are performed. This means that deferred expressions are `lvalue` expressions, that is, expressions that can read or write data on a JavaBean. The `#{employee.name}` expression acts as an `rvalue` expression, which only reads data during the initial request and acts as the `lvalue` expression during postback.

Value Expressions

Value expressions are used to refer to objects such as `JavaBeans`, collections, enumerations, and implicit objects, and their properties. An object is referred to by using the value expression containing the name of the object. Suppose the `${employee}` expression is used in a JSP page, where `employee` refers to the name of a JavaBean. When the Web container comes across the `${employee}` expression, it invokes the `PageContext.findAttribue (String)` method internally, which searches for the employee JavaBean in the request, session, and application scopes. If the employee JavaBean does not exist, a null value is returned.

To refer to an enum constant by using value expression, a String literal is used. The following code snippet shows the syntax for an enumeration declaration:

```
public enum PlayCardSuite {hearts, spades, diamonds, clubs}
```

The following code snippet shows the syntax to access a constant defined by the `PlayCardSuite` enumeration by using EL:

```
${PlayCardSuite.spades}
```

The expression shown in the preceding code snippet is an example of a value expression, which is used to refer to the eum constant of the `PlayCardSuite` enumeration.

Now let's discuss the syntax of value expressions to access the properties and elements of a collection in a JavaBean. For this, we use the (.) and ([ ]) operators. For example, the `name` property of the `employee` JavaBean can be referred to by using either the `${employee.name}` or `${employee["name"]}` expression. Double or single quotes can be used to represent a String literal. To access a specific element of a collection, such as a list or an array inside a JavaBean, the [ ] notation having an int value is used. For example, `${employee.phonenos[1]}` accesses the second phone number of the `phonenos` array.

Value expressions are of two types, `rvalue` and `lvalue`. An `rvalue` expression can only read data, and not write data. On the other hand, the `lvalue` expression can read as well as write data. Examples of `rvalue` expressions are `${"hello"}`, `${employee.age+20}`, and `${false}`. EL supports all types of literals in Java, such as Boolean, integer, floating point, and String.

Value expressions may be embedded in static text or as a value of a tag attribute that can take an expression. The following code snippet shows how to embed a value expression in static text:

```
<prefix:tag>
  some text ${expr} some text
</prefix:tag>
```

In the preceding code snippet, the `${expr}` expression is embedded within a text String. If the body of the <prefix:tag> tag is dependent on another tag, then the embedded expression is not evaluated.

Method Expressions

Method expressions are used to call public methods, which return a value or object. Such expressions are usually deferred expressions. JSF HTML tags represent UI components on a JSF page. These tags use method expressions to call functions that perform operations such as validating a UI component or handling the events generated on a UI component. The following code snippet shows the use of method expressions in a JSF page:

```
<h:form>
  <h:inputText
        id="email"
        value="#{employee.email}"
        validator="#{employee.validateEmail}"/>
  <h:commandButton
  id="submit"
  action="#{customer.submit}" />
</h:form>
```

The various elements shown in the preceding code snippet can be briefly described as follows:

- The `inputText` tag—Shows the `UIInput` component in the form of a text field on a Web page

- The `validator` attribute—Calls the `validateEmail` method of the employee JavaBean
- The `action` attribute of the `commandButton` tag—Calls the `submit` method, which carries out processing after submitting the Web page
- The `validateEmail` method—Refers to the method that is called during the validation phase of the JSF life cycle
- The `submit` method—Refers to the method that is called during the invoke application phase of the JSP life cycle

The TLD (Tag Library Descriptor) file is used to define the signatures of methods referenced by the attributes used in the preceding code snippet.

You are now familiar with the classification of EL expressions. Let's now discuss tag attribute types, such as static, dynamic, and deferred-method.

Describing Tag Attribute Types

Resolving EL expressions depends on the type of tag attribute defined in TLD. For custom tags, you need to specify the type of expression accepted by the custom tag. There are four categories of attribute types that define how an EL expression is evaluated. These attribute type categories are as follows:

- **Static attribute**—Refers to the attribute where the value of the `rtexprvalue` element is assigned as `false` and can be of any type, such as `int`, `String`, and `com.Example.Order`. A static attribute produces a translation error if an expression is passed to the attribute.
- **Dynamic attribute**—Refers to the attributes where the `rtexprvalue` element is assigned the `true` value and can be of any type, such as `int`, `String`, and `com.Example.Order`. Whenever an expression is provided to a dynamic attribute that is not deferred-value or deferred-method, the expression is parsed with the type similar to the attribute type, and the expression is resolved immediately.
- **Deferred-value attribute**—Refers to the attribute of the `javax.el.ValueExpression` type.
- **Deferred-method attribute**—Refers to the attribute of the `javax.el.MethodExpression` type. If an attribute is declared as deferred-value or deferred-method in TLD, the attribute must be of type `ValueExpression` or `MethodExpression`. The JSP container parses the expression but does not evaluate it. The result of parsing the expression is passed to the Tag handler attribute.

Resolving EL Expressions

Resolving an EL expression involves searching of the components of the EL expression, which are separated by the dot (.) operator. EL expressions are resolved from left to right. The unified EL API includes some classes and implementations to resolve EL expressions. Table 7.2 lists commonly used classes of the unified EL API:

Table 7.2: Commonly Used Classes of the EL API

Class	Description
ValueExpression	Represents a value expression.
MethodExpression	Represents a method expression.
ELResolver	Defines object resolution rules to resolve EL expressions.
ELContext	Stores the state of an EL resolution. The object of the ELContext class provides access to other objects, such as `JspContext`, to resolve EL expressions.

Several implementations of the `ELResolver` class are available to resolve an EL expression referring to a specific object o its property. The classes shown in Table 7.2 are used to build custom EL resolvers.

If the EL expression is a value expression, it can be resolved as follows:

1. A value expression is parsed and a `ValueExpression` object is generated, when a JSP page containing the EL value expression is requested for the first time.
2. The `getValue()` method of the `ValueExpression` object is called.

3. The `getValue()` method also calls the `getValue()` method of the suitable resolver.
4. The `setvalue()` method is called if the expression is of `lvalue` type.

If the EL expression is a method expression, it can be resolved as follows:

1. The `BeanELResolver` object is created, which searches the object that has the definition of the method referred in this expression.
2. The `MethodExpression` object is created to represent the method expression.
3. The `getMethodInfo()` method is invoked, which calls the `getValue()` method of the `BeanELResolver` object that resolves the expression.

After resolving an EL expression, if the `propertyResolved` flag of the `ELContext` object is set to true, the resolver is detached from the EL expression.

Many standard implementations of EL resolver classes exist in the EL API. For example, to resolve the `${employee.name}` expression, a standard `BeanELResolver` object is used, which first resolves or searches for the `employee` JavaBean and then resolves a specific property name. Table 7.3 lists all the standard EL resolver classes:

Table 7.3: Standard EL Resolver Classes

Class	Description
ArrayELResolver	Performs property resolution on arrays
BeanELResolver	Performs property resolution on JavaBeans
ListELResolver	Performs property resolution on objects of the List class
MapELResolver	Performs property resolution on objects of the Map class
ResourceBundleELResolver	Performs property resolution on the objects of the ResourceBundle class

The following are few EL expressions with their descriptions:

- **${arr[2]}**—Returns the value of the element at position 2 of an array named arr
- **${employee.name}**—Returns the value of the name property of the employee JavaBean
- **${list1[4]}**—Returns the value of fourth element of the list1 list
- **${map1.key1}**—Returns the value of the key1 key in the map1 map
- **${RB1.key1}**—Returns the message corresponding to the key1 key in the resource bundle named RB1

If you do not have an EL resolver implementation to handle your EL expression, you need to create a custom EL resolver and register it with the corresponding JSP application. A `CompositeELResolver` instance holds references to all standard and custom resolvers. This instance is iterated to check whether or not the current resolver can resolve the specified EL expression.

JSP 2.1 provides two more resolvers, which refer to implicit objects. These are `ImplicitObjectResolver` and `ScopedAttributeResolver`. Consider the expression `${sessionScope.map}`. When a JSP 2.1 container encounters this expression, it first resolves the implicit object `sessionScope` and then uses `MapELResolver` to resolve the map attribute as map represents an instance of the Map class. The `ScopedAttributeResolver` object resolves an object stored in one of the scopes, such as page, request, session, and application. For example, in the case of the `${employee}` expression, the `ScopedAttributeResolver` object searches for the employee JavaBean in all the scopes. If the value is not found, the object returns a null value.

NOTE

Various J2EE expert groups are trying to incorporate EL into the JSP specification.

In JSP 2.1, all classes and interfaces that belonged to the javax.Servlet.jsp.el package have been deprecated and introduced in the new unified EL APIs (javax.el).

Describing EL Operators

EL operators are introduced in the unified EL API so that Java programmers can perform arithmetic calculations. Now, you can create EL expressions that perform arithmetic operations by using the operators of EL.

Let's now discuss the various types of EL operators.

Exploring the Types of EL Operators

EL provides support for any type of arithmetic, relational, or logical operations. EL operators can be categorized as follows:

- Arithmetic operators
- Relational and logical operators
- The empty operator

Let's discuss these operators in detail.

Arithmetic Operators

All types of arithmetic operators (+,-, /,*, or %) are used in EL expressions. The % operator is used to divide the two integer values and return their reminder. Div and mod are alternative names of the division (/) and modulus (%) operators, respectively. The minus (-) operator is used as a unary operator to work with negative numbers. EL also supports floating point numbers, which are fractional numbers.

Relational and Logical Operators

EL provides a set of relational and logical operators that are used in many programming languages. All relational and logical operators can be denoted by using symbols or short text forms. Table 7.4 lists the relational operators used in EL:

Table 7.4: Relational Operators in EL

Operator Symbol	Short Text Form	Description
>	gt	Greater than
>=	ge	Greater than or equal to
<	lt	Less than
<=	le	Less than or equal to
==	eq	Equal to
!=	ne	Not equal to

In case any operand of the operators is a character, EL changes the character data into a numerical value.

Parentheses can override precedence of most but not all operators; Examples of operators whose precedence is not affected with the use of parenthesis are [ ] and ;. Table 7.5 shows a list of EL operators with their precedence in decreasing order:

Table 7.5: Order of Precedence of EL Operators

Order	Operator
1	[ ] . ;
2	( )
3	Unary operators such as -, !, and empty
4	* / %
5	+ -
6	< > <= >=

Table 7.5: Order of Precedence of EL Operators	
Order	**Operator**
7	== !=
8	&&
9	||
10	?:

The Empty Operator

The empty operator is a prefix operator, which means it takes one operand on its right side. This operator is used to check for empty values, which differ according to the data type of the operand. Table 7.6 lists the empty values for different data types:

Table 7.6: Empty Values for Different Data Types	
Data Type	**Empty Value**
String	""
Identifier	Null
Array	No elements
Map	No elements
List	No elements

Let's now learn to use EL operators.

Using EL Operators

Now, let's put all the EL operators in use by creating the Operators application. Listing 7.15 demonstrates several EL expressions using arithmetic, relational, logical, and empty operators (you can also find the operators.jsp file on the CD in the code\Chapter7\Operators folder):

Listing 7.15: Showing the Code for the operators.jsp File

```
<html>
  <head>
        <title>Using EL Operators</title>
  </head>
  <body>
        <h2>Using EL Operators</h2>
        <h3>Arithmetic Expressions</h3>
        <b>You Score in IELTS is ${1 + 2 * 4 - 6 / 2}.</b><br/>
        <b>Today temperature of Delhi is ${-4 - 8} degree celsius.</b><br/>
        <b>Is 3/4 equals to 0.75 ? ${(3/4 == 0.75)? "Yes" : "No"}</b><br/>
        <b>Mathematical Constant PI has value ${22/7} </b><br/>
        <b>GDP of India is  ${20 div 2}.</b><br/>
        <b>I have ${2003 mod 8} mobiles.</b><br/>
        <b>I have ${2003 % 8} mobiles.</b><br/>

        <h3>Logical Operators </h3>
        <b>This Logical and EL expression \${(1&lt;2) && (4&lt;3)} is ${(1<2) &&
        (4<3)}</b><br/>
        <b>This Logical or EL expression \${(1&lt;2) || (4&lt;3)} is
        ${(1<2) || (4<3)}</b><br/>
        <b>This Logical not EL expression \${!(1&lt;2)} is ${!(1<2)} </b><br/>

        <h3>Comparison Operators </h3>
        <b>Is 4 > 3 ?  ${4 > 3}</b><br/>
        <b>Is "a" > "b"? ${"a" > "b"}</b><br/>
        <b>Is 4 >= 3 ${4 >= 3}</b><br/>
```

```
            <b>Is 4 <= 3 ${4 < 3}</b><br/>
            <b>Is 4 == 4? ${4 == 4}</b><br/>
            <h3>empty Operator</h3>
            <b>empty "" ${empty ""}</b><br/>
            <b>empty "string" ${empty "string"}</b><br/>
            <b>empty null ${empty null}</b>
    </body>
</html>
```

Listing 7.15 shows you the use of the addition, subtraction, division, and modulus operators in EL expressions, under the heading `Arithmetic Expressions`. The conditional operator (?:) is also used in the line `<b> Is 3/4 equals to 0.75? ${(3/4 == 0.75)? "Yes": "No"} </b><br/>`. Under the second heading, namely `Logical Operators`, you learn how to perform some logical expressions, by using the &&, | |, and ! operators. If you want to print an EL expression as it is, you need to prefix it with backslash (/). Apart from this, you also learn to perform comparisons by using the <, <=, >, >=, and == relational operators, under the heading `Comparison Operators`. An empty prefix operator is also used on a blank String and null value.

Package the Operators application into the Operators.war file and then, deploy the `Operators.war` file on the Glassfish V3 application server. Now, open the Internet Explorer browser and type the URL `http://localhost:8080/Operators/operators.jsp` to view the output of the `operators JSP` page, as shown in Figure 7.19:

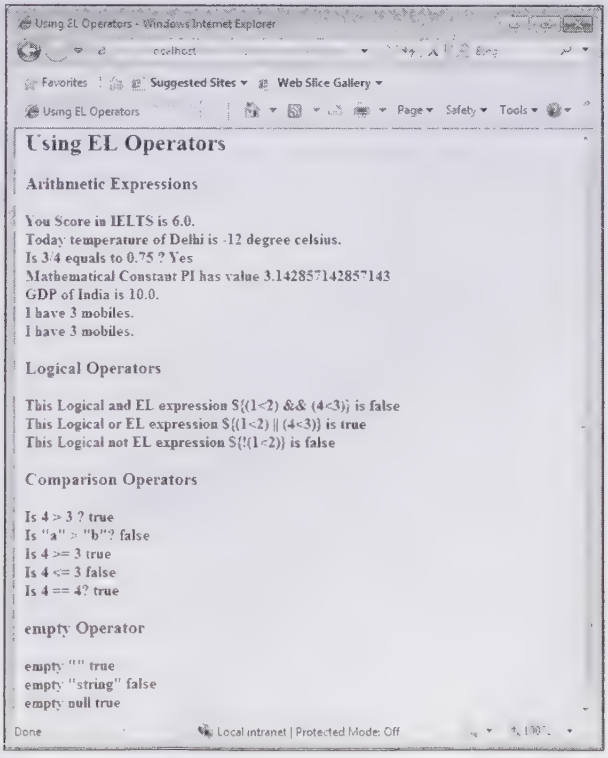


Figure 7.19: Displaying the Output of operators JSP Page

In Figure 7.19, all arithmetic numbers under the `Arithmetic Expressions` heading are displayed by using arithmetic EL expressions and all false and true results are displayed by using relational and logical operators. You have to scroll down to see the results after using the empty operator on various types of Strings.

Describing EL Objects

JSP programmers can directly use implicit objects in an EL expression. They do not need to write extra code to use these objects. Some of these objects allow access to variables in particular JSP scopes. The implicit scope

objects used in EL are `pageScope`, `requestScope`, `sessionScope`, and `applicationScope`. All these scope objects map the scope attribute names to their values, respectively.

You can also access HTTP request parameters by using the implicit objects `param` and `paramValues`. Request header information can be retrieved by using the `header` and `headerValues` implicit objects. The `param` and `header` objects map the parameter and header names to a `String`. The `paramValues` and `headerValues` objects map the parameter and header names of all the values for that parameter or header to a `String[]` array. In the earlier versions of Java Servlet, the `ServletRequest.getParameter(String name)` and `ServletRequest.getHeader(String name)` methods were used to map the parameter and header names to a String. Similarly, the `ServletRequest.getParameterValues(String name)` and `HttpServletRequest.getHeaders(String)` methods were used to map parameter and header names of all values for that parameter or header to a `String[]` array.

The role of `initParam` is to provide access to context initialization parameters. The implicit object `pageContext` provides access to all properties of a context of a JSP page, such as the `ServletContext` and `HttpSession` objects and their properties.

Table 7.7 provides a brief description of the implicit objects in EL:

Table 7.7: Implicit Objects in EL

Object Name	Description
pageContext	Represents an instance of the `pageContext` object and is used to manipulate page attributes and access the `servletContext`, `session`, `request`, and `response` objects.
pageScope	Maps page-scoped attribute names to their values.
requestScope	Maps request-scoped attribute names to their values.
sessionScope	Maps session-scoped attribute names to their values.
applicationScope	Maps application-scoped attribute names to their values.
param	Maps parameter names to a single String parameter value, obtained by calling the ServletRequest.getParameter(String name) method.
paramValues	Maps parameter names to a String[] array of all the values for that parameter, obtained by calling the ServletRequest.getParameterValues(String name) method.
Header	Maps header names to a single String header value, obtained by calling the ServletRequest.getHeader(String name) method.
headerValues	Maps header names to a String[] array of all the values for that header, obtained by calling the HttpServletRequest.getHeaders(String name) method.
cookie	Maps cookie names to a single Cookie object. Cookies are retrieved according to the semantics of the HttpServletRequest.getCookies() method.
initParam	Maps context initialization parameter names to their String parameter value that is obtained by calling the ServletContext.getInitParameter (String name) method. Context initialization parameters are usually set in the web.xml file.

Now, let's learn the use of implicit scope objects, such as `requestScope`, `sessionScope`, and `applicationScope`.

Showing the Use of Implicit EL Scope Objects

The example shown in this section illustrates how to access scoped variables in a JSP EL. The process of accessing scoped variables starts from the servlet, which is configured in the web.xml file to handle a request. The servlet performs the business logic on the data sent with the request and sets the output data in different scopes, such as request, session, or application. The servlet sets the output data in the request, session or application scope by invoking the `setAttribute()` method on the `HttpServletRequest`, `HttpSession`, or `ServletContext` object, respectively and then forwards the control to the JSP page to display the output data

by using the `RequestDispatcher.forward()` or `HttpServletResponse.sendRedirect()` method. There is another scope, called PageContext, which stores page scoped variables or objects used on a JSP or servlet page. Therefore, this scope is not shared by servlets and JSPs.

Now, let's see how scoped variables are accessed in EL. To access and display a scoped variable on a browser, EL requires expressions such as ${name}, where name is a scoped variable. When an EL supporting container encounters this expression, it searches for the name variable in the specified order: `PageContext`, `HttpServletRequest`, `HttpSession`, and `ServletContext`. If it gets the name variable, it calls the `toString()` method on the variable value and provides the output.

If you use the `<%=pageContext.findAttribute("name")%>` syntax to access the name variable, the search is only limited to the page scope. However, to search a resource in all scopes, you can use the following syntax of the `<jsp:useBean>` tag:

```
<jsp:useBean id="name" type="package.class" scope=" ">
```

To use the preceding syntax, you should have information about the scope of the servlet and the qualified path of the JavaBean class.

