parameters	Description
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
fieldName	Specifies the name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
Type	Specifies Enum value from the ValidatorType element
caseSensitive	Determines whether the matching alpha characters in the expression should be checked keeping case-sensitivity in view or not
regex	Specifies the Regular Expression for which to check a match

The implementation of `@StringRegexValidator` annotation is shown in the following code snippet:

```
@StringRegexValidator(message = "Default message",
   key = "i18n.key",
   shortCircuit = true,
   regex = "a regular expression",
   caseSensitive = true)
```

The **UrlValidator** Annotation

The `UrlValidator` annotation checks for a valid URL. It must be applied at the method level. The parameters of `UrlValidator` annotation are listed in Table 20.31:

Table 20.31: Parameters of the UrlValidator Annotation	
Parameters	**Description**
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
fieldName	Specifies the name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
type	Specifies Enum value from the ValidatorType element

The implementation of the `UrlValidator` annotation is shown in the following code snippet:

```
@UrlValidator (message = "Default message",
   key = "i18n.key", shortCircuit = true)
```

The Validation Annotation

Whenever you want to use annotation-based validation, you have to annotate the class or interface with the Validation annotation. The parameters of Validation annotation are listed in Table 20.32:

Table 20.32: Parameters of the Validation Annotation	
Parameters	**Description**
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
fieldName	Specifies name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
type	Specifies Enum value from the ValidatorType element

The following code snippet shows the implementation of the `Validation` annotation:

```
@Validation()
public interface AnnotationDataAware {
   void setDesignationObj(Designation b);
   Designation getDesignationObj();
   @RequiredFieldValidator(message = "You must enter your designation.")
   @RequiredStringValidator(message = "You must enter your designation.")
   void setData(String data);
   String getData();
}
```

In the preceding code snippet, you have to mark the interface with `@Validation` annotation as well as apply standard or custom annotations at the method level. The following code snippet shows an action class using the `@Validation` annotation:

```
@Validation()
public class MyAnnotationAction extends ActionSupport {
   @RequiredFieldValidator(type = ValidatorType.FIELD, message =
   "You must enter a number.")
   @IntRangeFieldValidator(type = ValidatorType.FIELD, min = "5", max =
   "20", message = "number must be between ${min} and ${max},
   current value is ${number}.")
   public void setNumber(int number) {
         this.number = number;
   }
   public int getNumber() {
         return number;
   }
   public String execute() throws Exception {
         return SUCCESS;
   }
}
```

The **Validations** Annotation

You can use various annotations of the same type by nesting the annotations within the `@Validations` annotation at the method level. The parameters of the Validations annotation are listed in Table 20.33:

Table 20.33: Parameters of Validations Annotation

Parameters	Description
requiredFields	Adds the list of RequiredFieldValidators
customValidator	Adds the list of CustomValidators
conversionErrorFields	Adds the list of ConversionErrorFieldValidators
dateRangeFields	Adds the list of DateRangeFieldValidators
emails	Adds the list of `EmailValidators`
fieldExpressions	Adds the list of FieldExpressionValidators
intRangeFields	Adds the list of IntRangeFieldValidators
requiredStrings	Adds the list of RequiredStringValidators
stringLengthFields	Adds the list of StringLengthFieldValidators
urls	Adds the list of `URLValidators`
visitorFields	Adds the list of VisitorFieldValidators
stringRegex	Adds the list of StringRegexValidators
regexFields	Adds the list of RegexFieldValidators
expressions	Adds the list of ExpressionValidators

The following code snippet shows an action class using the `@Validations` annotation:

```
@Validations(
   requiredFields={
         @RequiredFieldValidator(type = ValidatorType.SIMPLE,
         fieldName = "namefield",
         message = "You must enter your name in this field.")
   },
   requiredStrings ={
         @RequiredStringValidator(type = ValidatorType.SIMPLE,
         fieldName = "stringfield",
         message = "You must enter String value for this field.")
   },
   emails = {
         @EmailValidator(type = ValidatorType.SIMPLE,
         fieldName = "emailaddress",
         message = "You must enter your email address for this field.")
   },
)
public String execute() throws Exception {
   return SUCCESS;
}
```

The VisitorFieldValidator Annotation

The `VisitorFieldValidator` annotation allows you to forward the validator to the properties of your action class to use the validation files of the same action class. This annotation lets you use the Model-Driven development pattern and handles the validation logic in an action class. The parameters of VisitorFieldValidator annotation are listed in Table 20.34:

Table 20.34: Parameters of the VisitorFieldValidator Annotation	
Parameters	**Description**
message	Specifies an error message for a field.
key	Specifies i18n key from language specific properties file.
fieldName	Specifies the name of a field to be validated.
shortCircuit	Specifies if the current validator class should be used as shortCircuit.
type	Specifies Enum value from the ValidatorType element.
appendPrefix	Determines whether the name of this field validator should be appended to the field name of the visited field. The name must be appended as it helps in determining the full field name when an error occurs.
context	Determines the context used to validate the Object property.

The implementation of the VisitorFieldValidator annotation is shown in the following code snippet:

```
@VisitorFieldValidator(message = "Default message",
  key = "i18n.key",
  shortCircuit = true,
  context = "sample action alias", appendPrefix = true)
```

The CustomValidator Annotation

The `CustomValidator` annotation can be used for custom validators. You need to use the `ValidationParameter` annotation to provide additional parameters. The parameters of `CustomValidator` annotation are listed in Table 20.35:

Table 20.35: Parameters of the CustomValidator Annotation	
Parameters	**Description**
message	Specifies an error message for a field
key	Specifies i18n key from language specific properties file
fieldName	Specifies the name of a field to be validated
shortCircuit	Specifies if the current validator class should be used as shortCircuit
type	Specifies Enum value from the ValidatorType element

The implementation of `@CustomValidator` annotation is shown in the following code:

```
@CustomValidator(type = "CustomValidatorName", fieldname = "customField" )
```

We have learned about various annotations for validation in Struts 2 Application. Let's now discuss about validating a Struts2 application in the following subsection.

Validating Stuts2App Application

In order to induce validation into our application, we have to get back to our previous application called Struts2App. This application was a simple Struts2 application without any validation mechanism. To validate this application, we first have to edit our action class, UserAction.java, to incorporate validation into this action class. The code for validating application named Struts2App is given in Listing 20.16 (you can find the UserAction.java file on the CD in the code\JavaEE\Chapter 20\Struts2App\src\com\kogent\action folder):

Listing 20.16: Displaying the Code of the UserAction.java File

```
package com.kogent.action;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map;
import org.apache.struts2.interceptor.ApplicationAware;
import com.kogent.User;
```

```
import com.opensymphony.xwork2.ActionSupport;
public class UserAction extends ActionSupport implements ApplicationAware{
   String username;
   String password;
   String city;
   String email;
   String type;
   Map application;
   public void setApplication(Map application) {
         this.application=application;
   }
   public String getUsername() {
         return username;
   }
   public void setUsername(String username) {
         this.username = username;
   }
   public String getPassword() {
         return password;
   }
   public void setPassword(String password) {
         this.password = password;
   }
   public String getCity() {
         return city;
   }
   public void setCity(String city) {
         this.city = city;
   }
   public String getEmail() {
         return email;
   }
   public void setEmail(String email) {
         this.email = email;
   }
   public String getType() {
         return type;
   }
   public void setType(String type) {
         this.type = type;
   }
   public String execute() throws Exception {
         ArrayList users=(ArrayList)application.get("users");

         if(users==null){
                users=new ArrayList();
         }
         if(getUser(username)==null){
                users.add(buildUser());
                application.put("users", users);
         }else{
                this.addActionError("User Name is in use.");
                return ERROR;
         }
         return SUCCESS;
   }
   public String edit() throws Exception{
         ArrayList users=(ArrayList)application.get("users");
         User user=null;
         int index=0;
         Iterator it=users.iterator();
         while(it.hasNext()){
                user=(User)it.next();
                if(user.getUsername().equals(username)){
                       break;
```

```
                }
                index++;
        }
        User newuser=buildUser();
        users.set(index, newuser);
        application.put("users", users);
        return SUCCESS;
    }
    public String deleteUser() throws Exception{
        ArrayList users=(ArrayList)application.get("users");
        User user=null;
        int index=0;
        Iterator it=users.iterator();
        while(it.hasNext()){
                user=(User)it.next();
                if(user.getUsername().equals(username)){
                        break;
                }
                index++;
        }
        users.remove(index);
        application.put("users", users);
        return SUCCESS;
    }
    public User buildUser(){
        User user=new User();
        user.setUsername(username);
        user.setPassword(password);
        user.setCity(city);
        user.setEmail(email);
        user.setType(type);
        return user;
    }
    public User getUser(String username){
        User user=new User();
        boolean found=false;
        ArrayList users=(ArrayList)application.get("users");
        if(users!=null){
                Iterator it=users.iterator();
                while(it.hasNext()){
                        user=(User)it.next();
                        if(username.equals(user.getUsername())){
                                found=true;
                                break;
                        }
                }
                if(found){
                        return user;
                }
        }
        return null;
    }
    public void validate() {
        if ( (username == null ) || (username.length() == 0) ) {
        this.addFieldError("username", getText("app.username.blank"));
        }
        if ( (password == null ) || (password.length() == 0) ) {
        this.addFieldError("password", getText("app.password.blank"));
        }
        if ( (email == null ) || (email.length() == 0) ) {
                this.addFieldError("email", getText("app.email.blank"));
        }
    }
}
```

Recompile the UserAction.java and save it at the location code\JavaEE\Chapter20\Struts2App\WEB-INF\classes\com\kogent\action. Now, add some property files named Struts.properties and ApplicationResources.properties to the application.