Let's create the `ImplicitObjects` Web application containing a servlet, which sets attributes or variables in different scopes, and a JSP page, which accesses these variables by using the EL syntax. Listing 7.16 shows the code for the `SetscopedVariables.java` file, which sets the value of the `attribute1` attribute (you can also find this file on the CD in the `code\Chapter7\ImplicitObjects\src\com\kogent\` folder):

Listing 7.16: Showing the Code for the SetScopeVariables.java File

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SetScopeVariables extends HttpServlet
{
  public void doGet(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException
  {
        req.setAttribute("attribute1", "First Value");
        HttpSession session = req.getSession();
        session.setAttribute("attribute2", "Second Value");
        ServletContext application = getServletContext();
        application.setAttribute("attribute3",new java.util.Date());
        req.setAttribute("repeated", "Request");
        session.setAttribute("repeated", "Session");
        application.setAttribute("repeated", "ServletContext");
        RequestDispatcher dispatcher =
        req.getRequestDispatcher("GetScopeVariables.jsp");
        dispatcher.forward(req, res);
  }
}
```

In Listing 7.16, we set the attribute1, attribute2, and attribute3 attributes to First Value, Second Value, and current date and time Strings, respectively. The attribute1, attribute2, and attribute3 attributes are set in the request, session, and application scope, respectively. There is another attribute, called repeated, which is set in all the three scopes. Finally, the servlet transfers the control to the GetScopedVariables JSP page.

In Listing 7.17, the values of the attribute1, attribute2, attribute3 attributes are accessed, irrespective of the scopes where these attributes are stored. The code to access these attributes is provided in Listing 7.17. Apart from this, Listing 7.17 also contains some code statements that restrict the search of a particular attribute in a particular scope, such as sessionScope.attribute1. Listing 7.17 shows the use of some implicit objects (you can find the GetScopeVariables.jsp file on the CD in the code\Chapter7\ImplicitObjects folder):

Listing 7.17: Showing the Code for the GetScopeVariables.jsp File

```
<HTML>
<HEAD>
  <TITLE>Accessing Scoped Variables</TITLE>
```

```
</HEAD>
<BODY>
  <TABLE BORDER=2 ALIGN="CENTER">
  <TR><TH>
  Accessing Scoped Variables
  </TABLE>
  <P>
  <UL>
  <LI><B>Accessing attribute1 directly:</B> ${attribute1}
  <LI><B>Accessing attribute1 using requestScope implicit object:</B>
  ${requestScope.attribute1}
  <LI><B>Checking whether attribute1 is present in session scope using
  sessionScope implict object:</B>${sessionScope.attribute1}

  <LI><B>Accessing attribute2 directly:</B> ${attribute2}
  <LI><B>Checking whether attribute2 is present in application scope using
  applicationScope implict object:</B> ${applicationScope.attribute2}

  <LI><B>Accessing attribute3 directly:</B> ${attribute3}

  <LI><B>Accessing repeated attribute directly:</B> ${repeated}
  <LI><B>Checking repeated attribute only in application scope using
  applicationScope implicit object:</B> ${applicationScope.repeated}
  </UL>
</BODY>
</HTML>
```

Deploy the `ImplicitObjects` application on Glassfish V3 application server. To execute the example, open Internet Explorer and type the URL http://localhost:8080/ImplicitObjects/SetScopeVariables. Figure 7.20 shows the output of executing the SetScopeVariables servlet:

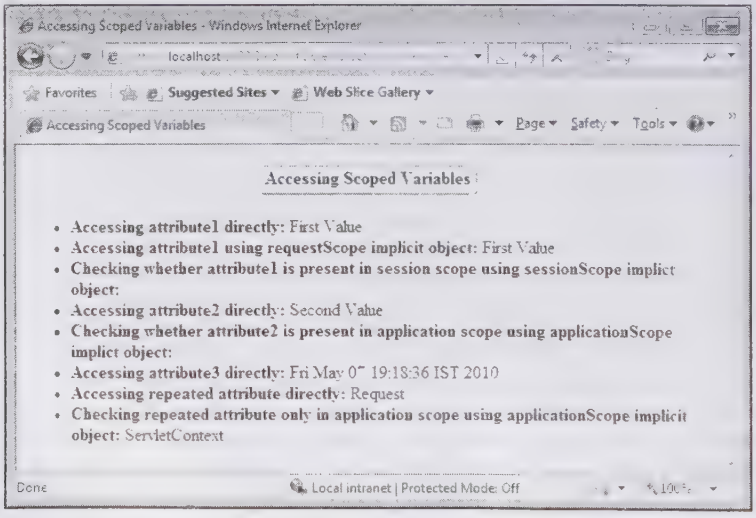


Figure 7.20: Accessing Attributes by using Scoped Variables

In Figure 7.20, you can see that no value is displayed in the third and fifth lines because the JSP container is not able to find attribute1 and attribute2, respectively in the application scope. This is another advantage of EL: it does not generate a NullPointerException exception or return null value if it is unable to find an attribute in a particular scope. It only displays an empty String as the output.

After learning how to use implicit scope EL objects, let's now discuss the implementation of the paramValues and pageContext implicit EL objects.

Showing the Use of Implicit EL Objects

In the example shown in this section, a user is asked to enter some information and then retrieve and display the information along with other information, such as the server name, JSESSIONID.

The index HTML page is a simple HTML form that contains the email text box, three check boxes, and a Submit button. When a user submits this form, the control is transferred to the impobjects2 JSP page. Listing 7.18 shows the code for the index.html file (you can also find this file on the CD in the code\Chapter7\ImplicitObjects\ folder):

Listing 7.18: Showing the Code for the index.html File

```
<html>
  <head>
        <title>Index Page</title>
  </head>
  <body>
        <form action="impobjects2.jsp" method="get">
        <table BORDER=2 ALIGN="CENTER">
        <TR><Th>
                Using Implicit Objects
        </th></TR>
        </table>
        <table>
                <tr><td colspan=2><h3>Pet Shopping Cart</h3></td></tr>
                <tr><td>
                        Your Email Id :<input type="text"
                        name=email size="30"/>
                </td></tr>
                <tr><td>
                        What type of pets do you have?
                </td></tr>
                <tr><td>
                        cat <input type="checkbox" name="pettype"
                        value="cat" />
                </td></tr>
                <tr><td>
                        dog <input type="checkbox" name="pettype"
                        value="dog" />
                </td></tr>
                <tr><td>
                        rabbit <input type="checkbox"
                        name="pettype" value="rabbit"/>
                </td></tr>
        </table>
        <table>
                <tr><td>
                        <input type=submit value="Submit">
                </td></tr>
        </table>
        </form>
  </body>
</html>
```

In Listing 7.19, the impobjects2 JSP page displays the e-mail id of the user by using the param implicit object. The <c:forEach> tag of the JSTL core library is used in the impobjects2 JSP page. The <c:forEach> tag iterates over the collection represented by the paramValues implicit object, assigns each value to the pet variable, and prints the values stored in the pet variable. You need to map the index.html file as the welcome page in the web.xml file. Listing 7.19 shows the code for the impobjects2.jsp file (you can also find this file on the CD in the code\Chapter7\ImplicitObjects\ folder):

Listing 7.19: Showing the Code for the impobject2.jsp File

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<HTML>
  <HEAD>
        <TITLE>Using Implicit Objects</TITLE>
  </HEAD>
  <BODY>
        <TABLE BORDER=2 ALIGN="CENTER">
        <TR><TH>
              Using Implicit Objects
        </TH></TR>
        </TABLE>
        <P><UL>
              <LI><B>Your Email ID is :</B> ${param.email}
              <Li><B>You have selected pets:</B>
                    <c:forEach var="pet" items="${paramValues.pettype}">
                      ${pet}  
                    </c:forEach>
              <LI><B>User-Agent Header:</B> ${header["User-Agent"]}
              <LI><B>JSESSIONID Cookie Value:</B>
                    ${cookie.JSESSIONID.value}
              <LI><B>Server:</B> ${pageContext.servletContext.serverInfo}
        </UL>
  </BODY>
</HTML>
```

In Listing 7.19, the JSTL core library is included to use its `<c:forEach>` tag. The `{header["User-Agent"]}` expression displays the browser associated with the user agent header. The implicit object `cookie` is used to retrieve the value of `JSESSIONID`, which is a large String consisting of alphabets and numbers. Finally, the `pageContext` object is used to fetch the server name, which processes the `index HTML` page and displays the results.

Store the `index.html` and `implobject2.jsp` files in the `ImplicitObjects` directory according to the standard directory structure of Web applications. To use the JSTL core library, you must place the `jstl.jar` file in the `lib` folder under the `WEB-INF` folder of the `ImplicitObjects` directory. Now, redeploy the `ImplicitObjects` application on the Glassfish V3 application server. Next, open the Internet Explorer browser and type the URL `http://localhost:8080/ImplicitObjects/`. The output of the `index HTML` page is displayed, as shown in Figure 7.21:

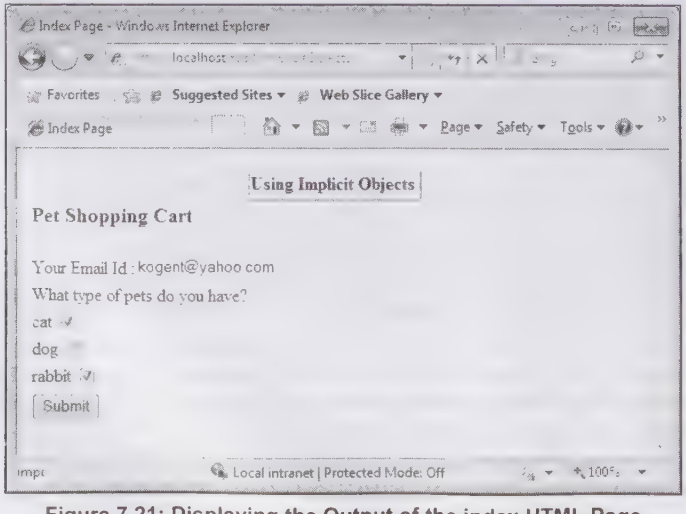


Figure 7.21: Displaying the Output of the index HTML Page

In Figure 7.21, the user enters `kogent@yahoo.com` as the e-mail id and then selects the check boxes corresponding to `cat` and `rabbit`. After clicking the `Submit` button, the request is redirected to the impobject2 JSP page and the submitted details are displayed, as shown in Figure 7.22:

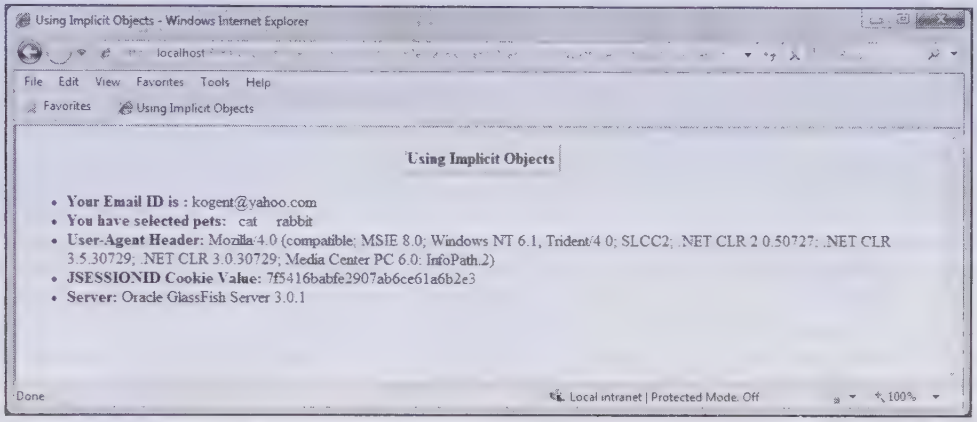


Figure 7.22: Displaying the User Details by using Implicit Objects

Figure 7.22 displays the e-mail id and the selected check boxes values, the compatible browser type, `JSESSIONID`, and the name of the server with version, in the form of a bulleted list.

You are now familiar with using expressions in EL. Now, let's explore how to use functions with EL.

Using Functions with EL

Sometimes, Java programmers may need to change the format of data or manipulate the data before it is displayed on a Web page. A custom tag library can be used for this purpose; however, EL provides no such built-in library. Therefore, Java programmers need to define their own methods. The prefix assigned to a function tag library is used to access EL functions. In other words, EL has the concept of qualified functions. EL functions are public static methods in Java classes, which are mapped in TLD.

As far as a tag library is concerned, each tag library may include zero or more static functions that are listed in the TLD file. Moreover, the name given to a function in the TLD file is exposed to EL. A public static method in the specified public class implements the functions. You have to be very careful about naming a function in a tag library, since the name of the function must be unique in the library. If the function is not declared correctly, or if there are multiple functions bearing the same name, an error is generated during translation time.

We have already covered method expressions, which appear similar but are not identical to EL functions. The differences between function expressions and method expressions are as follows:

- Functions are static methods that return single values but method expressions use non-static public methods.
- Functions are recognized during translation time but methods are recognized dynamically at runtime.
- The identification of a particular method is provided in a method expression and that method is invoked in the tag attribute definition by using the method expression. On the other hand, in functions, function parameters and their invocations are a part of EL expressions.

This section explains the use of EL functions with a Web application, `ELFunctions`. This application uses two EL functions. The first EL function accepts a signed integer value and returns its absolute value, while the second EL function accepts a floating point value and rounds it into an integer value. The `functions JSP` page includes a custom tag library represented by the URI `http://www.kogentindia.com/el-functions-taglib`, and invokes the `abs` and `round` functions by using the XML namespaces notation, such as `f:abs( )`. Listing 7.20 invokes two functions within EL expressions (you can find the functions.jsp file on the CD in the `code\Chapter7\ELFunctions` folder):

Listing 7.20: Showing the Code for the functions.jsp File

```
<%@ taglib prefix="f" uri="http://www.kogentindia.com/el-functions-taglib" %>
<html>
<head>
  <title>Using EL Functions</title>
</head>
<body>
  <h1>Using EL Functions </h1>
  The absolute value of \${num} is ${f:abs(-300)}.<br/>
  The rounded value of \${calc} is ${f:round(6.666666666666667)}.<br/>
</body>
</html>
```

A JSP container locates the imported tag library in the `web.xml` file of the `ELFunctions` application. The `web.xml` file consists of the `<taglib>` tag having two nested tags, `<taglib-uri>` and `<taglib-location>`. The `<taglib-uri>` tag takes the specified URI in the `uri` attribute of the `taglib` directive in the `functions JSP` page. The actual location of the TLD file is specified in the `<taglib-location>` tag, which is `/WEB-INF/function-taglib.tld`. Listing 7.21 shows code for the `web.xml` file (you can also find this file on the CD in the code\Chapter7\ ELFunctions\WEB-INF folder):

Listing 7.21: 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">

  <taglib>
        <taglib-uri>
              http://www.kogentindia.com/el-functions-taglib
        </taglib-uri>
        <taglib-location>
              /WEB-INF/function-taglib.tld
        </taglib-location>
  </taglib>

</web-app>
```

EL function definitions are contained within another TLD file. The `function-taglib.tld` file maps the function calls in the `functions.jsp` file to the `<function>` tags. The Java class implementing the function, which in this case is `java.lang.Math` for both the `abs` and `round` functions, is specified in the `<function-class>` tag. These functions signatures are specified in the `<function-signature>` tag. Listing 7.22 provides function definitions of the functions used in Listing 7.20 (you can find the function-taglib.tld file on the CD in the `code\Chapter7\ELFunctions\WEB-INF` folder):

Listing 7.22: Showing the Code for the function-taglib.tld File

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd"
version="2.0">

  <description>A taglib to define some EL accessible functions.</description>
  <tlib-version>1.0</tlib-version>
  <short-name>ELFunctionTaglib</short-name>
  <uri>/ELFunctionTagLibrary</uri>

  <function>
        <name>abs</name>
```

```
        <function-class>java.lang.Math</function-class>
        <function-signature>int abs( int )</function-signature>
    </function>

    <function>
        <name>round</name>
        <function-class>java.lang.Math</function-class>
        <function-signature>int round( double )</function-signature>
    </function>

</taglib>
```

Deploy the `ELFunctions` application on the Glassfish application server. Next, open Internet Explorer and type the URL http://localhost:8080/ELFunctions/functions.jsp. The functions JSP page is displayed, as shown in Figure 7.23:

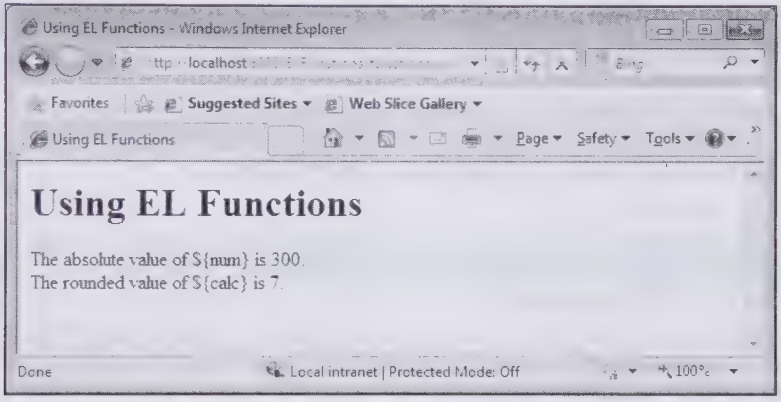


Figure 7.23: Displaying the output of EL Functions

Figure 7.23 displays the absolute value of the signed integer, i.e., 300. The second expression rounds 6.6666666667 to 7. In the `ELFunctions` application, the absolute value of the signed integer is returned by method evaluation but a null value is returned if the return type of the Java method is void.

If an exception is thrown during method evaluation, the exception must be wrapped in an `ELException` exception. This is because EL uses the `ELException` exception for any exception that may occur during parsing or expression evaluation.

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

Summary

The chapter has provided an introduction to the JSP technology. It has explored the new features introduced in JSP 2.1, and discussed the advantages of the JSP technology over Java Servlet technology. Moreover, you have learned about the different architectures of JSP pages. In addition, the chapter has explained the life cycle of a JSP page, and explored various basic tags, implicit objects, and action tags used in JSP. In the end of the chapter, you have learned about EL and the different functions used with EL.

The next chapter discusses the implementation of the various JSP tag extensions.

Quick Revise

Q1. Which of the following are the JSP life cycle methods?

A. jspInit()

B. jspService(ServletRequest, ServletResponse)

C. jspDestroy()

D. jspService(HttpServletRequest, HttpServletResponse)

Ans. The correct options are A, C, and D.

Q2. Which of the following uses of the `<jsp:useBean>` tag for a JSP page that uses the `java.sun.com.MyBean JavaBean` component are correct?

A. <jsp:useBean id="java.sun.com.MyBean" scope="page"/>

B. <jsp:useBean id="MyBean" class="java.sun.com.MyBean"/>

C. <jsp:useBean id="MyBean" type="java.lang.String" scope="page"/>

D. <jsp:useBean id="MyBean" beanName="java.sun.com.MyBean" scope="page">

Ans. The correct options are B and C.

Q3. Use ..... when resources require dynamic data or change frequently.

A. <jsp:insert>

B. <jsp:include>

C. <jsp:directive.include>

D. <%@ include %>

Ans. The correct option is B.

Q4. What is JSP?

Ans. JSP is a technology that combines the HTML/XML markup languages and elements of the Java programming language to return dynamic content to a Web client.

Q5. What are JSP directives?

Ans. Directive provides general information about a JSP page to the JSP Servlet engine. There are three types of directives: page, include, and taglib.

Q6. What is the role of EL expressions?

Ans. EL expressions are used to send a value of a Java expression to the implicit JSP object in a JSP page. In addition, JSP programmers can directly use implicit objects in an EL expression. They do not need to write extra code to use these objects. Some of these objects allow access to variables in particular JSP scopes.

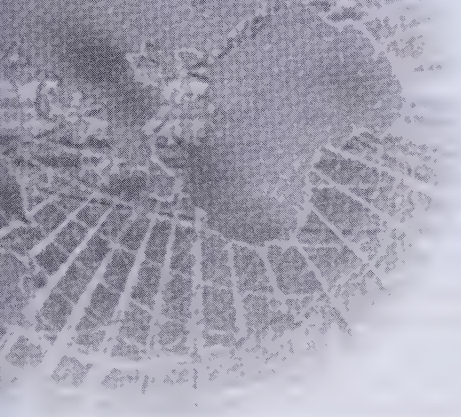
Q7. What are predefined variables or implicit objects?

Ans. To simplify code in JSP expressions and scriptlets, we use nine automatically defined variables, sometimes called implicit objects. These variables are request, response, out, session, application, config, exception, pageContext, and page.

Q8. What are JSP actions?

Ans. Actions are commands given to a JSP Servlet engine. They direct the engine to perform certain tasks during the execution of a page. The following are some examples of action tags that are used in a JSP page to define JSP actions:

- <jsp:include>
- <jsp:forward>
- <jsp:useBean>
- <jsp:setProperty>
- <jsp:getProperty>
- <jsp:plugin>



8

Implementing JSP Tag Extensions

JavaServer Pages (JSP) allows you to create and use tag extensions or custom tags in a Web page. The custom tags are defined as the user-defined tags that are present in JSP tag libraries (JSTL). These tags reduce the use of scripting elements in a JSP page. Whenever you want to use the JSP scripting elements, such as JSP expressions, scriptlets, and declaration, you need to provide Java code within these elements. The process of importing Java code in a JSP page combines the business logic of an application with the scripting elements used in a JSP. This reduces the capabilities of the presentation layer of the application. In addition, the use of Java code along with JSP elements in a JSP page leads to the difficulty in debugging and maintaining the JSP page. To overcome these problems, JSP custom tags are used, as they eliminate the need of using or importing Java code in JSP pages.

Tag extensions or custom tags allow you to create new and reusable tags that are defined in separate Java classes. These tag extensions play an important role in separating presentation layer from the business logic layer. After creating custom tags, you can use them in any JSP page with the help of their tag libraries. These tag libraries help in easy maintenance and debugging of JSP pages. Apart from tag extensions, JavaBeans are also used to separate business logic from presentation logic in a JSP page. The use of JavaBeans in a JSP page has a limitation that only three JSP standard tags, `<jsp:useBean>`, `<jsp:getProperty>`, and `<jsp:setProperty>` can be used to interact with JavaBeans. The JSP standard tags bind the instances of the scoped variables and then get or set those instances for the properties of JavaBeans.

JSP provides custom tags as extension mechanism to create new user-defined tags without the need of writing a Java code in a JSP page. Custom tags are defined in Java classes called tag handlers and are declared in a tag library with a unique name and a tag handler is used to process these tags.

This chapter begins with a discussion on the elements of tag extensions. Next, the classes and interfaces provided by the tag extension Application Programming Interface (API) are explored in detail. You also learn how to create applications of the simple and classic tag handlers. Further, the chapter explores about the JSP fragments that can be used in a JSP page. Towards the end, the chapter discusses another feature known as tag file, which allows you to create custom tags using the JSP syntax.

Exploring the Elements of Tag Extensions

A tag library provides the capability to define new and customized tags. Tag libraries can be imported into a JSP page by using the taglib directive. You can find the details of a tag library, such as name of the tags, classes to which the tags are mapped, and validator class in the Tag Library Descriptor (TLD) file. You should note that depending upon the context in which they are used, tag extensions and custom tags have been used interchangeably in this chapter.

The following are the major features of custom tags:

- **Reusability**—Allows you to reuse the custom tags in multiple JSP pages
- **Readability**—Makes the page cleaner, shorter, and readable by removing the complex Java code from the JSP page
- **Maintainability**—Specifies that an application can be modified and debugged during its lifetime by using custom tags
- **Portability**— Specifies that custom tag libraries are portable and not dependent on specific JSP containers

The custom tags are used in a `JSP` page to increase its performance and productivity. These tags generally refer to the custom actions developed by users and contain the following basic elements:

- The `TLD` file
- The tag handler
- The `taglib` directive

Let's now discuss these elements along with their usage in a JSP page.

The TLD File

The TLD file is an Extensible Markup Language (XML) document describing a tag library, which serves as a container of custom tags. These custom tags are used to encapsulate the functionalities to be provided within a JSP page. TLD is used by a JSP container to interpret the pages that include the `taglib` directives. The documentation of the tag library and version information of the JSP container for individual tags is also

provided in the TLD file. This file may contain the name of the `TagLibraryValidator` class to validate a JSP page that conforms to a set of constraints defined in a tag library. Each action in a tag library is described by giving its name, the class of its tag handler, information on any scripting variables created by the action, and information on attributes of the action. Scripting variable information can be given directly in TLD or through a `TagExtraInfo` class. The TLD file is used by JSP Engine tools without instantiating objects or loader classes. In JSP 2.1, the format of the TLD file is represented in XML schema, which provides a more extensible TLD that can be used as a single source document.

The taglib Directive

As discussed earlier, the taglib directive is used in a JSP page to implement the functionality of a custom tag. To set a custom tag in a JSP page, you need to provide the Universal Resource Identifier (URI) of the tag library and a prefix for the tag library.

The following code snippet shows how to specify a TLD in a JSP page:

```
<%@ taglib prefix="p" uri="Mytld.tld" %>
```

In the preceding code snippet, the custom tags, located in the Mytld.`tld` file, can be accessed by using the p prefix. The `taglib` directive used in a JSP page provides the following features:

- Declares that a tag library is used
- Identifies the location of the tag library with the help of the uri attribute
- Associates a tag prefix with the usage of actions in the library

The JSP container maps the uri attribute used in the `taglib` directive in the following two steps:

1. Resolves the value of the uri attribute into a TLD file resource path
2. Derives the TLD object from the TLD resource path

If the JSP container cannot locate a TLD resource path for a given URI, a fatal translation error, such as cannot load the class, may occur.

The Tag Handler

A tag handler is a Java class where a custom tag is defined. The JSP container invokes the tag handler object to evaluate a custom tag during the execution of a JSP page. You can define tag handlers for custom tags either by implementing interfaces or by extending an abstract base class from the `javax.servlet.jsp.tagext` package. The various tag handler interfaces included in JSP specifications are `Tag`, `BodyTag`, `IterationTag`, and `SimpleTag`. In addition, the JSP technology defines two types of tag handlers: classic tag handler and simple tag handler. The classic tag handlers are based on the original tag development methodology, which is used if the scripting elements are required in attribute-values pair or the tag body. On the other hand, simple tag handlers are used to evaluate custom tags that do not use scripting elements in attribute values or the tag body.

Let's now discuss the interfaces and abstract classes to create tag handlers by exploring the tag extension API.

Exploring the Tag Extension API

The tag extension API provides the `javax.servlet.jsp.tagext` package containing various classes and interfaces used to handle custom tags in a JSP page. The Tag interface works as a protocol between the tag handler and the implementation classes used in the JSP page. Tag extensions are similar to HTML/XML tags that are embedded in a JSP page.

NOTE

HTML stands for Hypertext Markup Language.

Some of the terms associated with the implementation of a tag extension are as follows:

- **Tag name**—Refers to a unique name that is a combination of prefix and suffix, separated by a colon. For example, in the <jsp:forward /> tag, jsp is a prefix and forward is a suffix.

- **Attributes**—Refers to a value that is assigned to an attribute of a tag. For example, the <jsp:forward/> tag has only one attribute, page. You should note that the use of attributes is optional.
- **Nesting**—Refers to a process in which a tag is used within another tag. For example, the <jsp:param /> tag is used within the <jsp:include /> and <jsp:forward /> tags. A tag that encloses the other tag within itself is called the parent tag and the tag that is enclosed inside the parent tag is called the child tag.
- **Body content**—Refers to the content, such as expressions and scriptlets that is provided between the start and end of a tag.

After having a brief knowledge about tag extensions and understanding their role in a JSP page, let's discuss the interfaces and classes provided in the tag extension API.

The Tag Extension Interfaces

The tag extension API provides a number of interfaces, within the javax.servlet.jsp.tagext package, to implement custom tags in a JSP page. These interfaces are used to provide interaction between the JSP engine and tag handler.

The following is the list of the tag extension interfaces defined in JSP 2.1:

- The `Tag` interface
- The `JspTag` interface
- The `IterationTag` interface
- The `BodyTag` interface
- The `DynamicAttributes` interface
- The `SimpleTag` interface
- The `TryCatchFinally` interface
- The `JspIdConsumer` interface

The `JspIdConsumer` interface is introduced in the JSP 2.1 version of the Java EE 6 platform.

The Tag Interface

The `Tag` interface defines the basic protocol between a tag handler and a JSP page. The classic tag handlers are built by using the `Tag` interface. The `Tag` interface provides various methods, such as `doStartTag()` and `doEndTag()` that are invoked in the tag handler by the JSP engine during the start and end of a tag. The `doStartTag()` method is invoked by the JSP engine while encountering the start of a tag in a JSP page. The value returned by the `doStartTag()` method indicates whether the body should be skipped or evaluated within a JSP page. The `doEndTag()` method is invoked by the JSP engine while encountering the end of a tag in a JSP page. The value returned by the `doEndTag()` method indicates whether or not the rest of the page should continue to be evaluated. The `doEndTag()` method is not executed in case an exception arises while evaluating the body content of a tag.

Table 8.1 describes various methods of the `Tag` interface:

Table 8.1: Describing the Methods of the Tag Interface

Method	Description
doStartTag()	Processes the start tag of a custom tag in the tag handler class. This method is invoked by the JSP engine in the tag handler class. The `doStartTag()` method is executed once the properties of the `pageContext` attribute are set. The doStartTag() method returns either the `Tag.EVAL_BODY_INCLUDE` field to evaluate the body of the custom tag or the `SKIP_BODY` field to skip the evaluation of the custom tag. If an error occurs during the processing of the start tag, the `JspException` exception is thrown.
doEndTag()	Processes the end tag of a custom tag in the tag handler class. This method is invoked by the JSP engine in the tag handler class. The doEndTag() method is invoked after the processing of the `doStartTag()` method is complete. The body of the custom tag may or may not be evaluated, depending on the value returned by the `doStartTag()` method.

Table 8.1: Describing the Methods of the Tag Interface	
Method	**Description**
	If the `doStartTag()` method returns the `EVAL_PAGE` field, the rest of the page continues to be evaluated. However, if the method returns the `SKIP_PAGE` field, the rest of the page is not evaluated, the request is completed, and the `doEndTag()` method of enclosing tag is not invoked. If this request is forwarded or included from another page (or servlet), only the evaluation of the current page is stopped. If an error occurs while processing the end tag, the `JspException` exception is thrown.
setParent(Tag t)	Sets the nearest enclosing tag handler for custom tag in a JSP page.
getParent()	Returns the immediate super class or ancestor class of the current class. This method is used in the nested structure of a tag handler.
release()	Releases any acquired resources.

Apart from methods, the Tag interface also provides fields, as described in Table 8.2:

Table 8.2: Describing the Fields of the Tag Interface	
Field	**Description**
`EVAL_BODY_INCLUDE`	Evaluates the body of a custom tag when this field is returned as a value for the `doStartTag()` method
`EVAL_PAGE`	Continues the evaluation of the rest of the JSP page when this field is returned as a value for the `doEndTag()` method
`SKIP_BODY`	Skips the body evaluation of the custom tag when this field is returned as a value for the `doStartTag()` and `doAfterBody()` methods
`SKIP_PAGE`	Skips the evaluation of the rest of the JSP page when this field is returned as a value for the `doEndTag()` method

The JspTag Interface

The `JspTag` interface serves as a base interface for the `Tag` and `SimpleTag` interfaces. This interface is mainly used for organizational and type-safety purposes. The `BodyTag`, `IterationTag`, `SimpleTag`, and `Tag` interfaces are the sub-interfaces of the `JspTag` interface.

The IterationTag Interface

The `IterationTag` interface is used to build tag handlers that repeatedly re-evaluate the body of a custom tag. The `IterationTag` interface extends the `Tag` interface. In other words, the `IterationTag` interface acquires all the methods of the Tag interface. Apart from the methods of the `Tag` interface, the `IterationTag` interface also provides the `doAfterBody()` method, which is invoked to determine whether or not to re-evaluate the body of the custom tag. The `doAfterBody()` method is invoked only after the body of the custom tag is evaluated.

Table 8.3 describes the method of the `IterationTag` interface:

Table 8.3: Describing the Method of the IterationTag Interface	
Method	**Description**
doAfterBody()	Determines whether or not to re-evaluate the body of a custom tag. If the body content of the custom tag is re-evaluated, the `doAfterBody()` method returns the `EVAL_BODY_AGAIN field`; otherwise, the `SKIP_BODY` field is returned. If an error occurs while processing the `doAfterBody()` method, the JspException exception is thrown.

Table 8.4 describes the fields of the `IterationTag` interface:

Table 8.4: Describing the Fields of the IterationTag Interface	
Field	**Description**
EVAL_BODY_AGAIN	Re-evaluates the body content of a tag when this field is returned by the `doAfterBody()` method
SKIP_BODY	Stops the evaluation of the body content of a custom tag, once it has been evaluated successfully

The BodyTag Interface

The `BodyTag` interface extends the `IterationTag` interface and acquires all the methods of the `IterationTag` interface. The `BodyTag` interface has some additional methods to evaluate the body content of a custom tag. Implementing the `BodyTag` interface is similar to the implementation of the `IterationTag` interface, except that the `doStartTag()` method can return the `SKIP_BODY` and `EVAL_BODY_INCLUDE` or `EVAL_BODY_BUFFERED` fields.

Table 8.5 describes various methods of the `BodyTag` interface:

Table 8.5: Describing the Methods of the BodyTag Interface	
Method	**Description**
`doInitBody()`	Evaluates the body of a custom tag before initializing it. The `doInitBody()` method is not invoked for empty tags or for non-empty tags whose `doStartTag()` method returns the `SKIP_BODY` or `EVAL_BODY_INCLUDE` field.
`setBodyContent(BodyContent b)`	Sets the body content of the custom tag. The `setBodyContent()` method is invoked by the JSP engine at every tag invocation. This method is invoked before the invocation of the `doInitBody()` method. The `setBodyContent()` method is not invoked for empty tags or for non-empty tags whose `doStartTag()` method returns the `SKIP_BODY` or `EVAL_BODY_INCLUDE` field.

The `BodyTag` interface provides the `EVAL_BODY_BUFFERED` field, which requests the creation of a new buffer to evaluate the body of the custom tag. The `EVAL_BODY_BUFFERED` field is returned as a value from the `doStartTag()` method only when the tag handler class has implemented the `BodyTag` interface.

The DynamicAttributes Interface

The `DynamicAttributes` interface is implemented by a tag handler to provide support for dynamic attributes. A tag handler, which implements the `DynamicAttributes` interface, also declares its support for dynamic attributes in TLD. The `DynamicAttributes` interface defines whether or not the tag handler accepts dynamic attributes. The tag handler should implement the `DynamicAttributes` interface, whenever it accepts the dynamic attributes; otherwise, the container considers it as an invalid tag handler.

Tag handlers implement the `DynamicAttributes` interface to accept dynamic attributes. The errors occurring at the translation-time are avoided by calling the `setDynamicAttribute()` method for the dynamic attributes that are not declared in TLD. The `DynamicAttributes` interface provides a single method, namely, `setDynamicAttribute(java.lang.String uri, java.lang.String localName, java.lang.Object value)`. The `setDynamicAttribute()` method avoids translation-time errors when a tag is declared to accept dynamic attributes that are passed as arguments and are not declared in TLD. If the tag handler does not accept the given attribute, the JspException exception is thrown. The container must not call the `doStartTag()` or `doTag()` methods for the tag.

Let's now discuss the `SimpleTag` interface of the tag extension API, which is used to build simple tag handlers.

The SimpleTag Interface

The simple tag handlers are created by implementing the `SimpleTag` interface, which defines the `doTag()` method to perform the same task as performed by the `doStartTag()`and `doEndTag()` methods of the classic tag handlers. The `doTag()` method is called only once for a given tag and performs various tasks, such as evaluating and iterating the body content of the tag.

Table 8.6 describes the methods of the `SimpleTag` interface:

Table 8.6: Describing the Methods of the SimpleTag Interface

Method	Description
`doTag()`	Handles various tasks, such as evaluating and iterating the body content of a tag. This method throws the `JspException` exception, if an error occurs while processing the custom tag. The `SkipPageException` exception is thrown, if the JSP page that (either directly or indirectly) invoked the custom tag needs to cease evaluation.
`getParent()`	Returns the parent of the custom tag for collaboration purposes.
`setJspBody(JspFragment jspBody)`	Provides the body of the custom tag as a `JspFragment` object. The `setJspBody()` method is invoked zero or more times prior to the `doTag()` method. If the `action` element is empty in the current JSP page, this method is not invoked.
`setJspContext(JspContext pc)`	Provides the simple tag handler with the `JspContext` object.
`setParent(JspTag parent)`	Sets the parent of the custom tag for collaboration purposes. The `setParent()` method is invoked when the current tag is nested within another tag invocation.

The TryCatchFinally Interface

The `TryCatchFinally` interface is used to handle the exceptions that may occur within the tag handler class. When the JSP container invokes the tag handler methods, such as `doStartTag()` and `doEndTag()`, these methods may throw the `JspException` exception. Therefore, you need to implement the `TryCatchFinally` interface in the tag handler class to handle such exceptions.

Table 8.7 describes the methods of the `TryCatchFinally` interface:

Table 8.7: Describing the Methods of the TryCatchFinally Interface

Method	Description
`doCatch(Throwable)`	Gets invoked whenever there an exception arises during the evaluation of the body of the current tag or whenever the `doStartTag()`, `doEndTag()`, `doInitBody()`, and `doAfterBody()` methods are invoked.
`doFinally()`	Gets invoked after the processing of the `doEndTag()` method is complete. This method can also be invoked if an exception occurs while evaluating the body of the current tag or when the `doStartTag ()`, `doEndTag ()`, `doInitBody ()`, and `doAfterBody ()` methods are invoked.

The JspIdConsumer Interface

The `JspIdConsumer` interface indicates that the JSP container should provide a compiler generated id to a tag handler. The `jspId` attribute of the tag handler is set by the container, which acts as the part of the tag property. Every tag of a JSP page has a unique id that can be used to access the tags of a tag handler. The `jspId` attribute is a String value and is similar to the `<jsp:id>` tag; however, the only difference is that `<jsp:id>` is evaluated at the execution time, while jspId is evaluated at the request time.

The `JspIdConsumer` interface provides a single method, `setJspId(String id)`, which helps in generating a unique identification during the translation process performed by the Web container.

After understanding the interfaces provided by the tag extensions API, let's now explore the tag extension classes.

The Tag Extension Classes

The tag extension API provides a number of classes, within the `javax.servlet.jsp.tagext` package, to implement custom tags. You should note that these classes are abstract in nature and help to establish the interaction between the JSP engine and a tag handler.