The ApplicationResources file is mapped to struts.custom.i18n.resources in the `struts.properties` file, as shown in the following code snippet:

```
struts.custom.i18n.resources=ApplicationResources
```

Save the Struts.properties file at the location `Chapter 20\Struts2App\WEB-INF\src` folder. Similarly, save the ApplicationResources.properties file at the location `code\JavaEE\Chapter 20\Struts2App\WEB-INF\src` folder. The code for the ApplicationResource.properties file is shown in the following code snippet:

```
# Resources for parameter 'ApplicationResources'
# Project Struts2Action
app.username.blank=User Name is Required.
app.password.blank=Password is Required.
app.email.blank=Email is Required.
```

After making the above mentioned changes, deploy Struts2App.war on Glassfish application server. Browse `http://localhost:8080/Struts2App` URL to run the Web application, as shown in Figure 20.10:

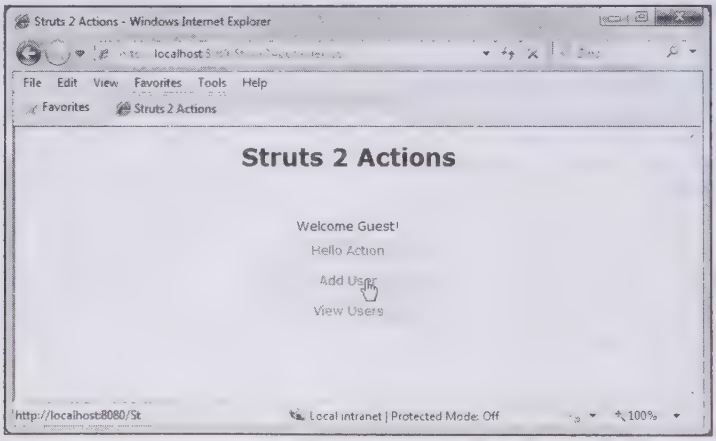


Figure 20.10: Showing the Output of the index.jsp Page

Click the `Add User` hyperlink to add a new user. The output of `add_user.jsp` page is shown in Figure 20.11:

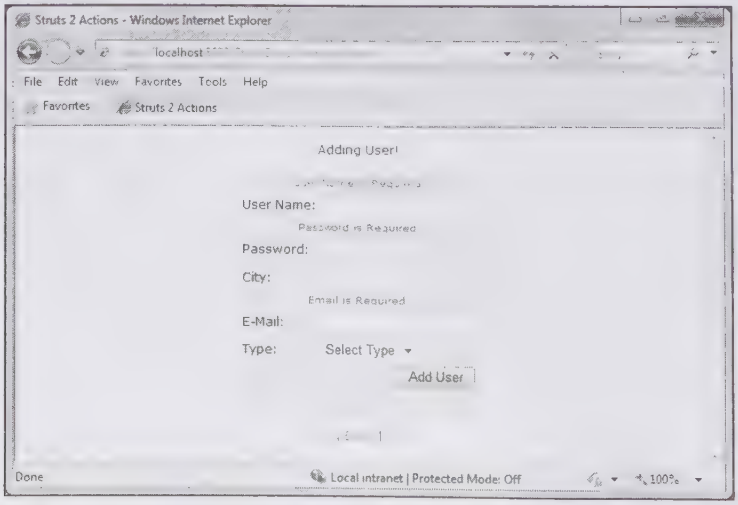


Figure 20.11: Showing the Support for Validation Errors in the add_user.jsp page

Figure 20.11 shows the `add_user.jsp` page, which provides validation by checking whether the fields are empty or not. If the fields are empty, a message is displayed to prompt the user to provide information in them.

Internationalizing Struts 2 Applications

The Struts2 framework has built-in support for internationalization, which allows Web pages to be displayed according to a client's requirements. In other words, it reduces the effort of the developer to customize the Web pages in different languages. Internationalization provides the translation of messages in a user's native language or locale. It also provides other culture-sensitive data according to the user's locale, such as time, money, and number system representation. Not only this, internationalization also provides the conversion of various non-textual elements, embedded in a Web page, in the form relevant to the user.

While implementing internationalization, you need to create a property file for each language in which you want to display the Web pages. These property files are referenced by the Web pages to ascertain the specific locale or language customization before they are rendered. The property file contains titles, messages, and other text in a specific language. The naming format for the property file is `ResourceBundleName.properties`. For example, the name of a property file with English as default language would be `ApplicationResources.properties`.

Let us suppose your Web site is being accessed by two categories of users from different locales, Germany and United Kingdom (UK). You want to greet the German users in German and the UK users in English. To do this, you need to create two property files, one for an English greeting and the other for a German greeting. The Web page displaying the greeting message would first reference the property file and then render the greeting message in the required language, depending on the user details entered by the users.

Internationalization is implemented using a set of simple Java property files. Each file contains a key/value pair for each message in the language appropriate for the requesting client. For example, in the previous example, two different resource files representing different languages are created to greet the English and German users in their respective languages, as shown in the following code snippet:

```
ApplicationResources.properties // English user
ApplicationResources_de.properties // German user
```

Let's now explore how to implement internationalization in Struts 2.

Describing Internationalization and Localization

Prior to discussing about implementing internationalization in Struts 2, let's explore the concepts of internationalization and localization. Two terms—internationalization and localization are used to describe the internationalization feature of the Struts 2. You should have noticed that internationalization is often abbreviated as `i18n`, as there are 18 letters between the first letter *I*, and the last letter *n*. For the same reason, Localization is sometimes abbreviated as `l10n`. Let's explore these concepts in detail next.

Exploring Internationalization

Internationalization refers to the process of designing an application in such a manner that it can be adapted to various languages and regions without engineering changes. In this definition of internationalization, two terms need to be paid due attention to—adapting to changes and engineering changes. The first term, *adapting*, refers to the capability of an application to mould itself according to the culture of a user by representing all messages in the user's native language. Therefore, to internationalize the application would mean developing the application such that it has the capability to adapt according to the user's regional details.

The second term, *engineering changes*, refers to the changes in the code. The regional details are not hard-coded in the application, which means internationalization does not require engineering changes. The control logic of the application remains the same; only the view is changed.

The main theory behind the implementation of internationalization feature involves using a key whenever region-specific data is involved in an application. This key is similar to an indicator, which indicates that a region-sensitive data is being used in a Web page. When that page is displayed to the user, the key is replaced by its value. The value of the key depends on the language of the user. Therefore, there are different values of a key

for different languages. All the possible values for this key are available in a file called `Resource Bundle`, which is stored outside the source code.

When a session begins, the Web browser provides local details of the user to the Web server. According to these local details of the user, a corresponding resource bundle is selected. Wherever there is a key in the page, an appropriate value of the key is retrieved from the selected resource bundle. The retrieval of key-value and its insertion is dynamic. In other words, it takes place at runtime.

It is to be noted that the local details provided by the browser are stored in the server as objects of the `ActionContext class`. This detail is temporarily saved for a single user session only. For different user sessions, these user-details are saved as different `ActionContext`. That is, the objects of the `ActionContext` class are unique for every session.

The `I18n` interceptor is used to remember the locale selected for a user's session. With every HTTP request, a new thread is created, and the interceptor pushes the locale into the `ActionContext` associated to the thread. All the locale-sensitive methods utilize this locale.

The class associated with i18n inceptor is `com.opensymphony.xwork2.interceptor.I18nInterceptor`. This class is directly derived from the `com.opensymphony.xwork2.interceptor.AbstractInterceptor` class, which implements all the methods of the `com.opensymphony.xwork2.interceptor.Interceptor interface`, such as `init()`, `destroy()`, and `intercept()`.

Whenever an HTTP request is received, the I18n interceptor looks for the parameters in the request header and sets the locale according to the parameters. This locale remains the same for the entire session of the request. However, it is also possible to change the locale for a session dynamically, which allows the user to select a language of his/her choice at any point in the entire session.

Let's suppose the current locale for a user's session is for the default language. A request, `someAction?request_locale=en_EN`, made by a browser is processed by the interceptor to set the locale for the session to the English language. It is important to configure an interceptor before a request is processed by it.

The following code snippet shows the configuration of the `i18n` interceptor for an action class:

```
<package name="some-default" extends="struts-default">
  <action name="someAction" class="examples.SomeAction">
        <interceptor-ref name="i18n"/>
        <interceptor-ref name="basicStack"/>
        <result name="success">secondPage.jsp</result>
  </action>
</package>
```

In the preceding code snippet, the i18n interceptor is configured in the struts.xml file to intercept the action class called `SomeAction by` using the `<interceptor>` element.

The next example shows the setting of a parameter to be processed by the `i18n` interceptor. Now, we can pass a request parameter named `parameterName` to the `i18n` interceptor to set a new locale according to the value of this parameter, as shown in the following code snippet:

```
<package name="some-default" extends="struts-default">
  <action name="someAction" class="examples.SomeAction">
        <interceptor-ref name="i18n">
              <param name="parameterName">newlocale</param>
        </interceptor-ref >
        <interceptor-ref name="basicStack"/>

        <result name="success">secondPage.jsp</result>
  </action>
</package>
```

Let's now explore the concept of localization in detail.

Exploring Localization

Localization plays a major role in implementing internationalization. It is a process of adding locale-specific components, such as property files, in Web applications to customize them according to specific languages.

Locale is an object representing the regional settings of the user's machine. In the definition, the term to be focused is *locale-specific*, which emphasizes on those elements of the application that are dependent on the local settings of the user's machine.

The main purpose of localization is to translate the region-specific information in the application into the corresponding language and data format. This includes changing the language used, along with the language of the currency, the time and date format, and so on, into the local terms specified by the user's machine.

Global Resource Bundle

A resource bundle is a file that contains the key-value pairs for a particular language. Different resource bundles are required for different languages. A resource bundle file is created in a text editor and saved with the `.properties` extension. Therefore, resource bundles are also known as property files. As resource bundles are global to all classes, these are also known as global resource bundles. The global resource bundles are saved as a property file with a name, as shown in the following code snippet:

```
ResourceBundleName.properties
```

A property file for a language is distinguished from other property files by specifying a two-letter International Ogranization for Standards (ISO) language code in the file name. This code is appended with the file name before the file extension.

Let's consider the property files for different languages, as described in Table 20.36:

Table 20.36: Property Files for Different Languages

Property File Name	Description	File Content
ResourceBundle_fr.properties	Properties file for the French language	app.greeting= Bonjour app.username= usager nom app.submit= s'assujettir
ResourceBundle_es.properties	Properties file for the Spanish language	app. Greeting= Hola app.username= Nombre de Usuario app.submit=somete
ResourceBundle.properties	Properties file for default language, which is English	app.greeting=Welcome app.username=Username app.submit=SUBMIT

These entries tell Struts that when the user has a locale that uses the French language, and the key `app.greeting` is encountered in the code, the value `Bonjour` should be substituted at in place of the app.greeting key. Similarly, when the `app.username` key is encountered, the `usager nom` value must be substituted. Similarly, for the Spanish language Locale, `Hola` and `Nombre de Usuario` must be substituted in place of the `app.greeting` and `app.username` keys, respectively.

It is for the developer to decide how many property files to be included in an application. When the locale specific property file for a user is not available in a Web application, a default property file is used to customize Web pages. In this example, the default property file contains the values of the keys in the English language. Therefore, for the users other than French and Spanish, `Welcome` and `Username` will be substituted for every occurrence of the `app.greeting` and `app.username` keys in the code.

After the property files are created, the next step is to deploy these files in the framework. In Struts 2, the deployment of property files is comparatively easier than Struts 1. In Struts 2, the property files are stored in the `ApplicationName/WEB-INF/classes` folder. Struts 2 applications map the property files with the help of a file, called `struts.properties`. This file contains the location of the property files. The following code snippet shows the entry of a property file in the struts.properties file:

```
struts.custom.i18n.resources=NameOfPropertyFile
```

The preceding code line shows an entry in the `struts.property` file, which describes the global resource for an application.

It is also possible to specify more than one global resource bundles by separating the filenames using commas, as shown in the following code snippet:

```
struts.custom.i18n.resources= File_1, File_2
```

If more than one resource bundles are specified, the action class is associated with all the specified files, and searches for the required key-value in both the resource bundles.