The following is the list of the tag extension classes defined in JSP 2.1:

- The `TagSupport` class
- The `BodyContent` class
- The `BodyTagSupport` class
- The `SimpleTagSupport` class
- The `TagAdapter` class
- The `TagLibraryInfo` class
- The `TagInfo` class
- The `TagFileInfo` class
- The `TagAttributeInfo` class
- The `PageData` class
- The `TagLibraryValidator` class
- The `ValidationMessage` class
- The `TagData` class
- The `VariableInfo` class
- The `TagVariableInfo` class
- The `FunctionInfo` class
- The `JspFragment` class

Let's now discuss each of these classes in detail.

The TagSupport Class

The `TagSupport` class is an abstract class that implements the `Tag` as well as `IterationTag` interfaces and provides support for all the methods of these interfaces. In addition, the `TagSupport` class provides a static method, which facilitates coordination among other cooperating tags.

Table 8.8 describes various methods of the `TagSupport` class:

Table 8.8: Describing the Methods of the TagSupport Class

Method	Description
`doAfterBody()`	Processes the body content of a tag after the invocation of the doStartTag() method. If an error occurs while processing the doAfterBody() method on the current tag, the `JspException` exception is thrown.
`doEndTag()`	Returns the `EVAL_PAGE` field, by default, on executing the end tag. If an error occurs while processing the end tag, the `JspException` exception is thrown.
`doStartTag()`	Returns the `SKIP_BODY` field after the default execution of the `start` tag. If an error occurs during the execution of the start tag, the `JspException` exception is thrown.
`getValue(java.lang.String k)`	Returns the String value associated with the specified key.
`getId()`	Returns either the values of the `id` attribute of a custom tag or the null value.

Table 8.8: Describing the Methods of the TagSupport Class	
Method	**Description**
`getValues()`	Enumerates through the keys to get the corresponding values.
`findAncestorWithClass(Tag from, java.lang.Class class)`	Returns the nearest class to the class in which instance is passed as a parameter of the method. The specified class can be used as replacement of the `getParent()` method of the `Tag` interface.
`removeValue(java.lang.String k)`	Removes the String value associated with the specified key.
`setId(java.lang.String id)`	Sets the `id` attribute for the current tag handler.
`setPageContext(PageContext pageContext)`	Sets the page context for the current tag handler.

Table 8.9 describes the fields of the `TagSupport` class:

Table 8.9: Describing the Fields Available in the TagSupport Class	
Field	**Description**
`id`	Specifies a String value for a tag
`PageContext`	Provides access to all the namespaces and page attributes associated with a JSP page

Let's discuss the BodyContent class of the tag extension API.

The BodyContent Class

The `BodyContent` class is a subclass of the `JspWriter` class and encapsulates the evaluation of the body of an action. This class does not contain any actions; instead, contains only the result of invocation of the actions.

The `BodyContent` class contains several methods to read its contents, convert the content into String, and clear the contents. The buffer size of the `BodyContent` class is not limited. You should note that the autoflush property cannot be enabled for this class because no backup stream is available for this class. To create instances of the BodyContent class, you can use either the `pushBody()` or `popBody()` method of the `PageContext` class.

Table 8.10 describes various methods of the `BodyContent` class:

Table 8.10: Describing the Methods of the BodyContent Class	
Method	**Description**
`getReader()`	Returns the value of the `BodyContent` object as a `Reader` object.
`getString()`	Returns the value of the `BodyContent` object as a `String`.
`writeOut(java.io.Writer out)`	Writes the contents of the `BodyContent` object into a `Writer` object. If an I/O error occurs while writing the contents of the `BodyContent` object into the given `Writer` object, the `IOException` exception is thrown.
`getEnclosingWriter()`	Returns the enclosing `JspWriter` object.
`clearBody()`	Clears the body of the buffer without throwing any exceptions.

Let's discuss the `BodyTagSupport` class of the tag extension API.

The BodyTagSupport Class

The `BodyTagSupport` class implements the `BodyTag` interface and acquires all the methods of the `BodyTag` interface with some additional methods of its own. This class acts as a base class to create tag handlers without defining each method of the `BodyTag` interface.

Table 8.11 describes the methods of the BodyTagSupport class:

Table 8.11: Describing the Methods of the BodyTagSupport Class	
Method	**Description**
getBodyContent()	Returns the current BodyContent object.
doAfterBody()	Determines whether or not to re-evaluate the body of a custom tag. When the doAfterBody() method returns the EVAL_BODY_AGAIN field, the body of a custom tag is re-evaluated. In case, it returns the SKIP_BODY field, the body content is not evaluated. If an error occurs while processing the doAfterBody() method, the JspException exception is thrown.
doEndTag()	Processes the end tag and returns the EVAL_PAGE field. If an error occurs while processing the doEndTag() method, the JspException exception is thrown.
doStartTag()	Returns the EVAL_BODY_BUFFERED field after the execution of the start tag. If an error occurs while processing the start tag, the JspException exception is thrown.
getPreviousOut()	Returns the enclosing JspWriter object from the bodyContent object.
doInitBody()	Prepares to evaluate the body of a custom tag before evaluating the body for the first time. If an error occurs while processing the custom tag, the JspException exception is thrown.
release()	Releases any acquired resources.
setBodyContent(BodyContent b)	Sets the body content of the custom tag.

The BodyTagSupport class provides a single field, namely, bodyContent, which specifies the body content for the tag handler.

The FunctionInfo Class

The FunctionInfo class provides information about a function defined in the tag library. This class is instantiated from the TLD file and is available only at the time of translation of a JSP page.

Table 8.12 describes the methods of the FunctionInfo class:

Table 8.12: Describing the Methods of the FunctionInfo Class	
Method	**Description**
getFunctionClass()	Returns the class of a function
getFunctionSignature()	Returns the signature of a function
getName()	Returns the name of a function

The JspFragment Class

The JspFragment class encapsulates the JSP code in an object that can be invoked multiple times. This class is generated manually either by the page author or by the TLD file.

NOTE

Page author is responsible for developing the JSP page and decides which directives and actions should be used in the JSP page.

Table 8.13 describes the methods of the `JspFragment` class:

Table 8.13: Describing Methods of the JspFragment Class

Method	Description
`getJspContext()`	Returns the `JspContext` object that is used by the current `JspFragment` object at the time of invocation of the tag
`invoke``(java.io.Writer out)`	Executes the `JspFragment` object and passes all the output to the `Writer` or `JspWriter` object that is obtained from the `getOut()` method of the `JspContext` class

The SimpleTagSupport Class

The `SimpleTagSupport` class implements the `SimpleTag` interface. This class provides the definition for all the implemented methods of the `SimpleTag` interface and eliminates the need to define each method in a simple tag handler.

Table 8.14 describes the methods of the `SimpleTagSupport` class:

Table 8.14: Describing the Methods of the SimpleTagSupport Class

Method	Description
`doTag()`	Processes a simple tag handler. The `JspException` exception is thrown to indicate that an error has occurred while processing the simple tag handler. The `SkipPageException` exception is thrown, if the JSP page that invoked the custom tag has to cease evaluation. A simple tag handler generated from a tag file must throw this exception if an invoked classic tag handler returns the `SKIP_PAGE` field or the `SkipPageException` exception.
`findAncestorWithClass(JspTag from, java.lang.Class class)`	Returns the nearest instance of the class specified as a parameter to the `SimpleTagSupport` class. The `findAncestorWithClass()` method uses the `getParent()` method from the `Tag` or `SimpleTag` interface for coordination among cooperating tags.
`getJspBody()`	Returns a fragment that encapsulates the body passed by the container through the `setJspBody()` method.
`getJspContext()`	Returns the page context passed by the container through the `setJspContext` object.
`getParent()`	Returns the parent tag of a simple tag on which the `getParent()` method is called for collaboration purposes.
`setJspBody(JspFragment jspBody)`	Stores the specified `JspFragment` object that encapsulates the body of the custom tag. The `setJspBody()` method is not called if the body of a simple tag is empty.
`setJspContext(JspContext pc)`	Stores the provided JSP context in the private `JspContext` field. A sub class can access JspContext through the `getJspContext()` method.
`setParent(JspTag parent)`	Sets the parent tag of the simple tag for collaboration purposes. The `setParent()` method is invoked when a simple tag is invoked within another tag invocation.

Let's now discuss the `TagAdapter` class.

The TagAdapter Class

The `TagAdapter` class allows collaboration between classic tag handlers and simple tag handlers. The `TagAdapter` class wraps the functionality of a simple tag handler in an object of type, `TagAdapter` class. The wrapped object of a simple tag handler is passed to the `setParent()` method of the `Tag` interface. To retrieve

the instance of the SimpleTag interface, the classic tag handler needs to invoke the getAdaptee() method. The classic tag handlers use the functionality of the wrapped object of simple tag handler with the help of the Tag interface. The constructor of the TagAdapter class creates a new TagAdapter object that wraps the given instance of a simple tag and returns the parent tag when the getParent() method is invoked.

Table 8.15 describes the methods of the TagAdapter class:

Table 8.15: Describing the Methods of the TagAdapter Class

Method	Description
getAdaptee()	Gets an instance of simple tag handler that is being adapted to the Tag interface
getParent()	Returns the parent of the simple tag

The tag extension API contains classes that provide translation-time information for processing custom tags in a JSP page. The classes that provide translation-time processing information are TagLibraryInfo, TagInfo, TagFileInfo, TagAttributeInfo, and PageData.

The TagLibraryInfo Class

The TagLibraryInfo class provides translation-time information associated with a taglib directive and TLD. Table 8.16 describes the methods of the TagLibraryInfo class:

Table 8.16: Describing the Methods of the TagLibraryInfo Class

Method	Description
getFunction(java.lang.String name)	Returns the information of a specified function by searching all the functions in a tag library. The getFunction() method returns null if the specified function does not exist.
getFunctions()	Returns an array of all the functions that are defined in a tag library. The getFunctions() method returns a zero length array if no functions are defined in the tag library.
getInfoString()	Returns the information (documentation) of TLD as a String value.
getPrefixString()	Returns the name (prefix) of custom action that is assigned in the taglib directive.
getReliableURN()	Returns Uniform Resource Name (URN) specified in the uri element of TLD.
getRequiredVersion()	Determines the required version of the JSP container as a String.
getShortName()	Returns the prefix as indicated in TLD.
getTag(java.lang.String shortname)	Returns the TagInfo object for the specified tag name by searching through all the tags in a tag library. The getTag() method returns null if the specified tag is not found.
getTagFile(java.lang.String shortname)	Returns the TagFileInfo object for the specified tag name by looking through all the tag files in a tag library. The getTagFile() method returns null if the tag with the specified shortname is not found.
getTagFiles()	Returns an array describing the tag files that are defined in a tag library The getTagFiles() method returns zero length array if the tag library defines no tag files.
getTags()	Returns an array describing the tags that are defined in tag library. The getTags() method returns a zero length array if no tags are defined in the tag library.

Table 8.16: Describing the Methods of the TagLibraryInfo Class	
Method	**Description**
`getURI()`	Returns the String value associated with the `uri` attribute of the `taglib` directive for the current library.

Table 8.17 describes the fields of the `TagLibraryInfo` class:

Table 8.17: Describing the Fields of the TagLibraryInfo Class	
Fields	**Description**
`functions`	Specifies an array describing the functions that are defined in a tag library
`info`	Specifies the information (documentation) for the TLD file
`jspversion`	Specifies the version of the JSP specification in which the tag library is written
`prefix`	Specifies the prefix assigned to the `taglib` from the `taglib` directive
`shortname`	Specifies the preferred short name (prefix) as indicated in TLD
`tagFiles`	Returns an array of the `TagFileInfo` type containing the tag files of a tag library
`tags`	Returns an array of the `TagInfo` class that provides the description of tags in the current tag library
`tlibversion`	Specifies the version of a tag library
`uri`	Specifies the String value associated with the uri attribute of the `taglib` directive for the current tag library
`urn`	Specifies the reliable `URN` indicated in TLD

The TagInfo Class

The `TagInfo` class provides information of a specific tag provided in a tag library. The `TagInfo` class is an instance of TLD.

Table 8.18 describes the methods of the `TagInfo` class:

Table 8.18: Describing the Methods of the TagInfo Class	
Method	**Description**
`getAttributes()`	Returns an array of tag attributes, which are defined in TLD. The `getAttributeInfo()` method returns zero length array if the tag has no attributes.
`getBodyContent()`	Returns the body content of the tag as a String value.
`getDisplayName()`	Returns the short name of the tag.
`getInfoString()`	Returns the information of the tag as a String value.
`getLargeIcon()`	Returns the path to a large icon.
`getSmallIcon()`	Returns the path to a small icon.
`getTagClassName()`	Returns the name of the tag handler class for the current tag.
`getTagExtraInfo()`	Returns the `TagExtraInfo` instance for extra tag information.
`getTagLibrary()`	Returns the instance of the `TagLibraryInfo` class.

Table 8.18: Describing the Methods of the TagInfo Class

Method	Description
`getTagName()`	Returns the name of tag.
`getTagVariableInfos()`	Returns an array of the `TagVariableInfo` objects that are corresponding to the variables specified by this tag; otherwise, returns a zero length array, if no variables are specified.
`getVariableInfo(TagData data)`	Returns information on the scripting objects created by the tag at runtime.
`hasDynamicAttributes()`	Specifies whether or not the tag handler supports dynamic attributes. The `hasDynamicAttributes()` method returns true if the tag handler supports dynamic attributes.
`isValid(TagData data)`	Specifies whether or not the instance of the `TagData` class passed as a parameter is valid.
`setTagExtraInfo(TagExtraInfo tei)`	Sets extra tag information by setting the instance of the `ExtraTagInfo` class.
`setTagLibrary(TagLibraryInfo tl)`	Sets the `TagLibraryInfo` element as it is dependent on TLD information as well as on specific tag library instance.
`validate(TagData data)`	Performs translation-time validation of the specified `TagData` instance and returns an array of the `ValidationMessage` class. The `validate()` method returns a zero length array if no error occurs while validating the `TagData` instance.

Table 8.19 describes the field of the `TagInfo` class:

Table 8.19: Describing the Fields of the TagInfo Class

Field	Description
`BODY_CONTENT_EMPTY`	Assigns a static constant for the `getBodyContent()` method when the body content is empty
`BODY_CONTENT_JSP`	Assigns a static constant for the `getBodyContent()` method when the body content is in JSP
`BODY_CONTENT_SCRIPTLESS`	Assigns a static constant for the `getBodyContent()` method when the body content is scriptless
`BODY_CONTENT_TAG_DEPENDENT`	Assigns a static constant for the `getBodyContent()` method when the body content is tag dependent

The TagFileInfo Class

The `TagFileInfo` class provides information about a tag file in a tag library. TLD instantiates the `TagFileInfo` class and is available only at translation-time. The following code snippet shows the constructor of the `TagFileInfo` class:

```
TagFileInfo(name, path, TagInfo)
```

The TagFileInfo class should be instantiated only from the tag library code, which is used for parsing a TLD in a JSP page. The following parameters are supplied to the constructor of the TagFileInfo class:

- **name**—Specifies a unique name for a tag
- **path**—Specifies a path to locate the .tag file implementing the `TagFileInfo` class
- **TagInfo**—Specifies the information about a tag

Table 8.20 describes the methods of the TagFileInfo class:

Table 8.20: Describing the Methods of the TagFileInfo Class

Method	Description
`getName()`	Returns a unique action name of the tag
`getPath()`	Returns a path to locate the .tag file
`getTagInfo()`	Returns a `TagInfo` object containing the information about the tag

The TagAttributeInfo Class

The `TagAttributeInfo` class contains information about tag attributes and this information is available at translation-time. TLD instantiates the `TagAttributeInfo` class, which has ID as its attribute. This ID attribute specifies a String value for a custom tag.

Table 8.21 describes the methods of the `TagAttributeInfo` class:

Table 8.21: Describing the Methods of the TagAttributeInfo Class

Methods	Description
`canBeRequestTime()`	Returns true if an attribute can hold a request-time value.
`getIdAttribute(TagAttributeInfo[] a)`	Returns the `TagAttributeInfo` reference with name `id`. The `getIdAttribute()` method goes through an array of `TagAttributeInfo` objects and searches the specific `id`.
`getName()`	Returns the name of the attribute.
`getTypeName()`	Returns the type of attribute as String.
`isFragment()`	Returns true if a tag attribute represents the `JspFragment` class.
`isRequired()`	Returns true if the attribute is required.
`toString()`	Returns a `String` representation of the current `TagAttributeInfo` object.

The TagExtraInfo Class

The `TagExtraInfo` class provides a mechanism to validate custom tags using tag attributes in TLD. The `<attribute>` tag contains a set of three tags, `<name>`, `<required>`, and `<rtexprvalue>`. The `<name>` tag specifies a name to an attribute, the `<required>` tag specifies whether an attribute is mandatory or not, and the `<rtexprvalue>` tag specifies whether or not the value is a runtime expression. These three tags provide a low level validation mechanism for custom tags. The JSP engine validates the syntax of tag attributes provided in a TLD file during the translation-time.

Table 8.22 describes the methods of the `TagExtraInfo` class:

Table 8.22: Describing the Methods of the TagExtraInfo Class

Method	Description
`getTagInfo()`	Returns the `TagInfo` object of the custom tag.
`getVariableInfo(TagData data)`	Returns an array of scripting variables defined by the specified tag. The `getVariableInfo()` method returns a zero length array, if no scripting variables are defined.
`isValid(TagData data)`	Returns whether or not the specified `TagData` instance is valid.
`setTagInfo(TagInfo tagInfo)`	Sets the specified `TagInfo` for the custom tag handler.

Table 8.22: Describing the Methods of the TagExtraInfo Class	
Method	**Description**
`validate(TagData data)`	Returns an array of the `ValidationMessages` object, which is the result of the translation-time validation of attributes. The `validate()` method returns a null object or a zero length array in case of no errors.

The PageData Class

The `PageData` class provides the translation-time information on a JSP page. This information corresponds to the XML view of the JSP page. Objects of the `PageData` class are generated by the JSP translator, when being passed to a `TagLibraryValidator` instance. The `PageData` class provides a single method, i.e. `getInputStream()`, which returns an input stream of a JSP page in the form of XML and the Stream encoding is done in UTF-8.

NOTE

UTF stands for Unicode Transformation Format.

The VariableInfo Class

The `VariableInfo` class determines the type of the scripting variables that are created or modified by a tag at runtime. Values for the scripting variables are assigned from scoped attributes. These scripting variables cannot be of primitive datatypes. In the object of the `VariableInfo` class, you can either provide fully qualified class name or short class name. You should ensure that when fully qualified class name is given, the specified class should be provided in the classpath of the Web application. If short class name is given in the VariableInfo objects, it should be in the context of the import directives of a JSP page and the classpath should remain in the Web application.

Table 8.23 describes the methods of the `VariableInfo` class:

Table 8.23: Describing the Methods of the VariableInfo Class	
Method	**Description**
`getClassName()`	Returns the type of the specified scripting variable
`getDeclare()`	Declares a scripting variable during runtime
`getScope()`	Returns the scope of the variable, such as AT_BEGIN, AT_END, or NESTED
`getVarName()`	Returns the name of the scripting variable

Table 8.24 describes the fields of the `VariableInfo` class:

Table 8.24: Describing the Fields of the VariableInfo Class	
Field	**Description**
AT_BEGIN	Specifies the scope information of scripting variable, which is visible after the execution of the start tag.
AT_END	Specifies the scope information of scripting variable, which is visible after the execution of the end tag.
NESTED	Specifies the scope information of scripting variable, which is visible only within the start/end tags.

The TagData Class

The object of the `TagData` class is used as an argument in the `validate()`, `isValid()`, `getVariableInfo()` methods of the `TagExtraInfo` class at the translation-time.

Table 8.25 describes the methods of the TagData class:

Table 8.25: Describing the Methods of the TagData Class

Method	Description
getAttribute(java.lang.String attName)	Returns the value associated with an attribute. If a static value is specified with the attribute, then this method returns a different value specified in the tag. If a value is not specified, then this method returns null.
getAttributes()	Returns an enumeration of the attributes specified with respect to the TagData class.
getAttributeString(java.lang.String attName)	Returns the value of a given attribute.
getId()	Returns the value of an id attribute of a tag, or null if no such attribute exists.
setAttribute(java.lang.String attName, java.lang.Object value)	Sets the value of an attribute.

The TagData class provides a single field, REQUEST_TIME_VALUE, which specifies a value as a request time expression. The tag extension API provides classes, such as TagLibraryValidator and ValidationMessage to validate a JSP page.

The TagLibraryValidator Class

The TagLibraryValidator class validates a JSP page during the translation-time and performs the validation on the XML representation of the JSP page. This class provides an important method, validate() to validate the page. The JSP engine calls the validate() method while compiling the JSP page using the taglib directive. The validate(String prefix, String uri, PageData page) method contains three parameters, prefix, uri, and page. The first two parameters are used to indicate how a tag library is mapped in the JSP page, while the last parameter is used to represent the XML view of the page being validated. The validate() method returns the list of sources of errors if the page is not valid; otherwise, it returns null in case it is valid.

Table 8.26 describes the methods of the TagLibraryValidator class:

Table 8.26: Describing the Methods of the TagLibraryValidator Class

Method	Description
getInitParameters()	Returns the init parameters data as the Map object (key-value pair).
release()	Releases any data kept by the instance of the TagLibraryValidator class for validation purposes.
setInitParameters(java.util.Map map)	Sets the init parameter as the name-value pair in TLD for the TagLibraryValidator object.
validate(java.lang.String prefix, java.lang.String uri, PageData page)	Allows you to validate a JSP page. The validate() method is invoked once for every unique tag library URI in the XML. If a page is valid, then this method returns null; otherwise, it returns an array of the ValidationMessage objects.

The ValidationMessage Class

A validation message is a piece of information that is provided by either the TagLibraryValidator or the TagExtraInfo object. A JSP container supports the <jsp:id> tag for high quality validation errors. This container tracks all the JSP pages passed to it and provides them a unique id, which is similar to the value of the <jsp:id> attribute. Each XML element in the XML view is recognized by this attribute. Then, this attribute can

be used by the TagLibraryValidator class in one or more ValidationMessage object's attribute. The JSP container can also use this ValidationMessage object to retrieve information about the location of errors.

Table 8.27 describes the methods of the ValidationMessage class:

Table 8.27: Describing the Methods of the ValidationMessage Class

Method	Description
getId()	Returns the <jsp:id> attribute. However, if there is no information about the ValidationMessage object, a null value is returned.
getMessage()	Returns the localized validation message.

Let's now create an application for classic tag handlers in the next section.

Working with Classic Tag Handlers

A classic tag handler is a Java class that implements the Tag, IterationTag, or BodyTag interface. The implementation class instantiates a tag handler object or reuses an existing tag handler object for each action provided in the JSP page. The handler object is responsible for the interaction between the JSP page and additional server-side objects.

Exploring the Life Cycle of Classic Tag Handlers

The life cycle of a classic tag handler begins with the creation of a tag handler instance. Context setting as well as parent setting is also a part of the life cycle process of a tag handler. A classic tag handler goes through the following seven phases in its life-time:

- **Tag handler instance creation**-Refers to a phase in which a JSP engine finds a new tag in the JSP page and creates an instance of the tag handler to handle the tag. This instance is then pooled by the container to be reused whenever the tag needs to be invoked.
- **Context setting**—Refers to a phase in which the tag handler instance is made aware of its execution environment. This is done by passing a reference of the current PageContext method to the tag handler instance through the setPageContext() method.
- **Parent setting**—Refers to the phase in which nesting of classic tags is possible as tag handlers can communicate with each other. For example, a nested tag can ask its parent tag for information; therefore, the setParent() method is invoked and the reference of parent tag is passed to the closest tag handler. In case of the non nested tag handler, a null value is passed to the tag handler.
- **Executing doStartTag() method**—Refers to the phase in which the JSP engine processes the start tag for the instance of tag handler. This method is invoked by the JSP page implementation object. This method returns the following constants to determine whether the tag body should be evaluated or not:
  - **SKIP_BODY**—Indicates the JSP engine to skip the body content of the tag. The SKIP_BODY field is returned in case of an empty tag. The body of the tag is skipped and the doEndTag() method is called to close the tag.
  - EVAL_BODY_INCLUDE—Indicates a constant returned by the doStartTag() method of the tag handler that implements the Tag interface, if the body of a tag needs to be evaluated.
  - EVAL_BODY_TAG—Indicates a constant that is used to evaluate the body content of the tag. This is generally used if the tag implements the BodyTag interface or extends the BodyTagSupport class.
- **Executing doEndTag() method**—Refers to the phase in which the doEndTag() method is invoked to close all the connections made in the JSP page. The tag handler calls this method to complete any server-side work and is also used to write the output in the JspWriter object. The tag handler can directly write the output generated by the pageContext.getOut() to the JspWriter object. The output can also be written to the enclosing writer by using the popBody() method in the pageContext. The following two constants are retrieved from the method to control the flow of evaluation:
  - EVAL_PAGE— Represents the constant that is used to continue the processing of the JSP page.

- SKIP_PAGE—Represents the constant that is used to force the JSP engine to skip the evaluation of the page.

- **Releasing State**—Refers to the phase in which the release() method is invoked to de-reference all the instances made by the tag handler and also to perform garbage collection.

Implementing Classic Tags

To understand how the classic tag handler is implemented, let's create a tag handler class, a TLD file, and a JSP file. Let's create a Tag class, CustomMessage.java, which extends the BodyTagSupport class that returns a message. The following are the broad-level steps to develop the classicTag application:

- Creating the tag handler for the classicTag application
- Creating TLD for the classicTag application
- Creating the home.html file for the classicTag application
- Creating the JSP file for the classicTag application
- Configuring the web.xml file for the classicTag application
- Defining the directory structure of the classicTag application
- Packaging, running, and deploying the classicTag application

Now, let's discuss each of these steps in detail.

Creating the Tag Handler for the classicTag Application

The tag handler contains the functionality of the tag coded inside it. Listing 8.1 provides the code for tag handler file, CustomMessage.java (you can find this file on the CD in the code\JavaEE\Chapter8\classicTag\src\com\kogent\tags folder):

Listing 8.1: Showing the Code of the CustomMessage.java File

```
package com.kogent.tags;
import javax.servlet.ServletRequest;
import javax.servlet.jsp.tagext.*;

public class CustomMessage extends BodyTagSupport

{

  private static final long serialVersionUID = 1L;
  public void setParamName(String s)
  {
        pname=s;
  }
  public String getParamName()
  {
        return pname;
  }
  public int doStartTag()
  {
        ServletRequest req=pageContext.getRequest();

        String pvalue=req.getParameter(pname);
        if ((pvalue.equals("japan")) || (pvalue.equals("Japan")))
        {
              return EVAL_BODY_INCLUDE;
        }
        else
        {

              return SKIP_BODY;
        }
  } //doStartTag//doStartTag
```

```
    public int doAfterBody()
    {
          return SKIP_BODY;
    }//doAfterBody

    public int doEndTag()
    {
          return EVAL_PAGE;
    }//doEndTag

    String pname;
}
```

In Listing 8.1, a classic tag handler class (`CustomMessage`) is created by extending the `BodyTagSupport` class to define the behavior of the `CustomMessage` tag, which is an empty tag. The JSP container starts the processing of a custom tag by invoking the `doStartTag()` method of the tag handler class. The `doStartTag()` method returns the `SKIP_BODY` constant to skip the execution of the body of the tag. After the successful execution of the `doStartTag()` method, the `doEndTag()` method is invoked that returns the `EVAL_PAGE` constant to evaluate the rest of the JSP page. Save the `CustomMessage.java` file in the `C:/JavaEE/Chapter 8/classicTag/src/com/kogent/tags` folder.

Creating TLD for the classicTag Application

All the custom tags must be declared in TLD. Listing 8.2 shows the declaration of the `CustomMessage` tag in the `CustomTags.tld` file (you can find this file on the CD in the `code\JavaEE\Chapter8\classicTag\WEB-INF` folder):

Listing 8.2: Showing the Code of the `CustomTags.tld` File

```
<?xml version="1.0" ?>
<taglib version="2.1" 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/web-jsptaglibrary_2_1.xsd">
  <jsp-version>2.1</jsp-version>
  <tlib-version>1.1</tlib-version>
  <short-name>tag</short-name>

  <tag>
        <name>check</name>
        <tag-class>com.kogent.tags.CustomMessage</tag-class>
        <body-content>JSP</body-content>

        <attribute>
              <name>paramName</name>
              <required>true</required>
              <rtexprvalue>false</rtexprvalue>
        </attribute>
  </tag>
</taglib>
```

In Listing 8.2, a TLD file, `CustomTags.tld`, is created where the classic tag named check is declared. TLD contains a parent tag named `<tag>` containing various child tags, such as `<name >`, `<tag-class>`, and `<body-content>`. The `<name>` tag is used to specify a name to the tag; whereas, the `<tag-class>` specifies a tag handler class encapsulating the functionality. Lastly, the `<body-content>` tag specifies whether the tag is empty or not. The `<body-content>` has been set to JSP, as the tag used in Listing 8.2 is not empty. Save the file in the `code\JavaEE\Chapter8\classicTag\WEB-INF` folder.

Creating the home.html File for the classicTag Application

The `classicTag` application displays the `home.html` file as the home page, which contains a `check` button. When a user clicks the checks button, the control is sent to the `TestPage.jsp` file, which further processes the

client request. Listing 8.3 shows the source code for the `home.html` file (you can find this file on the CD in the `code\JavaEE\Chapter8\classicTag` folder):

Listing 8.3: Showing the `home.html` File

```
<html>
  <body>
        <form action="TestPage.jsp">
              <b> PREDICT AND WIN!!!!</b>
              <BR/>
              <BR/>
              <p style="color:blue">The country which is known as "land of rising
              sun"</p>
              <input type="text" name="opt"/>
              <input type="submit" value="check"/>
        </form>
  </body>
</html>
```

Save the file in `C:/JavaEE/Chapter8/classicTag` folder. Let's now create the JSP file for the `classicTag` application.

Creating the JSP File for the classicTag Application

When the JSP container finds a custom tag in the JSP page, it searches for TLD in the `WEB-INF/tld` folder and executes the custom tag. The JSP file is created to use the custom tag by importing the tag library using the `taglib` directive. The code for the `TestPage.jsp` file is shown in Listing 8.4 (you can find this file on the CD in the `code\JavaEE\Chapter8\classicTag` folder):

Listing 8.4: Showing the `TestPage.jsp` File

```
<%@taglib uri="/WEB-INF/CustomTags.tld" prefix="tag"%>
<html>
  <body>
        <tag:check paramName = "opt">
              <b>congratulations you have won a prize!!!</b>
        </tag:check>
  </body>
</html>
```

In Listing 8.4, the `TestPage.jsp` file imports a tag library using the `taglib` directive. It then uses the `<CustomMessage>` tag to access the message associated with the `CustomMessage` tag. Save the file in the `C:/JavaEE/Chapter8/classicTag` folder.

Configuring the web.xml File for the classicTag Application

Configure the `web.xml` file for classic tags and set `home.html` as the welcome file, as shown in Listing 8.5 (you can find this file on the CD in the `code\JavaEE\Chapter8\classicTag\WEB-INF` folder):

Listing 8.5: Showing the `web.xml` File for the `classicTag` Application

```
<?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">

  <welcome-file-list>
        <welcome-file>home.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Save the file in the `C:/JavaEE/Chapter8/classicTag/WEB-INF` folder.

Defining the Directory Structure of the classicTag Application

After the files are created and the application is configured, you need to define the directory structure of the `classicTag` application. Figure 8.1 displays the directory structure of the `classicTag` application:

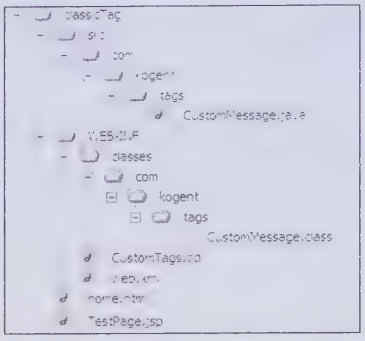


Figure 8.1: Showing the Directory Structure of the classicTag Application

Packaging, Running, and Deploying the classicTag Application

To package the `classicTag` application, you first need to create the `classicTag.war` file. Then, deploy this Web ARchive (WAR) file as a Web application on Glassfish application server. Browse `http://localhost:8080/classicTag` URL to run the classicTag application, as shown in Figure 8.2:

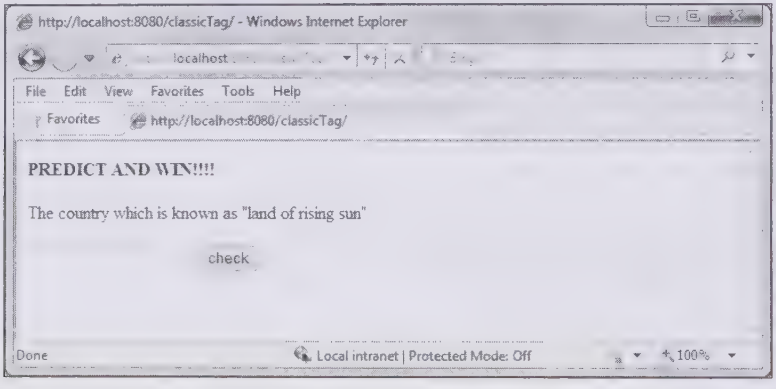


Figure 8.2: Displaying an HTML Page as the Home Page of the classicTag Application

In Figure 8.2, an HTML page that accepts a value from the user is displayed. In our case, we have entered Japan, as shown in Figure 8.3:

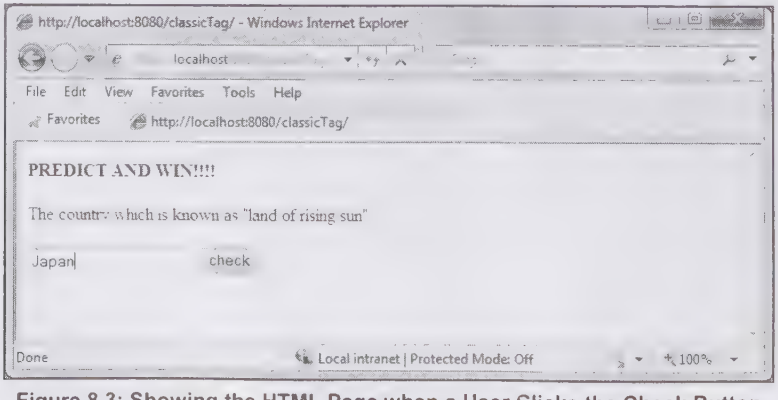


Figure 8.3: Showing the HTML Page when a User Clicks the Check Button

If the user enters the correct value, then a message is displayed, as shown in Figure 8.4:

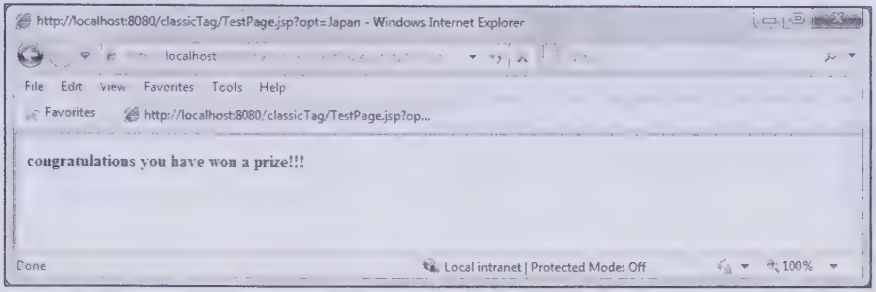


Figure 8.4: Showing an Output when an Answer Entered by a User is Correct

Now, let's create an application on simple tag handlers in the next section.

Working with Simple Tag Handlers

A simple tag handler is a Java class that implements the `SimpleTag` interface and has a no-argument constructor. This class is mainly used by authors to make the use of tags flexible in tag handlers. The `javax.servlet.jsp.tagext.SimpleTagSupport` class provides a default implementation of all the methods in simple tags.

In this section, let's first discuss the life cycle of simple tag handlers and then create the `simpleTag` application demonstrating the implementation of simple tag handlers.

Exploring the Life Cycle of Simple Tag Handlers

When a `JSP` container encounters a custom tag, such as a simple tag in a `JSP` page, it goes through the following phases:

- **Tag handler instance creation**—Refers to a phase in which a JSP engine finds a new tag in the JSP page and creates an instance of the tag handler to handle the tag. Unlike classic tag handlers, this instance of tag handler is never pooled by the container. Therefore, the instantiation of a tag is required, whenever a tag is invoked.
- **Context setting**—Refers to a phase in which the tag handler instance is made aware of its execution environment. This is done by passing the reference of current JspContext to the tag handler instance through the setJspContext() method.
- **Parent setting**—Refers to a phase in which `the setParent()` method is invoked on the tag handler instance.
- **Setting the Body Content**—Refers to a phase in which the `setJspBody()` method is called by the container, if a tag is not empty bodied. This method is invoked to set the body of the tag, as a `JspFragment` object. If a tag is empty in the page, the `setJspBody()` method is not called at all. The tag handler calls the `invoke()` method on the `JspFragment` object to evaluate the body multiple times.
- **Executing doTag() method**—Refers to a phase in which evaluation of all the logic, iteration, and body content of a tag occurs by calling the `doTag()` method.

Implementing Simple Tag Handler

Let's now create an application, simpleTag, to understand the use of simple tags in a `JSP` page. Create a tag handler class, a TLD file, and a JSP file in the `simpleTag` application. The following are the broad-level steps to develop the `simpleTag` application:

- Creating a tag handler for `simpleTag` application
- Creating a TLD file for `simpleTag` application
- Creating a JSP file for `simpleTag` application
- Configuring the `web.xml` file for `simpleTag` application
- Defining the directory structure of the `simpleTag` application

- Packaging, running, and deploying the simpleTag application

Let's discuss each of these steps in detail.

Creating the Tag Handler for simpleTag Application

The tag handler for the simpleTag application contains the functionality of the tag coded inside it. The code for the tag handler is provided in the CustomMessage1.java file, as shown in Listing 8.6 (you can find this file on the CD in the code\JavaEE\Chapter8\simpleTag\src\com\kogent\tags folder):

Listing 8.6: Showing the CustomMessage1.java File

```
package com.kogent.tags;
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;
import java.util.Date;
public class CustomMessage1 extends SimpleTagSupport {
  public void doTag() throws JspException,IOException {
  JspContext context=getJspContext();
  JspWriter Out=context.getOut();
  Out.println("Welcome!!! You are visting this Web page on"+new Date());
  }
}
```

In Listing 8.6, the doTag() method of the SimpleTagSupport class creates the instance of the JspWriter object to generate a response. The println() method is used to print the message showing the current date and time on the browser. The object of the Date class is supplied as a parameter to the println() method to show the current date and time on the browser. Save the file in the C:/JavaEE/Chapter8/simpleTag/src/com/kogent/tags folder.

Creating TLD for the simpleTag Application

You can create a TLD file for the simpleTag application to make an entry of the custom tag. The code for TLD, CustomTags.tld is shown in Listing 8.7 (you can find this file on the CD in the code\JavaEE\Chapter8\simpleTag\src\com\kogent\tags folder):

Listing 8.7: Showing the CustomTags.tld File

```
<?xml version="1.0" ?>
<taglib version="2.1" 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/web-   jsptaglibrary_2_1.xsd">
  <jsp-version>2.1</jsp-version>
  <tlib-version>1.1</tlib-version>
  <short-name>tag</short-name>
  <tag>
        <name>CustomMessage1</name>
        <tag-class>com.kogent.tags.CustomMessage1</tag-class>
        <body-content>empty</body-content>
  </tag>
</taglib>
```

In Listing 8.7, a TLD named, CustomTags.tld, is created to reuse the functionality encapsulated in the CustomMessage1.java file. The CustomTags.tld file uses the tag named <tag>, which is the parent tag containing child tags, such as <name>, <tag-class>, and <body-content>. The <name> tag specifies the name of the custom tag. The <tag-class> tag specifies the tag handler class, which holds the functionality of the custom tag and the <body-content> tag that specifies whether or not the custom tag is empty.

The <body-content> tag is set to empty, as the tag used in Listing 8.7 is empty. Save this file in the C:/JavaEE/Chapter8/simpleTag/WEB-INF folder.

Creating the JSP File for the simpleTag Application

The JSP file for the simpleTag application is created to use the custom tag by importing the tag library using the taglib directive, as shown in Listing 8.8 (you can find the test.jsp file on the CD in the code\JavaEE\Chapter8\simpleTag folder):

Listing 8.8: Showing the `test.jsp` File

```
<%@ taglib uri="/WEB-INF/CustomTags.tld" prefix="tag"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
        <link rel="stylesheet" type="text/css" href="mystyle.css">
  </head>
  <body>
        <tag:CustomMessage1/><br>
  </body>
</html>
```

Save the `test.jsp` file in the `C:/JavaEE/Chapter8/simpleTag` folder.

Configuring the web.xml File for the simpleTag Application

Now, configure the `web.xml` file for the `simpleTag` application and set the test.jsp file as the welcome file, as shown in Listing 8.9 (you can find this file on the CD in the `code\JavaEE\Chapter8\simpleTag\WEB-INF` folder):

Listing 8.9: Showing the `web.xml` File for `simpleTag` Application

```
<?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">
  <welcome-file-list>
        <welcome-file>test.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Save the `web.xml` file in the `C:/JavaEE/Chapter8/simpleTag/WEB-INF` folder. When the JSP container finds a custom tag in the JSP page, then it searches for the CustomTags.tld file in the WEB-INF directory and executes the tag.