Implementing Plugins in Struts 2

A plugin is a computer program that communicates with a main or host application. For example, a Web browser or an email program serves as a plugin that helps you to communicate with the application to provide a specific function, on request. The term plugin itself suggests that it is something, which can be plugged into the framework to introduce some extension points, such as new functionalities, classes, result types, interceptors, or packages into the existing framework. Struts 2 framework provides support for integration with other frameworks with the help of some plugins, such as Spring and JSF.

Therefore, these plugins are used to add extra functionality, function on demand, and component to make your Web application more efficient.

Plugins are used in Struts 2 applications with the help of JAR files available in the Struts framework. Some of these JAR files are struts2-tiles-plugin-2.0.6.jar, struts2-jsf-plugin-2.0.6, and struts2-spring-plugin-2.0.6, which are added into the applications' class path. The JAR file for a plugin may contain a `struts-plugin.xml` file, providing configuration information for the plugin. This `struts-plugin.xml` file follows the same pattern as that of the `struts.xml` file.

Let's now discuss the plugins available in Struts 2 framework in the following section.

Struts 2 Bundled Plugins

The plugins that come along with the distribution of the Struts 2 Framework are also known as Struts 2 bundled plugins. Table 20.37 describes bundled plugins in Struts 2:

Table 20.37: Bundled Plugins in Struts 2

PLUGIN	DESCRIPTION
Codebehind	Reduces the required configuration for the action class and the results by adding the Page Controller conventions.
Config Browser	Enables viewing of the configuration details such as action mapping, exception mapping, result mappings, and so on.
JasperReports	Enables the actions to generate reports through JasperReports.
JFreeChart	Enables the actions to return charts and graphs based on some data.
JSF	Provides support for Java Server Faces (JSF) components, and that too without any additional configuration.
Pell Multipart	Enables the Struts 2 Framework to use the Jason Pell's multipart parser, which is used to process the file uploads.
Plexus	Enables the creation and injection of actions, interceptors, and results by Plexus framework. Plexus is a dependency injection framework, such as Spring.
SiteGraph	Creates a graphical representation showing the flow amongst actions, interceptors, and results in a Struts 2 application.
SiteMesh	Provides support for features, such as headers and menu bars in Web pages.
Spring	Provides the support for creating actions, interceptors, and results by the Spring Framework.
Struts 1	Enables the use of Struts 1 actions and ActionForms in Struts2 applications.
Tiles	Provides a common look to the pages in the Web application by dividing pages into different fragments, known as `tiles`.

Table 20.37 describes various plugins available in Struts 2 framework. To understand the implementation of a plugin in a Struts 2 application, let's discuss about a commonly used and most popular plugin called Tiles plugin.

Tiles Plugin

The tiles plugin allows the actions to return tile pages containing different JSP pages. This is useful when a developer needs to implement a common look and feel among all the pages of an application. The tile layout is similar to a template layout.

Implementing Tiles Plugin

Let's now create an application to understand Struts 2 tiles plugin and to see how an action class can be integrated. Perform the following broad-level steps to create the Web application:

- Creating template
- Creating tiles enabled JSP pages
- Creating Action class
- Configuring struts.xml
- Creating tiles.xml
- Creating body.jsp
- Creating manager_menu.jsp
- Creating clerk_menu.jsp
- Configuring web.xml
- Exploring the directory structure of the application
- Packaging, running, and deploying the application

Creating Template

Template is a JSP page that defines a rectangular region to specify where other pages, such as header, footer, menu, and body should be positioned. A sample of template is shown in Figure 20.12:

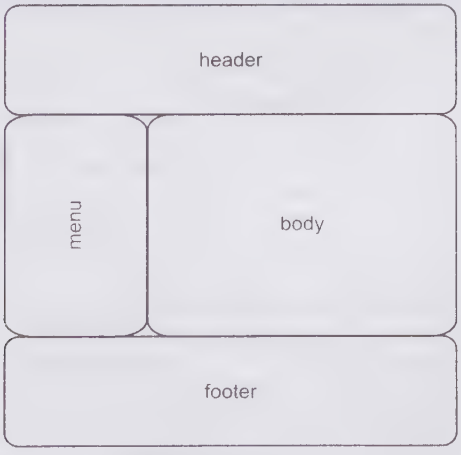


Figure 20.12: Displaying the Layout of a Template

In our application, template is represented by a JSP page, named layout.jsp. The code for layout.jsp is shown in Listing 20.17 (you can find the layout.jsp file on the CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.17: Displaying the Code of the layout.jsp Page

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<html>
  <head>
```

```
        <title>
                <tiles:getAsString name="title"/>
        </title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
    </head>
    <body topmargin="0">
        <table width="800" cellspacing="0" align="center" >
                <tr>
                        <td colspan="2" height="70">
                                <tiles:insertAttribute name="header"/>
                        </td>
                </tr>
                <tr height="300">
                        <td width="200" valign="top">
                                <tiles:insertAttribute name="menu"/>
                        </td>
                        <td width="600">
                                <tiles:insertAttribute name="body"/>
                        </td>
                </tr>
                <tr>
                        <td colspan="2" height="70">
                                <tiles:insertAttribute name="footer"/>
                        </td>
                </tr>
        </table>
    </body>
</html>
```

In Listing 20.17, the <tiles:getAsString/> and <tiles:insertAttribute/> tags are used to specify attributes, such as header, footer, menu, and body in layout.jsp page. These attributes are specified with values representing the name of JSP pages in the home.jsp page.

Creating Tiles Enabled JSPs

A JSP page can be easily changed to a tiled page (fragmented into tiles) using tiles tags. This type of page design using tiles tags helps in reducing duplication of code to design a number of pages having the same structure in an application. A tiles page can be created in the following two ways:

- Inserting template using <tiles:insertTemplate/>
- Inserting definition using <tiles:insertDefinition/>

The template is inserted into a JSP page (home.jsp) using the <tiles:insertTemplate/> tag. The attributes that are declared in the template (layout.jsp) are specified with a value that specifies the path of JSP pages using the <tiles:putAttribute/> tag in the home.jsp page. The code for the home.jsp file is shown in the Listing 20.18 (you can find the home.jsp file on the CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.18: Displaying the Code of the home.jsp File

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<tiles:insertTemplate template="/layout.jsp">
<tiles:putAttribute name="title"
value="www.simplylogin.com - Home page" type="string"/>
<tiles:putAttribute name="header" value="/header.jsp" />
<tiles:putAttribute name="menu" value="/mainmenu.jsp"/>
<tiles:putAttribute name="body" value="/mainbody.jsp"/>
<tiles:putAttribute name="footer" value="/footer.jsp"/>
</tiles:insertTemplate>
```

In Listing 20.18, the `<tiles:putAttribute/>` tag is used to fill all the four attributes declared in the layout.jsp page with a value specifying the path of JSP pages. The first attribute is of type String and the content directly gets displayed using <tiles:getAsString/> tag in the layout.jsp template. The basic structure of the home.jsp page is rendered by inserting the template (layout.jsp). Therefore, before you access home.jsp page, make sure that you have created all the four JSP pages, which are specified as values to the attributes, such as header, footer, menu, and body, in the home.jsp page. Therefore, let's create the following JSP pages before accessing the home.jsp file:

- header.jsp
- footer.jsp
- mainmenu.jsp
- mainbody.jsp

In addition, you also need to create login.jsp and loginform.jsp pages.

Creating header.jsp

The header.jsp creates a banner to be displayed in the Web pages. Listing 20.19 shows the code for header.jsp (you can find the header.jsp file on the CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.19: Displaying the Code of the header.jsp File

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<table width="100%" height="100%" bgcolor="#5a84da">
  <tr>
        <td align="left" valign="bottom" class="text">
        <s:date name="new java.util.Date()" format="dd MMMM yyyy"/> | </td>
        <td align="right">
        <h1 style="color:#ffffff;font-family:Book
        Antiqua">www.simplylogin.com</h1></td>
  </tr>
</table>
```

Creating footer.jsp

The content provided in the footer.jsp page is displayed at the bottom of the Web page. The code for the footer.jsp page is shown in Listing 20.20 (you can find the footer.jsp file on the CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.20: Displaying the Code of the footer.jsp File

```
<hr>
<div align=center style="font-size:10;letter-spacing:2">
All Rights are reserved with <br><b>Kogent Solutions Inc.</b><br>
G-2/16, Ansari Road, Daryaganj, New Delhi 110002.
</div>
</hr>
```

Creating mainbody.jsp

The mainbody.jsp is created to display textual content in the body of the Web pages. Listing 20.21 shows the code for the mainbody.jsp file (you can find the mainbody.jsp file on the CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.21: Displaying the Code of the mainbody.jsp File

```
<table width="400" align="center" cellpadding="10" bgcolor="#a362b3" >
<tr><td class="text">
<h2 align="center" style="font-family:Book Antiqua">A Tiles 2 Implementaion</h2>
<p align="justify">
This application is the result of Tiles 2 integration with
Struts 2 Framework. A Tiles Plugin is bundled with the Struts 2
distribution to support this integration.
The application uses Tiles 2 (version 2.0.1) and Struts 2 (version 2.0.6).
</p>
</td></tr>
</table>
```

Creating mainmenu.jsp

The mainmenu.jsp creates two hyperlinks named Home and Login. Listing 20.22 shows the mainmenu.jsp page providing these two hyperlinks (you can find the mainmenu.jsp file on CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.22: Displaying the Code of the mainmenu.jsp File

```
<p> </p>
<table width="150" cellpadding="3" cellspacing="0" align="center">
```

```
    <tr height="30" valign="middle">
          <td valign="middle" width="10">
                <img src="images/icon.gif">
          </td>
          <td align="left">
          <a href="home.jsp">Home</a></td>
    </tr>
    <tr height="30" valign="middle">
          <td valign="middle" width="10">
                <img src="images/icon.gif">
          </td>
          <td align="left">
          <a href="login.jsp">Login</a></td>
    </tr>
  </table>
```

The login hyperlink created in mainmenu.jsp is mapped to a JSP page called login.jsp. Let's now create login.jsp.

Creating login.jsp

The login.jsp is created to include definitions from login.def, which is configured in tiles.xml. The login.def contains view components, such as header.jsp, footer.jsp, mainmenu.jsp, and loginform.jsp.

Listing 20.23 shows code for the login.jsp page (you can find the login.jsp file on the CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.23: Displaying the Code of the login.jsp Page

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<tiles:insertDefinition name="login.def"/>
```

In Listing 20.23, the <tiles:insertDefinition/> element sets login.def from tiles.xml into login.jsp. Let's now create the loginform.jsp page.