Defining the Directory Structure of the simpleTag Application

After all the files are created and the `simpleTag` application is configured, you need to define the directory structure of the `simpleTag` application. Figure 8.5 displays the directory structure of the `simpleTag` application:

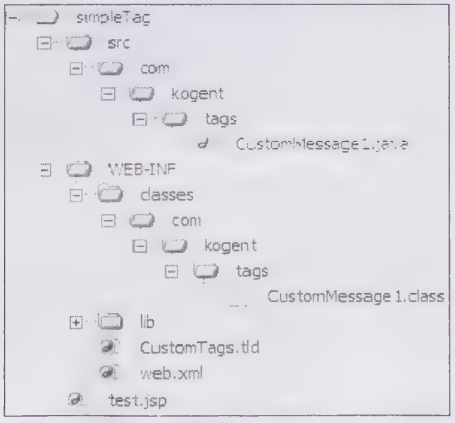


Figure 8.5: Displaying the Directory Structure of the simpleTag Application

Packaging, Running, and Deploying the simpleTag Application

You need to create the `simpleTag.war` file to package the `simpleTag` application and deploy this WAR file as a Web application on the Glassfish V3 application server. Browse `http://localhost:8080/simpleTag` URL to run the application, as shown in Figure 8.6:

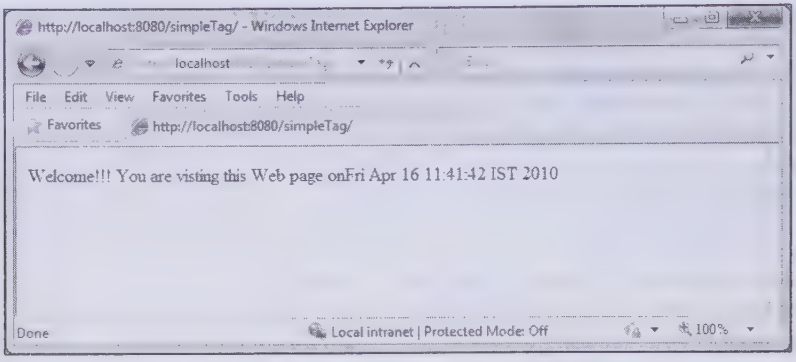


Figure 8.6: Showing a Simple Tag in a JSP Page

In Figure 8.6, the current time is displayed to the user accessing the JSP page with the help of a simple tag, named `CustomMessage1.java`. Now, let's discuss about JSP fragments in the next section.

Working with JSP Fragments

JSP fragments are portions of the JSP code in the `JspFragment` object, which can be invoked multiple times in an application. These fragments are configured either by specifying the `<jsp:attribute>` standard action in a tag file as a `fragment` type or as a body content of a simple tag.

The `JspFragment` instance is associated with the `JspContext` object of the current JSP page before being passed to a tag handler. This instance is also associated with the parent or simple tag instance, as whenever there is any custom action invoked from within a `Jsp fragment`, the `setParent()` method can be called with an appropriate value. The `invoke()` method of the `JspFragment` class executes the body of the `Jsp fragment` and directs the output to either the `Writer` or the `JspWriter` object returned by the `getOut()` method of the `JspContext` object associated with the fragment. The implementation of these methods might lead to the `JspException` exception, which must be handled by the tag handler itself. The following subsections describe the creation and invocation of a JSP fragment.

Creating a JSP Fragment

JSP fragments are created as an instance of a class that implements the `JspFragment` abstract class. This instance should be configured in such a manner that it produces the contents of the body of the fragment, when invoked. If the fragment specifies the body of a `<jsp:attribute>` action, then the fragment needs to evaluate the body of the `<jsp:attribute>` action each time the fragment is invoked. However, if the fragment defines the body of a simple tag, the behaviour of the fragment changes depending on the body content of the tag, whenever it is invoked. For example, if the body content is dependent on a tag, then the fragment must echo the contents of the body each time it is invoked. On the other hand, if the body content is scriptless, then the fragment must evaluate the body each time it is invoked.

The `JspFragment` instance is passed as a reference for the current JspContext. If the fragment requires to invoke a tag handler, then the value passed as reference must be used while calling the `setJspContext()` method. The `JspFragment` instance is associated with an instance of the tag handler of the nearest enclosing tag invocation. In case there is no enclosing tag, the null value is passed. Whenever the fragment invokes a tag handler, the fragment must use the passed value while calling the `setParent()` method.

Invoking a JSP Fragment

When the JSP fragment is created, it is passed to a tag handler for invocation. JSP fragments can be invoked either by using standard actions, such as`<jsp:invoke>` and `<jsp:doBody>` or by using a tag handler written in Java. The JSP fragment has a bean property of type, JSPFragment, and is passed to the tag handler. These fragments can be invoked in the tag handler by calling the `invoke()` method. The tag handler is responsible for setting the values of all the declared AT_BEGIN and NESTED variables in the calling page or in the JspContext of a tag.

If the <jsp:invoke> or <jsp:doBody> actions are used to invoke the JSP fragment, then you should specify a value for the var attribute and create the Writer object to expose the invocation as a String object. If the value for the varReader attribute is specified, a custom Writer object is created to expose the result of the invocation of the JSP fragment as a Reader object. During the invocation of the JSPFragment object, the JSP container:

- Specifies that before the execution of the fragment body, if a non-null value is assigned to the Writer object, the values of JspContext.getOut() method and implicit Out object must be updated to send the output to the Writer object. The container must call the pushBody(writer) method on the current JspContext object. In this method, the instance of the Writer class is passed to the fragment upon invocation, and the body of the fragment is evaluated.
- Specifies that if a classic tag handler is invoked and returns the SKIP_PAGE field as well as throws the SkipPageException exception, the JspFragment must throw the SkipPageException exception implying that the calling page needs to be skipped for evaluation.
- Specifies that after a fragment is evaluated, an exception is thrown and the value of the JspContext.getOut() method must be restored by calling the popBody() method on the current JspContext object.
- Specifies that if the <jsp:invoke> or <jsp:doBody> action is used to invoke a fragment and the value for the var or varReader attribute is provided, then a scoped variable with a name similar to the value of the var or varReader attribute is created (or modified) in the page scope. In addition, this value is set to a Reader object that can invoke the fragment automatically.
- Specifies that when the evaluation of a tag is completed, then the tag discards the fragment instance associated with it so that it can be reused by the JSP container.

Exploring the JspFragment Class

The JspFragment class encapsulates an object with a portion of the JSP code that can be invoked multiple times in an application. JSP fragments can encapsulate these portions of the JSP code either as a body of a simple tag or as the <jsp:attribute> standard action. The <jsp:attribute> standard action specifies a fragment attribute. The definition of the JSP fragment contains template text and JSP action elements, and does not contain any scriptlets or scriptlet expressions. The JspFragment abstract class is an implementation of container at the translation time and is capable of executing the defined fragment.

A tag handler can invoke the fragment zero or more times or pass the fragment to other tags for the purpose of communicating the values to/from a JspFragment object. The tag handlers store/retrieve values from/in the JspContext associated with the fragment.

Table 8.28 describes the methods of the JspFragment class:

Table 8.28: Describing the Methods of the JspFragment Class

Method	Description
getJspContext()	Returns the JspContext object that is bound to the current JspFragment instance during invocation
invoke(java.io.Writer out)	Executes the fragment and directs all output to the specified JSPWriter object returned by the getOut() method of the JspContext class associated with the fragment

Now, let's discuss tag files that allow you to create custom tags by using the JSP syntax.

Working with Tag Files

Tag files allow you to build custom tags by using the JSP syntax. These files are translated into Java code automatically by the JSP container similar to the process of producing Java servlets from JSP pages. The tag files hide the complexity of building custom JSP tag libraries.

Tag files are imported in JSP pages by using the following syntax:

```
<%@ taglib prefix=" " tagdir=" " %>
```

In the preceding syntax, tagdir specifies the path of the tag file and the <prefix:tagFileName> pattern is used to invoke tag files from JSP pages.

A tag file should have the .tag file extension so that it can be recognized by the JSP container. You can store the tag file in the /WEB-INF/tags directory of a Web application or in the /WEB-INF directory. The syntax used in the tag files is similar to that of the JSP pages. However, there are few differences in the tags used in the tag files and JSP pages. For example, the <%@taglib%> directive used in tag files is equivalent to the <%@page%> directive used in JSP pages.

Instead of declaring attributes and variables in a separate .tld file, the tag files use the <%@attribute%> and <%@variable%> directives to create attributes and variables. When a JSP page invokes a tag file, the custom tag that may have a body (between the <prefix:tagFileName> and </prefix:tagFileName> tags) is executed by the tag file with the help of the <jsp:doBody> standard action. The following tasks can be performed by using a tag file:

- Handling dynamic attributes in tag files
- Exporting variables from a tag file to a JSP page
- Using attributes to provide names for variables
- Invoking JSP fragments from tag files

Let's discuss these tasks one by one in detail.

Handling Dynamic Attributes in Tag Files

The JSP page invokes a tag file that accepts dynamic attributes by using the <shortname:tagfile> tag. A tag file accepts only declared attributes, by default. To support dynamic attributes, a tag file must use the <%@tag dynamic-attributes="varName"%> expression, which provides the name of the variable that holds all the dynamic attributes in the java.util.Map interface. The tag file iterates over the items of variables with the <c:forEach> action to retrieve the name and value of each attribute with the help of the ${variable.key} and ${variable.value} expressions. The tag file uses the <jsp:element> standard action to generate HTML elements whose names are obtained at runtime.

Exporting Variables from a Tag File to a JSP Page

A tag file exports the scopes of its local variables to the invoking JSP page by using the <%@variable%> directive. As each tag file has its own page scope, the local variables of the invoking page are not accessible in the invoked tag file and vice versa. The JSP page may use tag attributes and global variables to pass parameters to the invoked tag. The JSP container, for each attribute, creates a local variable in the page scope of the tag file that allows you to get an attribute's value by using the ${attributeName} constructs. Therefore, tag attributes act as parameters passed by value, while global variables act as parameters passed by reference.

The name of a variable can be specified in a tag file using <%@variable name-given="..."%>, or in a JSP page. The type of a variable can be indicated with <%@variable variable-class="..."%>.

After declaring a variable in a tag file with the <%@variable%> directive, you have to set its value by using the <c:set> tag. The JSP container creates another variable with the given name in the JSP page, and the second variable is initialized with the value of the corresponding variable from the tag file. Changing the value of the variable from the JSP page does not affect the corresponding variable from the tag file. However, the JSP container, updates the variable from the JSP page depending on the scope attribute from <%@variable%>, which can be either AT_BEGIN, NESTED or AT_END.

If scope is AT_END, the variable from the JSP page is initialized with the value of the tag file variable after the execution of the tag file.

If scope is AT_BEGIN, the variable from the JSP page is updated before the <jsp:doBody> and <jsp:invoke> tags and at the end of the execution of a tag file. If scope is NESTED, the JSP variable is updated only before the <jsp:doBody> and <jsp:invoke> tags with the value of the tag file variable. After the execution of the tag file, in the NESTED case, the JSP variable is restored to the value it had before the invocation of the tag file.

Using Attributes to Provide Names for Variables

Attributes are also used to provide names for variables in a tag file. Let's consider an example of the `variableAttr.tag` tag file defining a variable whose name is provided by a JSP page (`variableAttr.jsp`) as the value of a tag attribute (v). The tag file creates a variable by defining an alias (a) using `<%@variable name-from-attribute="v" ... alias="a" ...%>`.

The `variableAttr.tag` file sets the value of the variable with `<c:set var="a" value="..."/>`, outputs the name of the variable with `${v}`, and outputs the value of the variable with `${a}`. The `variableAttr.jsp` page invokes the tag file with `<demo:variableAttr v="x"/>` and works with the variable named x. The JSP container creates the x variable automatically, setting it to the value of a, which is `123`.

The code for the `variableAttr.tag` file is shown in the following code snippet:

```
<%@ attribute name="v" required="true" %>
<%@ variable name-from-attribute="v"
  variable-class="java.lang.Long"
  alias="a" scope="AT_END" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:set var="a" value="${123}"/>
<p> TAG: ${v} = ${a} </p>
```

The code for the `variableAttr.jsp` file is shown in the following code snippet:

```
<%@ taglib prefix="demo" tagdir="/technology/WEB-INF/tags/demo" %>
<demo:variableAttr v="x"/>
<p> JSP: x = ${x} </p>
```

Let's now learn how to invoke JSP fragments from tag files.

Invoking JSP Fragments from Tag Files

The tag files support JspFragment attributes, which are used to invoke Jsp fragments. To understand the invocation of JSP fragments, let's create a tag file, `fragmentAttr.tag` and a JSP page, `fragmentAttr.jsp`. The JSP page invokes the `fragmentAttr.tag` tag file with the help of the `<demo:fragmentAttr>` tag that provides two simple attributes (`attr1` and `attr2`) and a fragment attribute (template). All three attributes are declared in the tag file with `<%@attribute%>`. The template attribute is declared as a fragment attribute by using `fragment="true"` along with `<%@attribute%>`.

The tag file declares a nested variable (`data`) with `<%@variable%>`. The value of the variable is set by the tag file with `<c:set>`.

The code of the `fragmentAttr.tag` file is shown in the following code snippet:

```
<%@ attribute name="template" fragment="true" %>
<%@ attribute name="attr1" %>
<%@ attribute name="attr2" %>
<%@ variable name-given="data" scope="NESTED" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:set var="data" value="${attr1}"/>
<jsp:invoke fragment="template"/>
<c:set var="data" value="${attr1} - ${attr2}"/>
<jsp:doBody/>
<c:set var="data" value="${attr2}"/>
<jsp:invoke fragment="template"/>
<c:set var="data" value="${attr2} - ${attr1}"/>
<jsp:doBody/>
```

The code of the `fragmentAttr.jsp` file is shown in the following code snippet:

```
<%@ taglib prefix="demo" tagdir="/technology/WEB-INF/tags/demo" %>
<demo:fragmentAttr attr1="value1" attr2="value2">
  <jsp:attribute name="template">
      <p> Template: ${data} </p>
  </jsp:attribute>
  <jsp:body> <p> Body Content: ${data} </p> </jsp:body>
</demo:fragmentAttr>
```

In the preceding code snippet, the `<jsp:attribute>` and `<jsp:body>` standard actions are used to define the two JSP fragments , which are invoked from the tag file with `<jsp:invoke>` and `<jsp:doBody>`, respectively.

Now, let's recapitulate the topics covered in this chapter.

Summary

In this chapter, you have learned about the need and features of JSP tag extensions. Next, you have learned about the elements of JSP tag extensions and tag extension API, including classes and interfaces that are used to create the custom tags. The applications for simple tag handlers and classic tag handlers are also created in the chapter. Next, the chapter has explored the concept of fragments used in a JSP page. Finally, you have learned about tag files that are used to create custom tags using the JSP syntax.

In the next chapter, you learn about JavaServer Pages Standard Tag Library (JSTL), which provides custom tag libraries containing tags for conditional structures, iteration, internationalization, and HTTP manipulation, XML, and SQL statements.

Quick Revise

Q1. **..... is a valid taglib directive.**

A. **<%taglib uri="/stats" prefix="stats"%>**
B. **<%@ taglib uri="/stats" prefix="stats"%>**
C. **<%!taglib uri="/stats" prefix="stats"%>**
D. **<%taglib uri="/stats" prefix="stats"%>**
E. **<%@ taglib name="/stats" prefix="stats"%>**

Ans. B

Q2. **Which of the following elements are the valid `<taglib>` elements in the `web.xml file`?**

A. **uri**
B. **taglib-uri**
C. **tagliburi**
D. **tag-uri**
E. **taglib-location**

Ans. B, E

Q3. **The..... interface is implemented by a tag handler to provide support for dynamic attributes.**

A. **DynamicAttributes**
B. **BodyTag**
C. **IterationTag**
D. **JspTag**

Ans. A

Q4. **Which of the following are valid return types of the `doAfterBody()` method of the Iteration-Tag interface?**

A. **EVAL_BODY_INCLUDE**
B. **`SKIP_BODY`**
C. **EVAL_PAGE**
D. **SKIP_PAGE**

Ans. B

Q5. **The .....class validates a JSP page during translation-time and performs the validation on the xml representation of the JSP page.**

A. **TagLibraryValidator**
B. **TagData**
C. **FunctionInfo**
D. **TagVariableInfo**

Ans. A

Q6. **Which of the following are the life cycle phases of classic tag handlers?**

A. **Tag handler instance creation**
B. **Context setting**
C. **Executing doEndTag() method**
D. **Parent setting**
E. **All of the above**

Ans. E

Q7. **What is a tag handler?**

Ans. A tag handler is a Java class where a custom tag is defined. The JSP container invokes the tag handler object to evaluate a custom tag during the execution of a JSP page.

Q8. **What is the main difference between classic and simple tag handler?**

Ans. Classic tag handlers are cached and reused by container but simple tag handlers are not cached and reused by the containers.