Creating loginform.jsp

The loginform.jsp page simply gives a form with three input fields, which have been named as loginid, password, and type. The code for the loginform.jsp page is given in Listing 20.24 (you can find the loginform.jsp file on the CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.24: Displaying the Code of the loginform.jsp Page

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<table bgcolor="#a362b3" width="400" align="center" cellpadding="10">
<tr><td class="text" align="center">
| Login |
<br><s:actionerror/>
<s:form action="login" method="post" cssClass="text">
  <s:textfield name="loginid" key="app.loginid"/>
  <s:password name="password" key="app.password"/>
  <s:selectkey="app.type"name="type"list="#@java.util.HashMap@{'manager':
  'Manager', 'clerk':'Clerk'}"/>
  <s:submit value="Login"/>
</s:form>
</td></tr>
</table>
```

In Listing 20.24, the loginform.jsp uses Struts 2 tags to create a form and input fields.

Let's now create action classes for processing the loginform.jsp page.

Creating Action Class

The login.jsp page shows a login form, which, when submitted, has to be handled by some action class. The action class execution may give some result code, based on which the next view is decided. Listing 20.25 shows the code for the LoginAction class (you can find the LoginAction.java file on CD in the `code\JavaEE\Chapter20\Struts2Tiles\src\com\kogent\action` folder):

Listing 20.25: Displaying the Code of the LoginAction.java File

```
package com.kogent.action;
import com.opensymphony.xwork2.ActionSupport;
```

```
public class LoginAction extends ActionSupport {
String loginid;
String password;
String type;
public String getLoginid() {
   return loginid;
}
public void setLoginid(String loginid) {
   this.loginid = loginid;
}
public String getPassword() {
   return password;
}
public void setPassword(String password) {
   this.password = password;
}
public String getType() {
   return type;
}
public void setType(String type) {
   this.type = type;
}
public String execute() throws Exception {
   if(loginid.equals(password)){
   if("manager".equals(type)){
   return "manager";
}else{
return "clerk";
}
}else{
   this.addActionError(getText("app.invalid"));
   return ERROR;
   }
}
public void validate() {
   if ( (loginid == null ) || (loginid.length() == 0) ) {
   this.addFieldError("loginid", getText("app.loginid.blank"));
}
if ( (password == null ) || (password.length() == 0) ) {
   this.addFieldError("password", getText("app.password.blank"));
   }
}
}
```

In Listing 20.25, the possible result codes returned by the LoginAction class are manager, clerk, error, and input.

Configuring struts.xml

Let's see the action mapping provided in struts.xml file to process the login form submission. Listing 20.26 shows the code for the action mapping for login action (you can find the struts.xml file on CD in the `code\JavaEE\Chapter20\Struts2Tiles\WEB-INF\classes` folder):

Listing 20.26: Displaying the Code of the struts.xml File

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.1//EN"
"http://struts.apache.org/dtds/struts-2.1.dtd ">
<struts>
   <package name="default" extends="tiles-default">
   <action name="login" class="com.kogent.action.LoginAction">
   <result name="error">/login.jsp</result>
   <result name="input" type="tiles">login.def</result>
   <result name="manager"
   type="tiles">manager.welcome.def</result>
   <result name="clerk"
   type="tiles">clerk.welcome.def</result>
```

```
    </action>
    </package>
</struts>
```

In Listing 20.26, there are four results named error, input, manager, and clerk, configured with an action named login. If a JSP page is being used as the location, the result type should be the default dispatcher. However, the powerful feature added by tiles plugin is the use of tiles as result type, which helps in rendering a page definition from the tiles configuration file. We can use a JSP page and a page definition, instead of generating a view to the user, as shown in the following code snippet:

```
<result name="error">/login.jsp</result>
<result name="input" type="tiles">login.def</result>
```

In the preceding code snippet, the result named input is configured to read definition from login.def. We can, instead, give a page definition easily in tiles.xml file. The action may return two more result codes other than input and error. These result codes are manager and clerk, which are returned according to the type selected by the user while logging. The user should get two different consoles as their type is different. This can be implemented by giving new definitions using the same template, but different attributes. The two new definitions used are manager.welcome.def and clerk.welcome.def. The following code snippet shows the two results, manager and clerk, being configured in the struts.xml (Listing 20.26):

```
<result name="manager" type="tiles">manager.welcome.def</result>
<result name="clerk" type="tiles">clerk.welcome.def</result>
```

We need to provide these new definitions in the tiles.xml file to generate an appropriate view for two different types of users. Note that we only need to define different attributes to be used in these two definitions.

Creating tiles.xml

All the definitions used in this application are provided in tiles.xml file. Therefore, let's create the tiles.xml file and save it in the WEB-INF folder. Listing 20.27 shows the code for the tiles.xml file (you can find the tiles.xml file on CD in the `code\JavaEE\Chapter20\Struts2Tiles\WEB-INF` folder):

Listing 20.27: Displaying the Code of the tiles.xml File

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
  "-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
  "http://tiles.apache.org/dtds/tiles-config_2_1.dtd">
<tiles-definitions>
  <definition name="login.def" template="/layout.jsp">
<put-attribute name="title"value=www.simplylogin.com/Login type="string"/>
        <put-attribute name="header" value="/header.jsp"/>
        <put-attribute name="menu" value="/mainmenu.jsp"/>
        <put-attribute name="body"   value="/loginform.jsp"/>
        <put-attribute name="footer" value="/footer.jsp"/>
  </definition>
  <definition name="manager.welcome.def" template="/layout.jsp">
        <put-attribute name="title" value="www.simplylogin.com/Login
        Successfull - Manger" type="string"/>
        <put-attribute name="header" value="/header.jsp"/>
        <put-attribute name="menu" value="/manager_menu.jsp"/>
        <put-attribute name="body"   value="/body.jsp"/>
        <put-attribute name="footer" value="/footer.jsp"/>
  </definition>
  <definition name="clerk.welcome.def" extends="manager.welcome.def">
        <put-attribute name="title" value="www.simplylogin.com/Login
        Successfull - Clerk" type="string"/>
        <put-attribute name="menu" value="/clerk_menu.jsp"/>
  </definition>
  <definition name="manager.def" extends="manager.welcome.def">
        <put-attribute name="title" value="www.simplylogin.com -
        Manager" type="string"/>
        <put-attribute name="body"   value="/actionbody.jsp"/>
  </definition>
  <definition name="clerk.def" extends="manager.def">
<put-attribute name="title" value="www.simplylogin.com - Clerk" type="string"/>
        <put-attribute name="menu" value="/clerk_menu.jsp"/>
```

```
    </definition>
</tiles-definitions>
```

In Listing 20.27, `login.def` is created in the tiles.xml. This definition uses the template called layout.jsp and fills all its attributes with different view components, such as header.jsp, mainmenu.jsp, loginform.jsp, and footer.jsp. The definition, `manager.welcome.def` is also filled with all the required attributes. The `clerk.welcome.def` definition is created by extending the `manager.welcome.def` definition, and all the attributes from `manager.welcome.def` are filled with values according to the user.

The definition manager.def is created by extending manager.welcome.def definition. Similarly, clerk.def is created by extending the manager.def. These definitions use some previously created and used JSP pages. Now, let's create the following JSP pages:

- body.jsp
- manager_menu.jsp
- clerk_menu.jsp

Creating body.jsp

The body.jsp is created to display links and content according to the `Type` specified by the user, which is either `manager` or `clerk`. Listing 20.28 shows the code for the body.jsp page (you can find the body.jsp file on CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.28: Displaying the Code of the body.jsp File

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<table bgcolor="#a362b3" width="400" align="center" cellpadding="10">
  <tr><td class="text" align="center">
        | Successful Login!|
        <br><br><br><br>
        You have logged in as : <b><s:property value="loginid"/></b>
        <br><br><br><br>
        | Use Navigation Links provided in Menu Bar. |
  </td></tr>
</table>
```

The manager_menu.jsp and clerk_menu.jsp pages provide different sets of navigational links and are designed for two different types of user.

Creating manager_menu.jsp

The manager_menu.jsp is displayed only to the users belonging to type `manager`. Listing 20.29 shows the code for the manager_menu.jsp page (you can find the manager_menu.jsp file on CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.29: Displaying the Code of the manager_menu.jsp File

```
<p> </p>
<table width="150" cellpadding="3" cellspacing="0" align="center">
  <tr height="30" valign="middle">
        <td valign="middle" width="10">
              <img src="images/icon.gif">
        </td>
        <td align="left"><a href="home.jsp">Home</a></td>
  </tr>
  <tr height="30" valign="middle">
        <td valign="middle" width="10">
              <img src="images/icon.gif">
        </td>
        <td align="left"><a href="manager.action">New</a></td>
  </tr>
  <tr height="30" valign="middle">
        <td valign="middle" width="10">
              <img src="images/icon.gif">
        </td>
  <td align="left"><a href="manager.action">Edit</a></td></tr>
  <tr height="30" valign="middle">
        <td valign="middle" width="10">
```

```
                <img src="images/icon.gif">
            </td>
            <td align="left"><a href="manager.action">Delete</a></td>
        </tr>
        <tr height="30" valign="middle">
            <td valign="middle" width="10">
                <img src="images/icon.gif">
            </td>
            <td align="left"><a href="logoff.action">Logoff</a></td>
        </tr>
    </table>
```

Creating clerk_menu.jsp

The clerk_menu.jsp is displayed only to the users belonging to type `clerk`. Listing 20.30 shows the code for the clerk_menu.jsp page (you can find the clerk_menu .jsp file on CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.30: Displaying the Code of the clerk_menu.jsp File

```
<p> </p>
<table width="150" cellpadding="3" cellspacing="0" align="center">
    <tr height="30" valign="middle">
        <td valign="middle" width="10">
            <img src="images/icon.gif">
        </td>
        <td align="left"><a href="home.jsp">Home</a></td>
    </tr>
    <tr height="30" valign="middle">
        <td valign="middle" width="10">
            <img src="images/icon.gif">
        </td>
        <td align="left"><a href="clerk.action">Deposit</a></td>
    </tr>
    <tr height="30" valign="middle">
        <td valign="middle" width="10">
            <img src="images/icon.gif">
        </td>
        <td align="left"><a href="clerk.action">Withdraw</a></td>
    </tr>
    <tr height="30" valign="middle">
        <td valign="middle" width="10">
            <img src="images/icon.gif">
        </td>
        <td align="left"><a href="clerk.action">Transfer</a></td>
    </tr>
    <tr height="30" valign="middle">
        <td valign="middle" width="10">
            <img src="images/icon.gif">
        </td>
        <td align="left"><a href="logoff.action">Logoff</a></td>
    </tr>
</table>
```

Configuring the web.xml File

In Struts 2 applications, all the requests are mapped to a filter class named `FilterDispatcher`. In addition to this, we also need to register a listener provided by the tiles plugin to integrate tiles with Struts 2. The class `org.apache.struts2.tiles.StrutsTilesListener` provides a better support for Struts 2 features and, therefore, it is preferred over the traditional TilesListener. The configuration for the filters and listeners is provided in the web.xml file, shown in Listing 20.31 (you can find the web.xml file on CD in the `code\JavaEE\Chapter20\Sturts2Tiles\WEB-INF` folder):