Q9. **Which method of the simple tag handler is used to process the tag?**

Ans. doTag()

9

Implementing JavaServer Pages Standard Tag Library 1.2

JavaServer Pages Standard Tag Library (JSTL) was introduced as a component of the Java EE platform to enable Web designers to easily create JavaServer Pages (JSP) pages. Before the introduction of JSTL, Web designers had to extensively use scriptlets while creating JSP pages. Moreover, they had to spend a lot of time in coding custom tags. JSTL provides a solution to these problems with the help of tag libraries, which are capable of implementing common tasks, such as performing iteration, evaluating conditions, ensuring database access, processing Extensible Markup Language (XML) documents, and implementing internationalization, in JSP pages. JSTL also provides support for the unified Expression Language (EL). JSTL provides tag libraries such as the core tag library, the XML tag library, the internationalization tag library, the Structured Query Language (SQL) tag library, and the functions tag library. These tag libraries contain predefined sets of tags to use in JSP pages.

This chapter begins by providing an introduction to JSTL. Then it explains the tag libraries available in JSTL, including the core tag library, the XML tag library, the internationalization tag library, and the SQL tag library, and also explains how to implement these tag libraries.

Let's begin by introducing JSTL.

Introducing JSTL

Initially, Web designers used scriptlets in JSP pages to generate dynamic content. This resulted in readability issues and also made it difficult to maintain the JSP page. Custom tags were introduced to overcome the problems faced in using scriptlets. Although custom tags proved to be a better choice than scriptlets, they had certain limitations too. Web designers had to spend a lot of time in coding, packaging, and testing these tags before using them. This meant that Web designers were often left with little time to concentrate on the designing of Web pages.

The introduction of JSTL has helped Web designers overcome the shortcomings of custom tags, by encapsulating the common functionalities that the Web designer may need to develop Web pages. These functionalities included the use of tag libraries, such as core, SQL, and XML. JSTL is introduced particularly for those Web designers who are not well versed with Java programming. JSTL 1.2, introduced in the Java EE 5 platform, aligns with the unified EL. Note that the unified EL helps JavaServer Faces (JSF) to use JSTL tags. The same version of JSTL, i.e., JSTL 1.2, is used in the Java EE 6 platform as well.

Let's now learn about the features of JSTL.

Explaining the Features of JSTL

JSTL aims to provide an easy way to maintain JSP pages. The use of tags defined in JSTL has simplified the task of the designers to create Web pages. They can now simply use a tag related to the task that they need to implement in a JSP page. The main features of JSTL are as follows:

- Provides support for conditional processing and Uniform Resource Locator (URL)-related actions to process URL resources in a JSP page. You can also use the JSTL core tag library that provides iterator tags used to easily iterate through a collection of objects.
- Provides the XML tag library, which helps you to manipulate XML documents and perform actions related to conditional and iteration processing on parsed XML documents.
- Enables Web applications to be accessed globally by providing the internationalization tag library. Internationalization means that an application can be created to adapt to various locales so that people of different regions can access the application in their native languages. The internationalization tag library makes the implementation of localization in an application easy, fast, and effective.
- Enables interaction with relational databases by using various SQL commands. Web applications require databases to store information required for the application, which can be manipulated by using the SQL tag library provided by JSTL.
- Provides a series of functions to perform manipulations, such as checking whether an input String contains the substring specified as a parameter to a function, or returning the number of items in a collection, or the number of characters in a String. These functions can be used in an EL expression and are provided by the functions tag library.

Let's now explore JSTL tag libraries.

Exploring the Tag Libraries in JSTL

A tag library provides a number of predefined actions that bind functionalities to a specific JSP page. JSTL provides tag libraries that include a wide range of actions to perform common tasks. For example, if you want to access data from database, you can use SQL tag library in your application. JSTL is a standard tag library that is composed of five tag libraries. Each of these tag libraries represents separate functional areas and is used with a prefix. Table 9.1 describes the tag libraries available in JSTL:

Table 9.1: Tag Libraries in JSTL, with their Uniform Resource Identifier (URI) and Tag Prefix

Name of the Tag Library	Function	URI	Prefix
Core tag library	Variable support	http://java.sun.com/jsp/jstl/core	c
	Flow control		
	Iterator		
	URL management		
	Miscellaneous		
XML tag library	Core	http://java.sun.com/jsp/jstl/xml	x
	Flow control		
	Transformation		
Internationalization tag library	Locale	http://java.sun.com/jsp/jstl/fmt	fmt
	Message formatting		
	Number and date formatting		
SQL tag library	Database manipulation	http://java.sun.com/jsp/jstl/sql	sql
Functions tag library	Collection length	http://java.sun.com/jsp/jstl/functions	fn
	String manipulation		

After exploring the different tag libraries of JSTL, let's discuss these tag libraries in detail in the following sections.

Working with the Core Tag Library

The core tag library contains tags that are related to variables and flow control support in an application. For example, if you want to set the value of a variable, you can use the `set` tag of the core tag library. This library also provides a generic way to access URL-based resources. In addition, the core tag library specifies the content that can be included or processed within a JSP page.

Now, let's explore the different tags of the core tag library.

Exploring the Tags in the Core Tag Library

Let's discuss the tags available in the core tag library, based on their functionalities. Table 9.2 categorizes the tags of the core tag library based on their functionalities:

Table 9.2: Tags of the Core Tag Library

Function	Tag
Variable support	remove
	set

Table 9.2: Tags of the Core Tag Library	
Function	**Tag**
Flow control	choose when otherwise if
Iterator	forEach forTokens
URL management	import param redirect param url param
Miscellaneous	catch out

Next, we describe the functions of the core tag library in detail.

Describing the Variable Support Tags

JSTL provides the following tags for variable support in an application:

- The <c:set> tag
- The <c:remove> tag

The <c:set> Tag

The <c:set> tag sets the property and value of an EL variable in a JSP scope (i.e., page, request, session, or application). This tag also creates an EL variable, if the variable does not already exist.

The syntax of the <c:set> tag can be used in the following two ways:

- The <c:set> tag without body
- The <c:set> tag with body

The <c:set> Tag without Body

The following code snippet shows the use of the <c:set> tag without a body to set a value for a variable:

```
<c:set var="bookName" scope="session" value="black book:Java EE 6"/>
  <c:out value="${bookName}"/>
```

The preceding code snippet shows a value being set for the bookName variable by using the <c:set> tag. This tag does not contain the body element; however, the value of the bookName variable is set with the help of a value attribute.

The <c:set> Tag with Body

Variables can be set by using the <c:set> tag within the body of another tag, as shown in the following code snippet:

```
<c:set var="bookName">
Black book:Java EE 6
</c:set>
<c:out value="${bookName}"/>
```

In the preceding code snippet, the value for the bookName variable is set in the body of the <c:set> tag.

The <c:remove> Tag

The <c:remove> tag is used to remove a variable from a scope. The following code snippet shows the use the <c:remove> tag:

```
<c:remove var="bookName" scope="session"/>
```

In the preceding code snippet, the <c:remove> tag removes the bookName variable from the session scope.

Describing the Flow Control Tags

Before the introduction of JSTL, coding conditional constructs by using scriptlets was a difficult task. Later, JSTL provided the following tags in the core tag library to control the flow of execution in a JSP page:

- The <c:if> tag
- The <c:choose> tag
- The <c:when> tag
- The <c:otherwise> tag

You should note that the <c:choose>, <c:when>, and <c:otherwise> tags are dependent on each other on the basis of the condition specified in a JSP page. As the syntax of these tags cannot be explained independently, they are described together under the *The <c:choose>, <c:when>, and <c:otherwise> Tags* section.

Let's discuss these tags in the following sections.

The <c:if> Tag

The <c:if> tag allows conditional execution of actions specified in the body of a tag. The actions executed are based on the conditions that are set according to the value of the `test` attribute. The following code snippet shows the use of the <c:if> tag:

```
<c:if test="${(empty user.userid || empty user.password)}">
  Provide User ID and password
</c:if>
```

In the preceding code snippet, the `Provide User ID and password` message is displayed if the user ID or password property is null.

The <c:choose>, <c:when>, and <c:otherwise> Tags

The <c:choose>, <c:when>, and <c:otherwise> tags work similar to the switch statement used in Java. The `<c:choose>` tag provides context to the `<c:when>` and `<c:otherwise>` tags. The `<c:when>` tag has an attribute named `test` to specify a condition. If the specified condition evaluates to true, the JSP container evaluates the body of the `<c:when>` tag. In such a case, no other <c:when> tags below the <c:choose> tag are evaluated and the flow of execution moves to the line after the closing <c:choose> tag. If none of the <c:when> tags evaluates to true, then the <c:otherwise> tag is evaluated.

The following code snippet shows the use of the <c:choose>, <c:when>, and <c:otherwise> tags to produce text, based on the type category of an employee:

```
<c:choose>
<c:when test="${employee.type == 'part-time'}" >
...
</c:when>
<c:when test="${employee.type == 'full-time'}" >
...
</c:when>
<c:when test="${employee.type == 'permanent'}" >
...
</c:when>
<c:when test="${employee.type == 'on contract'}" >
...
</c:when>
<c:otherwise>
...
</c:otherwise>
</c:choose>
```

In the preceding code snippet, the nested <c:when> tags evaluate the condition based on the employee's category. When the test condition of a nested <c:when> tag is evaluated to true, the body of the <c:when> tag is executed. If none of the test conditions is true, the body of the <c:otherwise> tag is executed.

Exploring Iterator Tags

JSTL provides the <c:forEach> tag to iterate over a collection of objects, such as java.util.Collecton, java.util.Map, java.lang.String, java.util.Iterator, and java.util.Enumeration. The <c:forEach> tag also iterates over an array of the java.lang.Object type and an array of primitive data types.

The `java.util.Iterator` and `java.util.Enumeration` collection objects must be used with utmost caution, as these objects are not resettable. Therefore, they must be used within one iteration tag. If an array of a String object contains a list of comma-separated values, such as January, February, March, April, May, then the <c:forEach> tag iterates over it as array of the java.lang.String object. The following code snippet demonstrates the implementation of the `<c:forEach>` tag in a JSP page:

```
<table>
      <c:forEach var="name" items="${names}">
      <tr>
      <td>
            Name>:<c:out value="${name}"/>
      </td>
      </tr>
      </c:forEach>
</table>
```

In the preceding code snippet, the <c:forEach> tag iterates over a collection of the name object, which contains a list of names. The value of each name is retrieved and displayed in a table format.

The <c:forEach> tag can also be used to iterate over an array of primitive data types, as shown in the following code snippet:

```
The list of even numbers between 0 to 50:
   <br/>
         <c:forEach var="i" begin="0" end="50" step="2">
         <c:out value="${i}"/>
   <br/>
   </c:forEach>
```

In preceding code snippet, the `<c:forEach>` tag is used to display a list of even numbers between 0 and 50. In this case, the `begin` attribute is used to set the initial value for the `i` variable. On the other hand, the `end` attribute is used to set a value until which the iteration will continue. In the preceding code snippet, the 0 value is assigned to the `begin` attribute and 50 value is assigned to the `end` attribute.

Exploring URL Tags

JSTL provides various URL tags to implement common URL-related actions. Some of these actions include importing resources, redirecting Hypertext Transfer Protocol (HTTP) responses, URL rewriting, and specifying request parameters.

The `<jsp:include>` tag helps to include static and dynamic resources from the same context as that of the current JSP page. This tag cannot access resources that are outside a Web application as a resource conflict may occur if a resource is being accessed by various Web applications. For this reason, JSTL provides the `<c:import>` tag, which is more powerful than the `<jsp:include>` tag and helps to reduce the possibility of resource conflict among various Web applications. The syntax to use the <c:import> tag is as follows:

```
<c:import url = "jsp page">
</c:import>
```

Exploring Miscellaneous Tags

The core tag library provides some other tags, known as miscellaneous tags, which are used to perform various tasks, such as implementing the try and catch block. An important miscellaneous tag is the <c:catch> tag, which helps a JSP page to handle exceptions. Code that can raise exceptions is kept within the body of the <c:catch> tag. This tag can catch exceptions that are subclasses of the java.lang.Throwable class, which includes all Java exceptions.

Instead of handling an exception by using an error page, you can use the <c:catch> tag. This tag prevents the invocation of an error page in case an exception is not handled in a JSP page.

The following code snippet shows the use of the <c:catch> tag:

```
<c:catch var ="catchException">
<% int x = 2/0;%> /--raises exception
</c:catch>
The exception is: ${catchException.message}
```

In the preceding code snippet, an integer is divided by zero, which raises an exception that is caught by the <c:catch> tag.

After having a basic conceptual knowledge of the core tag library, let's learn to use the tags of this library in a Web application.

Using the Core Tag Library in the coreTagApp Application

In this section, let's create a Web application, `coreTagApp`, to demonstrate the use of the core tag library. In the coreTagApp Web application, the core tag library is used to implement the functionality of searching a Java book based on its audience level. Moreover, you can display the details of a Java book with the help of the details JSP page created in the Web application. The broad-level steps to create the `coreTagApp` Web application are as follows:

- Create the JSP pages
- Configure the application
- Define the directory structure of the application
- Package, deploy, and run the application

Now let's perform these steps in the following sections.

Creating the JSP Pages

The coreTagApp application contains the following three JSP pages:

- **The select JSP page**—Helps to search books based on audience level
- **The details JSP page**—Displays the details of the books
- **The menu JSP page**—Provides parameters named option1 and option2, which accept different values based on the condition specified in the details.jsp page.

The select JSP page is the home page of the `coreTagApp` application. This page contains a combo box named `level`, which contains the Beginner, Intermediate, and Advance values. These values represent the audience level, which is used as a parameter to search a Java book. The select JSP page also contains a button, `Submit`, which a user clicks after selecting an audience level. Listing 9.1 provides the code for the select.jsp file (you can find the select.jsp file on the CD in the `code\JavaEE\Chapter9\coreTagApp` folder):

Listing 9.1: Showing the Code for the select JSP Page

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
        <title>welcome  to the world of Java books.....</title>
        <link rel="stylesheet" type="text/css" href="mystyle.css">
  </head>
<body>
  <c:set var="message" value="Welcome to JSTL!" scope="session" />
  <h2 align="center"><c:out value="${message}"/></h2>
  <form action="details.jsp" method="post">
  <table align="center">
<tr><td colspan=2 align="center"><h4>welcome to the world of Java books</h4></td></tr>
  <tr><td>A place where you can get information on Java books</td></tr>
  <tr><td> select audience level <select name="level">
        <option>Beginner</option>
        <option>Intermediate</option>
        <option>Advance</option>
  </select>
  </td></tr>
```

```
    <tr><td colspan="2"><input type="submit" value="Submit"></td></tr>
    </table>
  </form>
  </body>
  </html>
```

In Listing 9.1, the <c:set> core tag is used to set the value for a variable, called `message`, through an attribute called `value`. The <c:out> tag is used to display the value of the `message` variable by using the `value` attribute. This attribute is assigned the value of the `message` variable used as an expression.

After a user has selected an appropriate audience level and clicked the `Submit` button, the request is forwarded to another JSP page, named `details`. Listing 9.2 shows the code for the details.jsp file (you can find the details.jsp file on the CD in the `code\JavaEE\Chapter9\coreTagApp` folder):

Listing 9.2: Showing the Code for the details.jsp File

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
        <title>book detail:</title>
        <link rel="stylesheet" type="text/css" href="mystyle.css">
  </head>
<body>
  <c:set var="level" value="${param.level}" scope="session" />
  <c:choose>
        <c:when test="${level eq 'Beginner'}">
        <b> Books for beginner level audience are as follows:</b>
        </c:when>
  <c:when test="${level eq 'Intermediate'}">
  <b> Books for intermediate level audience are as follows:</b>
  </c:when>
  <c:when test="${level eq 'Advance'}">
  <b>Books for advance level audience are as follows: </b>
  </c:when>
  <c:otherwise>
  </c:otherwise>
  </c:choose>
  <c:if test="${level eq 'Beginner'}">
        <c:import url="menu.jsp">
        <c:param name="option1" value="EJB 3.0 in Simple steps"/>
        <c:param name="option2" value=" Beginning Java 2"/>
  </c:import>
  </c:if>
  <c:if test="${level eq 'Intermediate'}">
        <c:import url="menu.jsp">
        <c:param name="option1" value="Thinking in Java"/>
        <c:param name="option2" value="Java 2 platform unleashed"/>
        </c:import>
  </c:if>
  <c:if test="${level eq 'Advance'}">
        <c:import url="menu.jsp">
<c:param name="option1" value="Java Server programming J2EE 1.4:Black Book"/>
<c:param name="option2" value="Java Server programming JavaEE 5:Black Book"/>
  </c:import>
  </c:if>
  <c:url value="select.jsp" var="backurl"/>
  <br><br><a href="${backurl}">Back</a>
</body>
</html>
```

In Listing 9.2, the audience level selected in the select JSP page is set in the `level` variable with the help of the <c:set> tag. The `level` variable is checked against a condition specified in the `test` attribute of multiple <c:if> tags. The actions are executed based on the conditions that are set according to the value of the `test` attribute. The <c:import> tags are nested within each of the <c:if> tags to import the menu JSP page. This page contains

parameters named option1 and option2, which are provided different values by using multiple <c:param> tags in the details JSP page. The code for the menu.jsp file is shown in Listing 9.3 (you can find the menu.jsp file on the CD in the code\JavaEE\Chapter9\coreTagApp folder):

Listing 9.3: Showing the Code for the menu.jsp File

```
<hr>
  <table align="center" width="300" cellspacing="3" >
  <tr align="center" bgcolor="#fee9c2" height=20>
        <td>${param.option1}</td>
        <td>${param.option2}</td>
  </tr>
</table>
<hr>
```

Listing 9.3 shows parameters named option1 and option2, which accept different values based on the condition specified in the details JSP page.

Now, after creating the required JSP pages of the coreTagApp application, let's configure the application.