Listing 20.31: Displaying the Code for the web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
```

```
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_4.xsd">
  <display-name>Tiles 2 Plugin Example</display-name>
  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <listener>
  <listener-class>org.apache.struts2.tiles.StrutsTilesListener</listener-class>
  </listener>
  <welcome-file-list>
  <welcome-file>home.jsp</welcome-file>
  </welcome-file-list>
  </web-app>
```

In Listing 20.31, we have not provided any context parameter to configure the CONTEXT_FACTORY and tiles configuration file. Therefore, the defaults are implied in Listing 20.31. The default tiles configuration file is tiles.xml, and the context factory used is StrutsTilesContainerFactory.

Save the files according to the directory structure that is discussed in the following section.

Exploring the Directory Structure of the Application

A directory structure of an application depicts the location where all the required files are stored. The directory structure of the application being developed is shown in Figure 20.13:

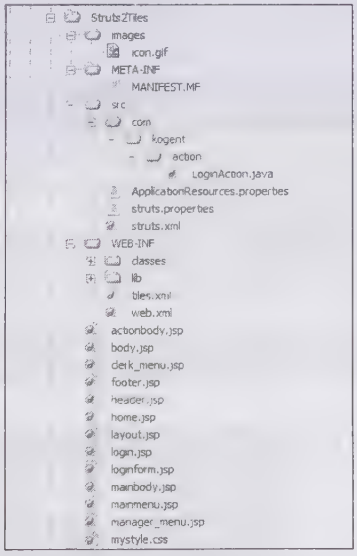


Figure 20.13: Displaying the Directory Structure of Struts 2Tiles

Prior to packaging our application, we must have all the required JAR files in our WEB-INF/lib folder. Therefore, add the following JAR files to your WEB-INF/lib folder:

- tiles-core-2.0.6.jar
- tiles-api-2.0.6.jar
- commons-beanutils-1.7.0.jar
- commons-digester-2.0.jar

- commons-logging-api-1.1.jar
- struts2-tiles-plugin-2.1.8.1.jar
- struts2-core-2.1.8.1.jar
- xwork-core-2.1.6.jar
- freemarker-2.3.15.jar
- ognl-2.7.3.jar

Let's now package, deploy, and run the application.

Packaging, Deploying, and Running the Application

We will now create Struts2Tiles.war and deploy it on the Glassfish application server. Browse th `http://localhost:8080/Struts2Tiles` URL to run the application, as shown in Figure 20.14:

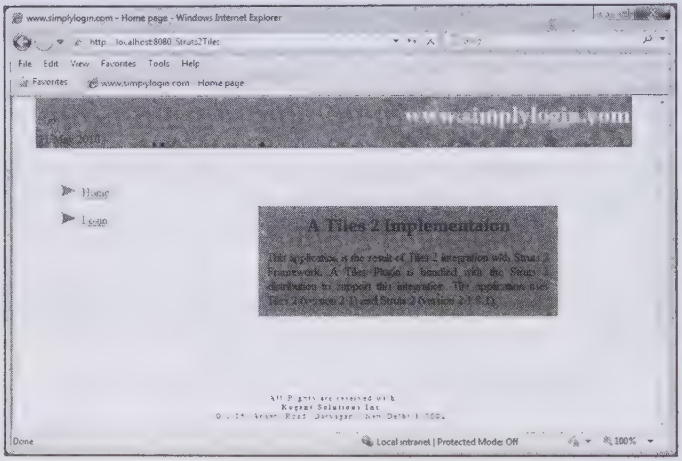


Figure 20.14: Displaying the Output of the home.jsp Page

The home.jsp page is displayed, by default, as this page is configured as the welcome page of the Struts2Tile application. Click the `Login` hyperlink, shown in Figure 20.14. The output of the login.jsp page, which uses th definition login.def, is shown in Figure 20.15:

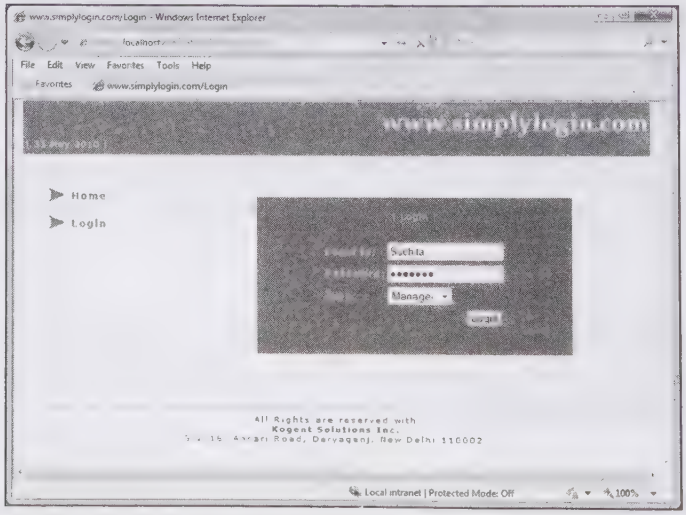


Figure 20.15: Displaying the Output of the login.jsp Page

Enter the login id and password and select `Manager` as type of user. This results in the rendering of the definition manager.welcome.def with its output shown in Figure 20.16. Observe the list of options shown in the menu bar created using manager_menu.jsp page:

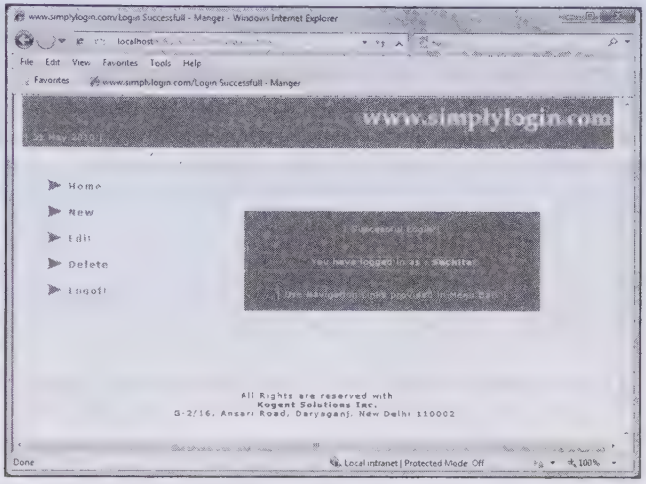


Figure 20.16: Displaying the Output of the manager.welcome.def Definition

Similar to the manager type, you can select `Clerk` as the type of user. After selecting the Clerk type and clicking the Login button, you get a different output using clerk.welcome.def definition, which uses a different value for menu attribute and the list of options, as displayed in Figure 20.17:

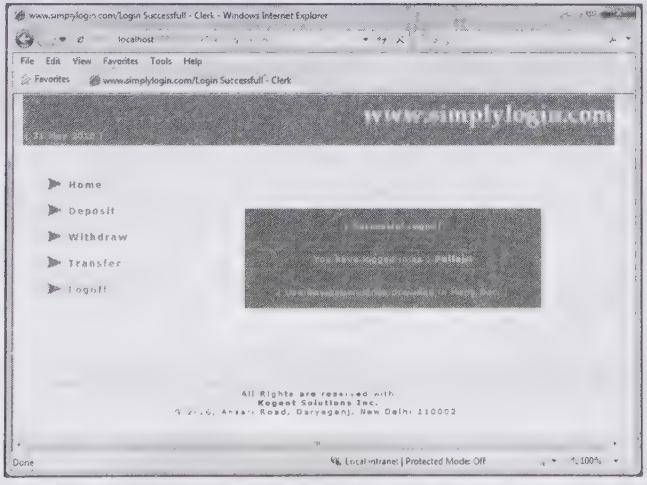


Figure 20.17: Displaying the Output of the clerk.welcome.def Definition

Now, compare Figure 20.16 and Figure 20.17. These figures are designed using the same template layout.jsp and, therefore, have a similar structure. They both use reusable tiles, such as header, footer, and body. However, they are different for the title and menu attributes, which simply make the two pages distinct from each other, even though the look and feel of the pages are same. This is done by adding few definitions in the struts.xml file as compared to creating new JSP pages for `clerk` and `manager` actions. The Home, Deposit, Withdraw, Transfer, and Logoff hyperlinks provided in Figure 20.17 may not be working; so to activate them, we can provide some new action mappings and new page definitions. The following code snippet shows the new action mappings added in the struts.xml file (Listing 20.26):

```
<action name="clerk">
  <result type="tiles">clerk.def</result>
```

```
  </action>
  <action name="manager">
    <result type="tiles">manager.def</result>
  </action>
  <action name="logoff">
    <result type="tiles">login.def</result>
  </action>
```

The new definitions required are clerk.def and manager.def, which can be added to our tiles.xml file similar t other definitions added.

The two definitions to be added to the tiles.xml file are given in the following code snippet (Listing 20.27):

```
<definition name="manager.def" extends="manager.welcome.def">
<put-attribute name="title" value="www.simplylogin.com -
Manager" type="string"/>
<put-attribute name="body" value="/actionbody.jsp"/>
</definition>
<definition name="clerk.def" extends="manager.def">
<put-attribute name="title" value="www.simplylogin.com - Clerk" type="string"/>
<put-attribute name="menu" value="/clerk_menu.jsp"/>
</definition>
```

The only new view component used in these definitions is actionbody.jsp. Create this JSP page before using new definitions. Listing 20.32 shows actionbody.jsp (you can find the actionbody.jsp file on CD in the `code\JavaEE\Chapter20\Struts2Tiles` folder):

Listing 20.32: Displaying the Code for the actionbody.jsp Page

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<table bgcolor="#a362b3" width="400" align="center" cellpadding="10">
  <tr><td class="text" align="center">
        <br><br>
              <h2>| Under Construction |</h2>
        <br><br><br><br>
        This page is underconstruction. <br>Please try after some time.
  </td></tr>
</table>
```

Click the `Logoff` hyperlink, shown in Figures 20.16 and 20.17, to return to the page rendered using definitions login.def.

The new menu options, shown in Figures 20.16 (New, Edit, and Delete) and 20.17 (Deposit, Withdraw, and Transfer), can be clicked to invoke actions named `manager` and `clerk`, respectively, which are configured in struts.xml. Figure 20.18 shows the Web page displayed on the basis of the new actions using the `manager.def` definition:

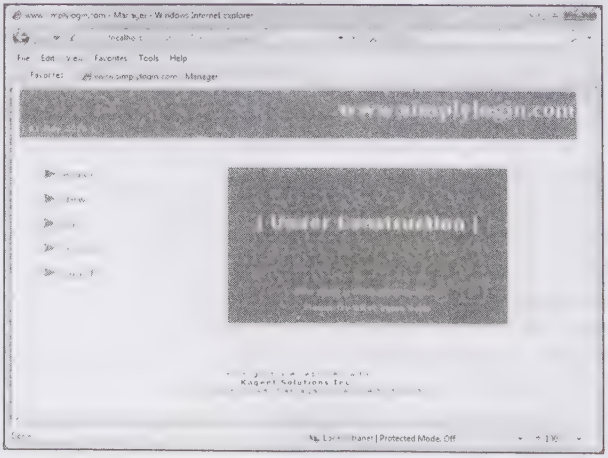


Figure 20.18: Displaying the Output of manager.welcome.def

In this application, we have implemented Struts 2 tiles plugin to integrate our Struts 2 based application with tiles. This integration results in the development of Web pages having a consistent look and feel throughout the application. The Web page development using tiles surely makes life easy for Web page designers, as it reduces the duplicated codes and thereby the efforts to design a new page.

Integrating Struts 2 with Hibernate

Struts 2 framework does not support the data persistence and Object Relational Mapping features in a Struts 2 application. Integrating Struts 2 with Hibernate incorporates these features in an application. Hibernate is an Object Oriented Mapping technology that maps object view of data to a relational database. In addition, Hibernate allows you to perform various database operations, such as create, read, update, and delete (CRUD) to store and maintain data persistently in the database.

While integrating Struts 2 with Hibernate in an application, we need to include the relevant JAR files in the lib directory of the application. The relevant JAR files are shown in Figure 20.19:

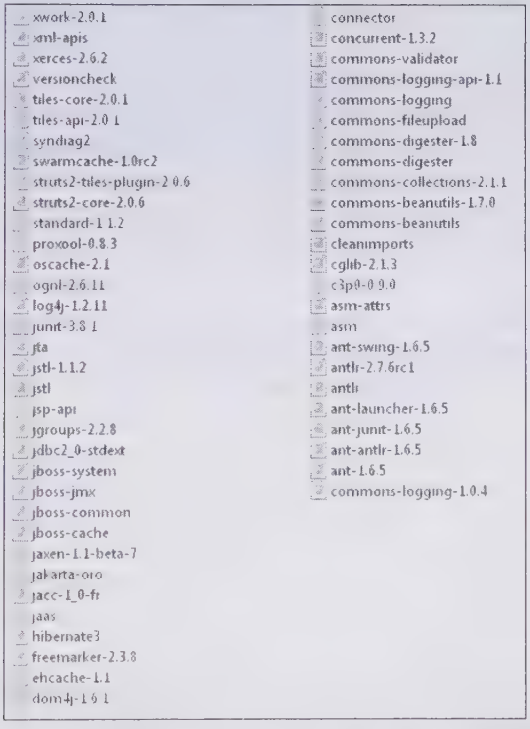


Figure 20.19: JAR files for integrating Struts 2 and Hibernate

NOTE

For more information about Hibernate, refer Chapter 15, Implementing Java Persistence Using Hibernate.

In this section, we create an application, `StrutsHibernateApp`, demonstrating the integration of Struts 2 and Hibernate 3. In `StrutsHibernateApp`, a user enters the text to be searched and clicks the `Search` button. The Struts framework then processes the request and passes the result to the action class for further processing.

To create `StrutsHibernateApp`, perform the following broad-level steps:

- Set the MySQL Database and table
- Configure Hibernate
- Develop Hibernate Struts Plugin

Setting the MySQL Database and Table

In this application, we are using the MySQL database; you need to enter the valid username and password to access the database. By default, the password of MySQL is `root`. Now, let's create a database, `strutshibernate`, for the `StrutsHibernateApp` application. The `StrutsHibernateApp` application searches for title entered by the user in the books table of the `strutshibernate` database. The books table of the `strutshibernate` database contains relevant book details, such as book id, author, and description of the book. Listing 20.33 provides the SQL script for setting up the database and table:

Listing 20.33: Displaying the Code of the SQL Script for StrutsHibernateApp

```
Create database strutshibernate;
Use strutshibernate;
Create table books(
id int not null auto_increment,
title varchar(50) not null,
description varchar(50) not null,
author varchar(50) not null,
primary key ( id )
) type = myisam;

insert into books values (1, 'JSP', 'Discussing JSP and Servlets', 'Suchita');
insert into books values (2, 'Struts', 'Integrating Struts with Hibernate', 'Vikash');
insert into books values (3, 'AJAX', 'Understanding AJAX', 'Deepak');
insert into books values (4, 'Servlets', 'Understanding Servlets', 'Shalini');
```

Listing 20.33 creates a database, strutshibernate, and a table named books, having various fields, such as id, title, description, and author. The values are also inserted in the books table, as shown in Listing 20.33.

Now, after creating the database and table, let's configure Hibernate.

Configuring Hibernate

The Hibernate configuration file for the application, hibernate.cfg.xml, is used to provide information required to establish a connection with the database. The configuration details used to map the domain objects to database table are also provided in the hibernate.cfg.xml file. Let's perform the following operations to configure Hibernate:

- Create the hibernate.cfg.xml file
- Create the Books.hbm.xml file
- Create the Book JavaBean

Creating the hibernate.cfg.xml File

Let's create the hibernate.cfg.xml file to provide database connection details. Listing 20.34 shows the code for the hibernate.cfg.xml file:

Listing 20.34: Displaying the Code for the hibernate.cfg.xml file

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver
</property>
<property name="hibernate.connection.url">jdbc:mysql://localhost/struts-
 hibernate</property>
<property name="hibernate.connection.username">root</property>
<property name="hibernate.connection.password"></property>
<property name="hibernate.connection.pool_size">10</property>
<property name="show_sql">true</property>
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>
<property name="hibernate.hbm2ddl.auto">update</property>
<!-- Mapping files -->
<mapping resource="/com/kogent/dao/hibernate/Books.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

In Listing 20.34, the <mapping resource=" "> tag is used to specify the mapping of the Books.hbm.xml file. Save the hibernate.cfg.xml file in the \StrutsHibernateApp\src\ folder.

Now, let's create the Books.hbm.xml file.

Creating the Books.hbm.xml File

The hibernate.cfg.xml file maps the Books.hbm.xml file for configuring the columns of the books table. The Books.hbm.xml file provides the Hibernate mapping for the fields of the books table. Listing 20.35 shows the code for the Hibernate mapping in the Books.hbm.xml file:

Listing 20.35: Displaying the Code of the Books.hbm.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping auto-import="true" default-lazy="false">
<class name="com.kogent.dao.hibernate.Books" table="books">
  <id name="id" type="java.lang.Integer" column="id">
        <generator class="increment" />
  </id>
<property name="title" type="java.lang.String" column="title" not-null="true"
  length="50"/>
  <property name="description" type="java.lang.String" column="description"
not-null="true" length="50"/>
  <property name="author" type="java.lang.String" column="author" not-null="true"
  length="50"/>
</class>
</hibernate-mapping>
```

Listing 20.35 specifies the mapping for each column of the books table and configures the Books JavaBean class. Now, let's create the Books JavaBean object.

Creating the Books JavaBean

In this subsection, we create a JavaBean named Books, which is used to store and retrieve the book details from the database. Listing 20.36 shows the code for the Books JavaBean:

Listing 20.36: Displaying the Code for the Books.java File

```
package com.kogent.dao.hibernate;
import java.io.Serializable;
public class Books implements Serializable
{
   private Integer id;
   private String title;
   private String description;
   private String author;
   public Books(Integer id, String title, String description, String author)
   {
         this.id = id;
         this.title = title;
         this.description = description;
         this.author = author;
   }
   public Books()
   {
   }
   public Integer getId()
   {
         return this.id;
   }
   public void setId(Integer id)
   {
         this.id = id;
   }
   public String getTitle()
   {
         return this.title;
   }
   public void setTitle(String title)
   {
```

```
        this.title = title;
    }
    public String getDescription()
    {
        return this.description;
    }
    public void setDescription(String description)
    {
        this.description = description;
    }
    public String getAuthor()
    {
        return this.author;
    }

    public void setAuthor(String author)
    {
        this.author = author;
    }
}
```

Listing 20.36 provides setter and getter methods for the id, title, description, and author fields of the books table.

Developing Struts Hibernate Plugin

We now develop the Struts Hibernate Plugin. This plugin creates the Hibernate Session factory and caches the session details in the servlet context. This strategy of creating Hibernate Session factory enhances the performance of the application. Listing 20.37 shows the code required to integrate Struts and Hibernate:

Listing 20.37: Displaying the Code for the Struts Hibernate Plugin

```
package com.kogent.plugin;
//import required packages
public class HibernatePlugIn implements PlugIn
{
  private String configfilepath = "/hibernate.cfg.xml";
  public static final String SESSIONsessionfactory_KEY =
        SessionFactory.class.getName();

  private SessionFactory sessionfactory = null;

  public void destroy()
  {
        try {
              sessionfactory.close();
        }catch(HibernateException e)
        {
              System.out.println("Unable to close Hibernate
               Session Factory: " + e.getMessage());
        }
  }
 public void init(ActionServlet servlet, ModuleConfig config) throws ServletException {
        System.out.println("************************************");
        System.out.println("**** Initilizing HibernatePlugIn   **********");
        Configuration configuration = null;
        URL configFileURL = null;
        ServletContext context = null;

        try
        {
        configFileURL = HibernatePlugIn.class.getResource(configfilepath);
        context = servlet.getServletContext();
        configuration = (new Configuration()).configure(configFileURL);
        sessionfactory = configuration.buildSessionFactory();
        context.setAttribute(SESSIONsessionfactory_KEY, sessionfactory);
        }catch(HibernateException e)
        {
              System.out.println("Error while initializing hibernate: " +
                    e.getMessage());
        }
        System.out.println("************************************"); }
  public void setConfigFilePath(String configFilePath)  {
  if ((configFilePath == null) || (configFilePath.trim().length() == 0))
```

```
    {
            throw new IllegalArgumentException("configFilePath cannot be
            blank or null.");
    }
    System.out.println("Setting 'configFilePath' to '"  +
    configFilePath + "'...");
    configfilepath = configFilePath;
    }
}
```

In Listing 20.37, the configfilepath variable stores the name of the Hibernate configuration file and defines `SESSION_FACTORY_KEY` to store the session factory instance in the servlet context. Therefore, during the startup of the application, the init() method is invoked, which initializes the session factory and caches the session factory instance in the servlet context. The Struts Hibernate plugin created in Listing 20.37 also needs to be configured in the struts-config.xml file, as shown in the following code snippet:

```
<plug-in classname="com.kogent.plugin.HibernatePlugIn"></plug-in>
```

Summary

This chapter discusses the need for Struts2 framework for creating an enterprise-ready Java Web application. It introduces the concept of action classes with the mechanism to configure these classes using various configuration files and also the concept of interceptors, along with the support of OGNL in Struts2. It further discusses the mechanism to perform validation in Struts 2 and an in-built feature of Struts2, known as internationalization. The implementation of tiles plugin in Struts2 is also discussed in this chapter.

In the next chapter, we learn about another interesting framework of Java, known as Spring 3.0.

Quick Revise

Q1. What is Struts 2?

Ans. Struts 2 is an open source framework used to create Java Web applications based on MVC architecture.

Q2. Explain the work flow of an application in Struts 2.

Ans. The steps for the work flow in Struts 2 are as follows:

- The user sends a request through a user interface provided by the view, which further passes this request to the controller represented by the FilterDispatcher class in Struts 2.
- The controller servlet filter receives the input request coming from the user through the interface provided by the view, instantiates an object of the suitable action class, and executes different methods over this object.
- If the state of model is changed, all the associated views are notified about the changes.
- Then the controller selects the new view to be displayed according to the result code returned by the action class.
- The view presents the user interface. The view queries about the state of the model to show the current data, which is retrieved from the action class.

Q3. List the steps used by Struts 2 to process a client's request.

Ans. Struts 2 processes a request in the following steps:

- Receiving of request
- Pre-processing by Interceptors
- Invoking the Action class method
- Invoking the Result class
- Processing by Interceptors
- Responding to the user

Q4. What is IoC?

Ans. IoC is a programming design pattern used to reduce copuling in programs and is implemented by using the `ObjectFactory` factory.

Q5. List the components of Struts 2 based applications.

Ans. The following are the components of Struts 2 based applications:

- Controller
- Struts-config.xml
- Struts.xml
- Action classes
- Model
- View

Q6. List the different types of annotations in Struts2.

Ans. Annotations in Struts 2 can be generally divided into four different types:

- Action annotation
- Interceptor annotation
- Validation annotation
- Type Conversion annotation

Q7. What is the use of Action interface in Struts 2?

Ans. Struts 2 framework provides Action interface that can be used to create action classes to simplify the code in the action classes.

Q8. What is Dependency Injection and Inversion of Control?

Ans. Dependency Injection (DI) and Inversion of Control (IoC) are programming design patterns that help to reduce coupling in a program.

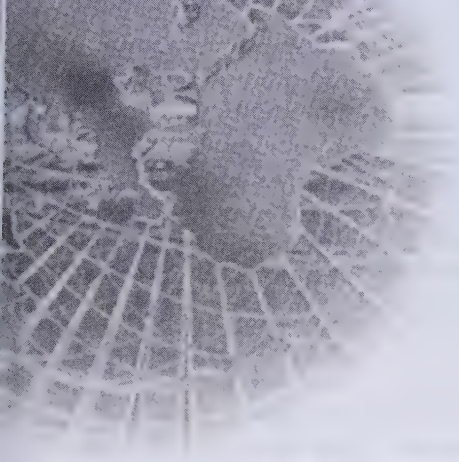
Q9. List the common aware interfaces supported by Struts 2.

Ans. The common aware interfaces that Struts 2 supports are:

- The ApplicationAware Interface
- The ParameterAware Interface
- The ServletRequestAware Interface
- The ServletResponseAware Interface
- The SessionAware Interface

Q10. What is OGNL?

Ans. Object Graph Navigation Language (OGNL) is an expression language used to manipulate and retrieve different properties of Java objects.



21

Working with Spring 3.0

Prior to the advent of Enterprise Java Beans (EJB), Java developers needed to use JavaBeans to create Web applications. Although JavaBeans helped in the development of user interface (UI) components, they were not able to provide services, such as transaction management and security, which were required for developing robust and secure enterprise applications. The advent of EJB was seen as a solution to this problem. EJB extends the Java components, such as Web and enterprise components, and provides services that help in enterprise application development. However, developing an enterprise application with EJB was not easy, as the developer needed to perform various tasks, such as creating Home and Remote interfaces and implementing lifecycle callback methods, which lead to complexity of providing code for EJBs. Due to this complication, developers started looking for an easier way to develop enterprise applications.

The Spring framework has emerged as a solution to all these complications. This framework uses various new techniques, such as Aspect-Oriented Programming (AOP), Plain Old Java Object (POJO), and dependency injection (DI), to develop enterprise applications, thereby removing the complexities involved while developing enterprise applications using EJB. Spring is an open-source, lightweight framework that allows Java EE 5 developers to build simple, reliable, and scalable enterprises applications. This framework mainly focuses on providing various ways to help you manage your business objects. It makes the development of Web applications much easier as compared to classic Java frameworks and Application Programming Interfaces (APIs), such as Java Database Connectivity (JDBC), JavaServer Pages (JSP), and Java Servlet.

This chapter provides an overview of the Spring framework and describes its architecture. Various features of Spring have also been discussed in this chapter. In addition, the chapter explores Spring Inversion of Control (IoC), Spring AOP, and transaction management. This chapter also discusses about the Spring Web Model View Controller (MVC) Framework along with its implementation using a simple Spring-based application.

Introducing Features of the Spring Framework

The Spring framework can be considered as a collection of subframeworks, also called layers, such as Spring AOP, Spring Object-Relational Mapping (Spring ORM), Spring Web Flow, and Spring Web MVC. You can use any of these modules separately while constructing a Web application. The modules may also be grouped together to provide better functionalities in a Web application.

The features of the Spring framework, such as IoC, AOP, and transaction management, make it unique among the list of frameworks. Some of the most important features of the Spring framework are as follows:

- **IoC container**—Refers to the core container that uses the DI or IoC pattern to implicitly provide an object reference in a class during runtime. This pattern acts as an alternative of service locator pattern. The IoC container contains assembler code that handles the configuration management of application objects.

  The Spring framework provides two packages, namely, `org.springframework.beans` and `org.springframework.context`, which helps in providing the functionality of the IoC container. We will discuss more about the IoC container in the *Exploring Dependency Injection and Inversion of Control* section of this chapter.

- **Data access framework**—Allows the developers to use persistence APIs, such as JDBC and Hibernate, for storing persistence data in database. It helps in solving various problems of the developer, such as how to interact with a database connection, how to make sure that the connection is closed, how to deal with exceptions, and how to implement transaction management. It also enables the developers to easily write code to access the persistence data throughout the application.

- **Spring MVC framework**—Allows you to build Web applications based on MVC architecture. All the requests made by a user first go through the controller and are then dispatched to different views, that is, to different JSP pages or Servlets. The form-handling and form-validating features of the Spring MVC framework can be easily integrated with all popular view technologies such as JSP, JasperReport, FreeMarker, and Velocity.

- **Transaction management**—Helps in handling transaction management of an application without affecting its code. This framework provides Java Transaction API (JTA) for global transactions managed by an application server and local transactions managed by using the JDBC, Hibernate, Java Data Objects (JDO), or other data access APIs. It enables the developer to model a wide range of transactions on the basis of

Spring's declarative and programmatic transaction management. We will discuss declarative and programmatic transactions later in this chapter.

- **Spring Web Service**—Generates Web service endpoints and definitions based on Java classes, but it is difficult to manage them in an application. To solve this problem, Spring Web Service provides layered-based approaches that are separately managed by Extensible Markup Language (XML) parsing (technique of reading and manipulating XML). Spring provides effective mapping for transmitting incoming XML message request to any object, and helps the developer to easily distribute XML message (object) between two machines. Spring Web Services are distributed separately, which can be downloaded from the Spring framework website (http://static.springframework.org/spring-ws/site/downloads/releases.html).
- **JDBC abstraction layer**—Helps the users in handling errors in an easy and efficient manner. The JDBC programming code can be reduced when this abstraction layer is implemented in a Web application. This layer handles exceptions such as DriverNotFound. All `SQLExceptions` are translated into the `DataAccessException class`. Spring's data access exception is not JDBC specific and hence Data Access Objects (DAO) are not bound to JDBC only.
- **Spring TestContext framework**—Provides facilities of unit and integration testing for the Spring applications. Moreover, the Spring `TestContext` framework provides specific integration testing functionalities such as context management and caching, DI of test fixtures, and transactional test management with default rollback semantics.

What's New in Spring 3.0

Various revisions have been introduced for the Spring framework, such as Spring 2.0 released in October 2006, Spring 2.5 released in November 2006; and we have Spring 3.0 as the latest revision. The Spring 3.0 framework is released with the support for Java 5; therefore, Spring 3.0 accompanies all the features of Java 5. You can use the Spring 3.0 framework along with the other frameworks, such as JavaServer Faces (JSF) and Struts 2, to add additional functionalities in a Spring application. The following are the features of the Spring 3.0 framework:

- **Support for Java 5**—Provides annotation-based configuration support along with other features of Java 5, such as generics and varargs, which can be used in Spring 3.0 applications. You can use JDK 1.5 or above to run the Spring 3.0 applications.
- **Support for Spring Expression Language (SpEL)**—Allows you to use SpEL to provide XML and annotation-based bean definition.
- **Support for Representational State Transfer (REST) Web services**—Adds the functionality of REST Web services in Spring 3.0.
- **Support annotation-based formatting**—Allows you to use various annotations to automatically format or convert the bean fields to specified format. For example, the @DateTimeFormat(iso=ISO.DATE) is used to convert the date format.
- **Introduces various new packages in the JAR file**—Allows you to use various new packages, such as org.springframework.aop, org.springframework.beans, org.springframework.context, org.springframework.context.support, org.springframework.expression, org.springframework.instrument, org.springframework.jdbc, org.springframework.jms, org.springframework.orm, org.springframework.oxm, org.springframework.test, org.springframework.transaction, org.springframework.web, org.springframework.web.portlet, org.springframework.web.servlet, and org.springframework.web.struts, in a Spring application.

After having a brief overview about the new features of the Spring 3.0 framework, let's explore the Spring architecture.

Exploring the Spring Framework Architecture

The Spring framework consists of seven modules, which are shown in Figure 21.1. These modules are Spring Core, Spring AOP, Spring Web MVC, Spring DAO, Spring ORM, Spring context, and Spring Web flow. These modules provide different platforms to develop different enterprise applications; for example, you can use

Spring Web MVC module for developing MVC-based applications. Figure 21.1 shows the modules of the Spring framework architecture:

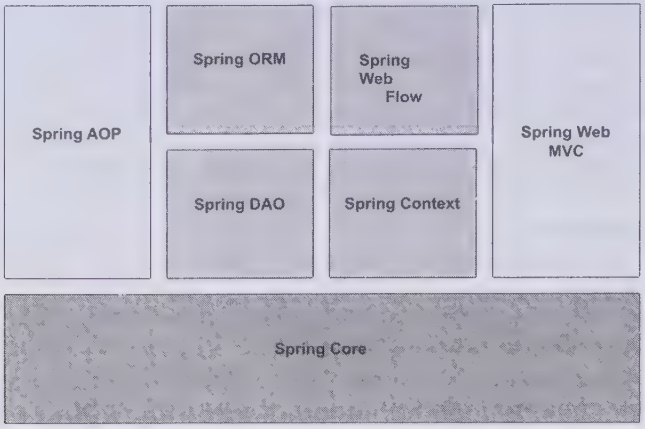


Figure 21.1: Displaying the Modules of the Spring Framework Architecture

Now, let's discuss each module shown in Figure 21.1 in the following subsection.

Explaining the Spring Core Module

The Spring Core module, which is the core component of the Spring framework, provides the IoC container. There are two types of implementations of the Spring container, namely, bean factory and application context. Bean factory is defined using the org.springframework.beans.factory.BeanFactory interface, and acts as a container for beans. The Bean factory container allows you to decouple the configuration and specification of dependencies from program logic.

In the Spring framework, the Bean factory acts as a central IoC container that is responsible for instantiating application objects. It also configures and assembles the dependencies between these objects. There are numerous implementations of the `BeanFactory` interface. The `XmlBeanFactory` class is the most common implementation of the `BeanFactory` interface. This allows you to express the object to compose your application and remove interdependencies between application objects.

Explaining the Spring AOP Module

Similar to Object-Oriented Programming (OOP), which breaks down the applications into hierarchy of objects, AOP breaks down the programs into `aspects` or `concerns`. Spring AOP module allows you to implement concerns or aspects in a Spring application. In Spring AOP, the aspects are the regular Spring beans or regular classes annotated with @Aspect annotation. These aspects help in transaction management, and logging and failure monitoring of an application. For example, transaction management is required in bank operations such as transferring an amount from one account to another. Spring AOP module provides a transaction management abstraction layer that can be applied to transaction APIs.

Explaining the Spring ORM Module

The Spring ORM module is used for accessing data from databases in an application. It provides APIs for manipulating databases with JDO, Hibernate, and iBatis. Spring ORM supports DAO, which provides a convenient way to build the following DAOs-based ORM solutions:

- Simple declarative transaction management
- Transparent exception handling
- Thread-safe, lightweight template classes
- DAO support classes
- Resource management

Explaining the Spring Web MVC Module

The Web MVC module of Spring implements the MVC architecture for creating Web applications. It separates the code of model and view components of a Web application. In Spring MVC, when a request is generated from the browser, it first goes to the DispatcherServlet class (Front Controller), which dispatches the request to a controller (SimpleFormController class or AbstractWizardFormController class) using a set of handler mappings. The controller extracts and processes the information embedded in a request and sends the result to the DispatcherServlet class in the form of model object. Finally, the DispatcherServlet class uses ViewResolver classes to send the results to a view, which displays these results to the users.

Explaining the Spring Web Flow Module

The Spring Web Flow module is an extension of the Spring Web MVC module. Spring Web MVC framework provides form controllers, such as SimpleFormController class and AbstractWizardFormController class, to implement predefined workflow. The Spring Web Flow helps in defining XML file or Java Class that manages the workflow between different pages of a Web application. The Spring Web Flow is distributed separately and can be downloaded through http:\\`www.springframework.org` website.

The following are the advantages of Spring Web Flow:

- The flow between different UIs of the application is clearly provided by defining Web flow in XML file
- Web flow definitions help you to virtually split an application in different modules and reuse these modules in multiple situations
- Spring Web Flow lifecycle can be managed automatically

Explaining the Spring DAO Module

The DAO package in the Spring framework provides DAO support by using data access technologies such as JDBC, Hibernate, or JDO. This module introduces a JDBC abstraction layer by eliminating the need of providing tedious JDBC coding. It also provides programmatic as well as declarative transaction management classes. Spring DAO package supports heterogeneous Java Database Connectivity and O/R mapping, which helps Spring work with several data access technologies

For easy and quick access to database resources, the Spring framework provides abstract DAO base classes. Multiple implementations are available for each data access technology supported by the Spring framework. For example, in JDBC, the `JdbcDaoSupport` class and its methods are used to access the `DataSource` instance and a preconfigured `JdbcTemplate` instance. You need to simply extend the `JdbcDaoSupport` class and provide a mapping to the actual `DataSource` instance in an application context configuration to access a DAO-based application.

Explaining the Spring Application Context Module

The Spring Application context module is based on the Core module. Application context `org.springframework.context.ApplicationContext` is an interface of `BeanFactory`. This module derives its feature from the `org.springframework.beans` package and also supports functionalities such as internationalization (I18N), validation, event propagation, and resource loading. The Application context implements MessageSource interface and provides the messaging functionality to an application.

In the next section, we discuss how DI and IoC are used in the Spring framework.

Exploring Dependency Injection and Inversion of Control

IoC is a principle of software construction, where the developers no longer need to create objects from classes. Instead, an application gets the objects from outside sources such as an XML configuration file. The concept of IoC is used to reduce coupling in an application. Coupling is a term that describes the degree to which one module depends on another module for proper functioning of an application. Loose coupling allows easier maintainability and higher reusability of an application. In this way, IoC facilitates better software design. Spring IoC container is the core of the Spring framework. The two interfaces, such as ApplicationContext and BeanFactory, allow you to manage beans in the Spring IoC container.

DI is a programming design pattern, which is used to reduce coupling in an application. DI allows the developers to inject (use) an object directly in a class, which means that they no longer need to depend on a class to create an object. Spring framework supports loose coupling between one module and other module of an application by using DI. The basic principle of DI is that objects define their dependencies (other objects they work with) using constructor argument or argument to a factory method. It means that objects should only have as many dependencies as are required to perform their job. In other words, we can say that the number of dependencies should be minimum in an application.

The key benefit of injecting objects in an application is that you can modify implementation part of dependencies without being concerned about the depending object. For example, if the Car class knows about its dependency on the Petrol class through an interface, any other effect of changes in the implementation of the Petrol class to the Car class is not required. Even if we modify the implementation of the Petrol class, it does not affect the Car class.

Let's now discuss both these concepts in detail.

Explaining DI

As already learned, DI is a concept of injecting an object into a class rather than explicitly creating the object in a class, since the IoC container injects object into the class during runtime. The three important types of DIs are setter injection, constructor injection, and method injection. Setter injection is implemented through JavaBeans setter methods. The Constructor Injection is implemented through Constructor arguments, where the IoC container is responsible for implementing methods at runtime. The method injection is implemented through a number of callback methods and an API for traditional lookup.

The benefits of DI are as follows:

- Ensures that components are simpler to write and maintain, as they do not require runtime collaboration lookup
- Reduces coding effort, as JavaBeans can be injected in a class using DI
- Allows business objects to run in different DI frameworks or outside any framework without any change in the code.

Explaining IoC Container

The actual representation of the Spring IOC container is the `BeanFactory` interface. This interface is an implementation of the factory design pattern, which creates and destroys beans. Various implementations of the BeanFactory interface exist; the `XmlBeanFactory` class being the most commonly used BeanFactory implementation. This `XmlBeanFactory` class loads its beans based on the definitions specified in an XML file, and takes this XML configuration metadata to create a fully configured system or application.

The two models of objects supported by the `BeanFactory` interface are as follows:

- **Singleton**—Serves as the default and most commonly used model in which the only shared instance of the object is retrieved on lookup with a particular name. This object model is ideal for stateless service objects.
- **Prototype**—Refers to the non-singleton model. Every new retrieval of a bean results in the creation of a new object, and in this way, it is used to allow each caller to have its own distinct object references.

Two important methods in the `BeanFactory` interface are getBean(String name) and getBean(String name, Class requiredType) methods. The getBean(String name) method allows you to get a bean on the basis of the value of the name parameter. The getBean(String name, Class requiredType) method allows you to specify the required class of the returned bean and throws an exception if it doesn't exist.

Sub-interfaces of BeanFactory provide extra functionality, such as message lookup, and configuring metadata of a bean. The following are the sub-interfaces of the BeanFactory interface:

- **ListableBeanFactory**—Provides methods to enumerate the beans in a bean factory. For example, the getBeanDefinitionNames() method of the ListableBeanInterface interface returns the names of all the beans defined in the factory, the getBeanNamesForType(Class type) method returns the names of all the bean classes of a certain type, and the getBeanDefinitionCount() method returns the number of bean classes defined in the factory.

- **AutowireCapableBeanFactory**—Helps in configuring an existing external object and facilitates its dependencies. This is done through the autowireBeanProperties() and applyBeanPropertyValues() methods. By using the specified autowire strategy, the autowire() method creates a new bean instance of the given class.
- **ConfigurableBeanFactory**—Provides additional configuration options on a bean factory that can be applied during the bean initialization stage.
- **ApplicationContext**—Provides extra functionality of Bean Factory. This interface provides support for message lookup, internationalization, and an event handling mechanism.

One of the most striking features of Spring is AOP, which will be discussed in the next section.

Exploring AOP with Spring

AOP is a complement of OOP, which is defined as a programming technique. The important unit of modularity in AOP is aspect; whereas in OOP, the units of modularity are classes and objects. Aspects provide the modularization of various concerns, such as transaction management, which crosscut multiple types and objects. In AOP, such concerns are termed as cross-cutting concerns. For example, security is a cross-cutting concern in an application, as EmployeeService, SalaryService, and other services of that application must implement the security functionality.

The Spring IoC containers don't depend on AOP; however, you can use these containers with AOP, depending on the requirements of the application. The following are the uses of AOP in Spring:

- Replaces the EJB declarative services with new declarative enterprise services such as declarative transaction management.
- Allows you to implement custom aspects by using OOP with AOP.
- Allows AOP to be combined with Acegi security framework to provide declarative security service. Acegi security framework is a kind of Spring security framework that provides security solutions for Java EE-based Web applications.

Describing the AOP Concepts

The following list defines and explains the various key terms of the AOP module:

- **Advice**— Defines both what and when of an aspect. We know that each aspect has a purpose in AOP; the purpose of an aspect is known as advice. In real world, an advice defines what the job is and when it can be performed. The advice can be applied either before or after method invocation by using the `@Aspect` annotation.
- **Joinpoint**—Refers to a point where an aspect can be plugged in. It is the single location in the code where an advice should be executed (i.e., method invocation). For specifying a joinpoint, you have to implement the `org.springframework.aop.Pointcut` interface, and the information about the joinpoint can be accessed by the MethodInvocation interface.
- **Pointcut**—Refers to many joinpoints that are grouped together to perform an advice, which makes a pointcut. As we know that in Spring, a joinpoint is always a method invocation; therefore, we can say that a pointcut is a set of methods invocation.
- **Aspect**—Combines both advice and pointcuts. Therefore, an aspect consists of information such as what is the role of the aspect, and where and when to apply the aspect.
- **Introduction**—Adds methods and fields to an existing object. For using this concept, Spring has introduced some interfaces that must be implemented by the advised class.
- **Target object** —Refers to an object that is being adviced by one or more aspects.
- **AOP proxy**—Refers to an object created by the Spring AOP to implement the aspect. In Spring, there are two types of AOP proxies–JDK dynamic proxy and code generator library (CGLIB) proxy.
- **Weaving**—Refers to the process that links aspects with other application types or objects to create an advised object. This process can be performed at compile time, load time, and runtime.