Configuring the coreTagApp Application

The coreTagApp application is configured by using the web.xml file. Listing 9.4 provides the code for the web.xml file (you can find the web.xml file on the CD in the code\JavaEE\Chapter9\coreTagApp\WEB-INF folder):

Listing 9.4: Showing the Code for the web.xml File for the coreTagApp Application

```
<?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">
  <welcome-file-list>
  <welcome-file>select.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

In Listing 9.4, the select JSP page is mapped as the welcome file. Therefore, the select JSP page is displayed as the home page of the coreTagApp Web application.

Now, let's define the directory structure of the coreTagApp application.

Defining the Directory Structure of the coreTagApp Application

After creating the JSP pages of the coreTagApp application and configuring the application, you need to define the directory structure of the application. Figure 9.1 displays the directory structure of the coreTagApp application:

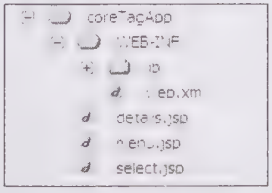


Figure 9.1: Showing the Directory Structure of the coreTagApp Application

Create a directory structure as shown in Figure 9.1, and arrange all the files of the application accordingly. Now, the application is ready to be packaged, deployed, and run.

Packaging, Deploying, and Running the coreTagApp Application

Perform the follow these steps to package, deploy, and run the coreTagApp application:

1. Create the coreTagApp.war file and deploy it on the GlassFish V3 application server.

2. Run the coreTagApp application by using the `http://localhost:8080/coreTagApp` URL. The `select` home page of the application appears, as shown in Figure 9.2:

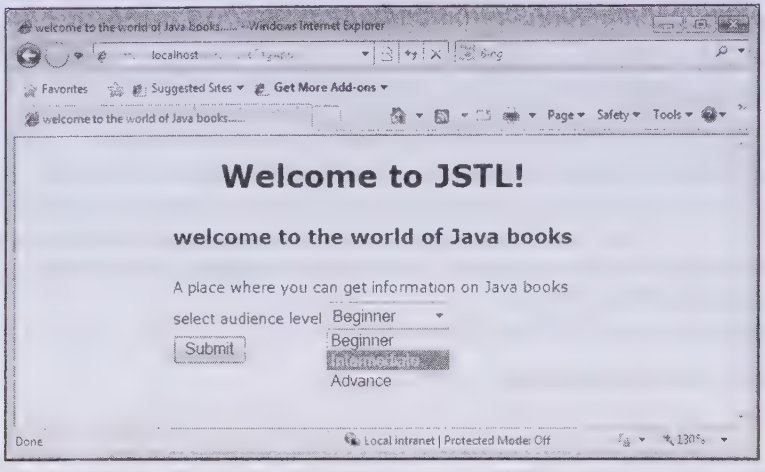


Figure 9.2: Showing the select JSP Page

3. Select the audience level and click the Submit button (Figure 9.2).

The details JSP page appears, which retrieves the value of the `level` parameter by using the param.level expression. This parameter is then used to dynamically retrieve the required books, based on the selected audience level. The `<c:if>` tag is used to evaluate the `level` parameter and set the name of the required books by importing the menu JSP page.

The details JSP page is shown in Figure 9.3:

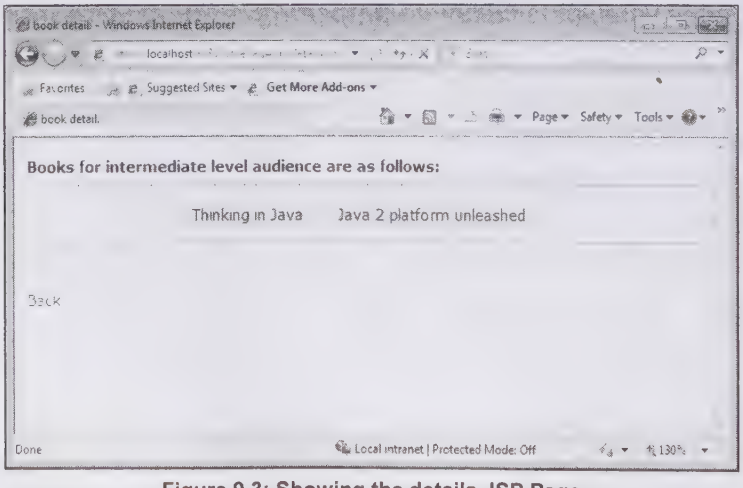


Figure 9.3: Showing the details JSP Page

Let's now move ahead and discuss the XML tag library.

Working with the XML Tag Library

JSTL provides XML tags in a JSP page to manipulate XML documents. XML tags are similar to the tags provided in the core tag library. However, what differentiates XML tags and makes them more powerful than the core library tags is their support for XML Path (XPath) expressions. XPath is a language used to extract specific portions of an XML document to manipulate the data in these portions. In JSTL XML tags, the XPath expressions are assigned to the `select` attribute to select portions of XML data streams. XPath is used as a language for the

select attribute only. This implies that the value specified for the `select` attribute is evaluated by using the XPath language, whereas, for all other attributes, the values are evaluated by using the rules associated with JSP EL.

Other than the standard XPath syntax, JSTL XML tags also support scopes, such as $foo, $param, $header, $cookie, $pageScope, $sessionScope, $requestScope, and $applicationScope, to access data of a Web application within an XPath expression.

Some examples of implementation of scopes by using XPath expressions are as follows:

- $sessionScope:profile: Specifies the profile and name of the session-scoped EL variable
- $initParam:mycom.productId: Specifies the String value of the mycom.productId context parameter

The scope named `$sessionScope` of an XPath expression is used to set the session scope of an EL variable named profile. Similarly, `$initParam`, the scope of an XPath expression, initializes the `mycom.productId` context parameter, with a String value.

Let's now learn about the various tags of the XML tag library.

Exploring the Tags of the XML Tag Library

XML tags can be categorized on the basis of their functionalities, as shown in Table 9.3:

Table 9.3: Tags of the XML Tag Library

Function	Tags

Core	`out`
	`parse`
	`set`

Flow Control	`choose`
	`when`
	`otherwise`
	`forEach`
	`if`

Transformation	`transform` `param`

Now, let's discuss each of the functional categories of the XML tag library.

Describing XML Core Tags

The core XML tags enable you to parse and access XML data efficiently. The XML core tags available in JSTL are as follows:

- The <x:parse> tag
- The<x:out> tag
- The<x:set> tag

Let's now discuss each of these tags in detail.

The <x:parse> Tag

The `<x:parse>` tag is used to parse an XML document and save the result to a variable or the `varDom` attribute. The `varDom` attribute contains a `String` value that is the name of a scoped variable. After the XML document is parsed, it can be used for manipulation by an XPath expression.

The syntax of the `<x:parse>` tag can be used in the following two ways:

- The <x:parse> tag without body
- The <x:parse > tag with body

The <x:parse> Tag without Body

An XML document that you want to parse can be specified with the `xml` attribute of the <x:parse> tag.

The syntax to use the <x:parse> tag without a body to parse an XML document by using the xml attribute is as follows:

```
<x:parse xml="XMLDocument"
{var="var" [scope="scope"]|varDom="var" [scopeDom="scope"]}
[systemId="systemId"]
[filter="filter"]/>
```

The following code snippet parses and saves an XML document to the parsed variable:

```
<c:import url="book.xml" var="book"/>
<x:parse xml="${book}" var="parsed"/>
```

The <x:parse> Tag with Body

XML documents can also be parsed by placing the XML content directly inside the <x:parse> tag. The syntax to parse an XML document by using the <x:parse> tag with body is as follows:

```
<x:parse
{var="var" [scope="scope"]|varDom="var" [scopeDom="scope"]}
[systemId="systemId"]
[filter="filter"]>
XML Document to parse
</x:parse>
where scope is {page|request|session|application}
```

Table 9.4 describes the attributes of the <x:parse> tag:

Table 9.4: Attributes of the <x:parse> Tag

Name of the Attribute	Type	Description
xml	String, Reader, javax.xml.transform.Source, or an object exported by <x:parse>, <x:set> or <x:transform>	Specifies the source XML document to be parsed. If a source is exported by the <x:set> tag, the source must correspond to a well-formed XML document.
systemId	String	Specifies the system identifier (URI) for parsing an XML document.
filter	org.xml.sax.XMLFilter	Represents the filter to be applied to the source XML document.
var	String	Specifies the name of the scoped variable exported to the parsed XML document.
scope	String	Specifies the scope for the variable containing the parsed XML document.
varDom	String	Specifies the name of the scoped variable that is exported to the parsed XML document. This scoped variable is of the org.w3c.dom.Document type.
scopeDom	String	Specifies the scope for the varDom variable.

After learning the use of the <x:parse> tag, let's discuss the <x:out> tag.

The <x:out> Tag

The <x:out> tag displays an output of an XPath expression. The syntax of the <x:out> tag is as follows:

```
<x:out select="XPathExpression" [escapeXml="{true|false}"]/>
```

In the preceding syntax, the XPath expression is assigned to the select attribute. The result of the evaluation of the expression is written to the current JSPWriter object, which is the equivalent of using the <%=...%> JSP expression tag or the <c:out> tag.

After an XML document is parsed and saved to a variable, you can use XPath expressions to extract specific elements from the document. The <x:out> tag is used to access specific elements from the parsed XML

document. The following code snippet shows you how to access the title and description elements of a parsed XML document:

```
<x:out select="$parsed/book/title"/>
<x:out select="$parsed/book/description"/>
```

In the preceding code snippet, the <x:out> tag is used to access a subelement, such as title and description of the parent element, book, of the parsed XML document. Table 9.5 describes the attributes of the <x:out> tag:

Table 9.5: Attributes of the <x:out> Tag

Name of the Attribute	Type	Description
escapeXml	String	Specifies a Boolean value used to determine whether or not the characters, such as <, >, and & should be converted into their corresponding character entity code
Select	String	Specifies the XPath expression to be evaluated

Let's now discuss the <x:set> tag.

The <x:set> Tag

The <x:set> tag is used to evaluate an XPath expression. The result of the <x:set> tag expression is stored in a scoped variable.

The syntax to use the <x:set> tag is as follows:

```
<x:set select="XPathExpression"
var="varName" [scope="{page|request|session|application}"]/>
```

In the following code snippet, the <x:set> tag is used to set the value of the `title` element of a parsed XML document to the name variable:

```
<x:set var="name" select="$parsed/book/title"/>
```

Table 9.6 describes the attributes of the <x:set> tag:

Table 9.6: Attributes of the <x:set> Tag

Name of the Attribute	Type	Description
scope	String	Specifies the scope for a variable containing the result of an XPath expression
select	String	Specifies an XPath expression to be evaluated
var	String	Specifies a name of a scoped variable to hold the result of an XPath expression

This concludes the discussion on XML core tags. Now, let's learn about XML flow control tags.

Describing Flow Control Tags

XML flow control tags control the flow of execution based on the result of XPath expressions. These tags are similar to the core tags of the core tag library, except that they apply to XPath expressions. The XML tag library provides the following flow control tags:

- The <x:if> tag
- The <x:choose> tag
- The <x:when> tag
- The <x:otherwise> tag
- The <x:forEach> tag

Let's discuss these tags one by one.

The <x:if> Tag

The <x:if> tag is used to evaluate the expression provided in the `select` attribute. If the specified expression evaluates to true, the body content of the <x:if> tag will be rendered. The syntax of the <x:if> tag is as follows:

```
<x:if select="PathExpression"
[var="varName"] [scope="{page|request|session|application}"]>
body content
</x:if>
```

Table 9.7 describes the attributes of the <x:if> tag:

Table 9.7: Attributes of the <x:if> Tag

Name of the Attribute	Type	Description
select	String	Evaluates the test condition that specifies whether or not the body content of the <x:if> tag should be processed.
var	String	Returns the name of the exported scoped variable after the condition specified in the <x:if> tag is satisfied. This scoped variable is of boolean type.
scope	String	Specifies the scope for the variable containing the result of an XPath expression.

The <x:choose> Tag

The <x:choose> tag is similar to the <c:choose> tag, which processes the body content of the first <x:when> tag whose test condition evaluates to true. If none of the test conditions of multiple <x:when> tags evaluates to true, the body content of the <x:otherwise> tag is processed, if present. The syntax to use the <x:choose> tag is as follows:

```
<x:choose>
body content (<x:when> and <x:otherwise> subtags)
</x:choose>
```

The <x:when> Tag

The purpose of using the <x:when> tag is similar to the <c:when> tag. However, the implementation of these tags varies. As compared to the <c:when> tag, the <x:when> tag does not support the `test` attribute; instead, it uses the `select` attribute, which specifies an XPath expression to be evaluated. The <x:when> tag is used to provide an alternative option within the <x:choose> tag.

The syntax of the <x:when> tag is as follows:

```
<x:when select="XPathExpression">
body content
</x:when>
```

In the preceding syntax, the value specified in the `select` attribute of the <x:when> tag is an XPath expression. This XPath expression returns a Boolean value. If the first <x:when> tag evaluates to true, the JSP container processes the body content of the <x:when> tag and writes the result to the current JSPWriter object. The <x:when> tag provides a single attribute, `select`, which represents a test condition that should be true to process the body content of the <x:when> tag.

The <x:otherwise> Tag

The <x:otherwise> tag is used optionally as the last alternative within the <x:choose> tag. The syntax to use the <x:otherwise> tag is as follows:

```
<x:otherwise>
. . .. .. . ..
</x:otherwise>
```

Within the <x:choose> tag, if none of the nested <x:when> test conditions evaluates to true, the body content of the <x:otherwise> tag is evaluated by the JSP container, and the result is written to the current JSPWriter object.

The following code snippet shows the interaction among the <x:choose>, <x:when>, and <x:otherwise> tags:

```
<x:choose>
<x:when select='book/price>400'> This is a meant for advance
  audience</x:when>                                          level
<x:when select='book/price<300'> This is a meant for intermediate level audience</x:when>
<x:otherwise>This book is meant for beginner level audience </x:otherwise>
</x:choose>
```

In preceding code snippet, the <x:when> tags nested within <x:choose> tag test the condition on the price tag extracted from book.xml. When the test condition of a nested <x:when> tag is evaluated to be true, then the body of the <x:when> tag is executed. If none of the test condition is true, then the body of the <x:otherwise> tag executes.

The <x:forEach> Tag

The <x:forEach> tag iterates over the result of an XPath expression The syntax of the <x:forEach> tag is as follows:

```
<x:forEach [var="varName"] select="XPathExpression">
body content
</x:forEach>
```

Table 9.8 describes the attributes of the <x:forEach> tag:

Table 9.8: Attributes of the <x:forEach> Tag

Name of the Attribute	Type	Description
select	String	Represents an XPath expression to be evaluated.
var	String	Provides a name for the exported scoped variable representing the current item of an iteration. The accessibility of this scoped variable is nested and its type depends on the result of the XPath expression of the select attribute.

Next, we learn about transformation tags.

Describing Transformation Tags

The XML transform tags help transform XML documents by using Extensible Stylesheet (XSLT). The following code snippet shows the use of the <x:transform> tag:

```
<c:import url="http://acme.com/customers" var="xml"/>
<c:import url="/WEB-INF/xslt/customerList.xsl" var="xslt"/>
<x:transform xmlText="${xml}" xsltText="${xslt}"/>
<x:transform source="${xml}" xslt="${xslt}">
</x:transform>
```

In the preceding code snippet, an external document of the XML type (retrieved by an absolute URL) is translated by a local XSLT (context relative path). The result of this translation is written to a JSP page. At times, you may need to use the same XSLT transformation to different source XML documents. To prevent multiple transformations, you can process the transformation XSLT once, and then save the transformer object for successive transformations. JSTL 1.2, which is delivered as a part of the JSP 2.1 specification, allows you to save transformer objects to improve the performance of a JSP page.

The transformations tag category contains the following two tags:

- <x:transform>
- <x:param>

Now, let's discuss each of these tags in detail.

The <x:transform> Tag

The <x:transform> tag applies a transformation to an XML document based on a specified XSLT.

The syntax of the <x:transform> tag can be used in the following two ways:

- The <x:transform> tag without body
- The <x:transform> tag with body
- The <x:transform> tag with body and optional parameters

The <x:transform> Tag without Body

The syntax to use the <x:transform> tag without body content is as follows:

```
<x:transform
xml="XMLDocument" xslt="XSLTStylesheet"
```

```
[xmlSystemId="XMLSystemId"] [xsltSystemId="XSLTSystemId"]
[{var="varName" [scope="scopeName"]|result="resultObject"}]/>
```

The <x:transform> Tag with Body

The syntax to use the <x:transform> tag with body content is as follows:

```
<x:transform
xml="XMLDocument" xslt="XSLTStylesheet"
[xmlSystemId="XMLSystemId"] [xsltSystemId="XSLTSystemId"]
[{var="varName" [scope="scopeName"]|result="resultObject"}]>
<x:param> tags </x:param>
</x:transform>
```

The <x:transform> Tag with Body and Optional Parameters

The syntax to use the <x:transform> tag with body and optional transformation parameters is as follows:

```
<x:transform
xslt="XSLTStylesheet"
xmlSystemId="XMLSystemId" xsltSystemId="XSLTSystemId"
[{var="varName" [scope="scopeName"]|result="resultObject"}]
XML Document to parse
optional <x:param> actions </X:param>
</x:parse>
where scopeName is {page|request|session|application}
```

Table 9.9 describes the attributes of the <x:transform> tag:

Table 9.9: Attributes of the <x:transform> Tag

Name of the Attribute	Type	Description
result	javax.xml.transform.Result	Specifies an object that in turn specifies how the transformation result should be processed.
scope	String	Specifies the scope for the variable containing the transformed XML document.
var	String	Specifies the name of the scoped variable that is exported to the parsed XML document. The scoped variable is of the org.w3c.dom.Document type.
xml	String, Reader, javax.xml.transform. Source, or object exported by <x:parse>, <x:set> or <x:transform>	Specifies an XML document as the source to be translated (if an XML document is exported by the <x:set> tag, then the <xml:transform> tag should transform the complete XML document).
xmlSystemId	String	Specifies the system identifier (URI) to parse an XML document.
xslt	String, Reader or javax.xml.transform.Source	Specifies a transformation XSLT as a String, Reader, or Source object.
xsltSystemId	String	Specifies the system identifier (URI) to parse an XSLT stylesheet.

The result of the transformation of an XML document by using XSLT is not stored in any object by default, rather, it is written directly on to a JSP page. You can store the result of a transformation by the following ways:

- Specifying the `result` attribute of the <x:transformation> tag with a value of the `Result` type
- Specifying the `var` and `scope` attributes of the <x:transformation> tag with a `Document` object saved in scoped variable.

Let's now discuss the <x:param> tag.

The <x:param> Tag

The <x:param> tag sets the transformation parameters for the nested action of the <x:transform> tag. The syntax of the <x:param> tag is as follows:

```
<x:param name="name" value="value"/>
Syntax 2: Parameter value specified in the body content
<x:param name="name">
parameter value
</x:param>
```

Table 9.10 describes the attributes of the <x:param> tag:

Table 9.10: Attributes of the <x:param> Tag

Name of the Attribute	Type	Description
name	String	Specifies the name of the transformation parameter
value	Object	Specifies the value of the transformation parameter

The <x:param> tag must be nested within the <x:transform> tag to set the transformation parameters. The value of the transformation parameter is specified either through the `attribute` value, or through the body content of the <x:transform> tag.

Let's now discuss how to implement the XML tags in a Web application.

Using the XML Tag Library in the XMLTagApp Application

In this section, we create the `XMLTagApp` Web application, to parse, extract, and transform an XML document.

The broad-level steps to create the `XMLTagApp` Web application are as follows:

- Create an XML file
- Create a JSP page
- Configure the application
- Define the directory structure of the application
- Package, deploy, and run the application

Creating the XML file

Let's create an XML file named books.xml to store or maintain the details of the various books in the `XMLTagApp` Web application. The code for the books.xml file is shown in Listing 9.5 (you can find the books.xml file on the CD in the `code\JavaEE\Chapter9\XMLTagApp` folder):

Listing 9.5: Showing the Details of the Books in the books.xml File

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

  <book>
  <title>Java EE Black Book</title>
  <description>The Best Book on Java EE </description>
  <author>Kogent Solutions Inc.</author>
  </book>

  <book>
  <title>Java 6 Black Book</title>
  <description>The Best Book on Java 6</description>
  <author>Kogent Solutions Inc.</author>
  </book>

</books>
</xml-body>
```

In Listing 9.5, the books.xml file is parsed and its elements are extracted in the details JSP page, which is the home page of the `XMLTagApp` Web application.

Creating the details JSP Page

Create a JSP page named details, which parses, extracts, and manipulates the data in the XML file by using XML tags. Listing 9.6 provides the code for the details JSP page (you can find the details.jsp file on the CD in the code\JavaEE\Chapter9\XMLTagApp folder):

Listing 9.6: Showing the Code for the details.jsp File

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
        <title>My JSP 'details.jsp' starting page</title>
        <link rel="stylesheet" type="text/css" href="mystyle.css">
  </head>
<body>
  <c:import url="books.xml" var="book"/>
  <x:parse xml="${book}" var="parsed"/>
  Details on the Books are as follows:
  <table>
  <tr>
  <th>Title of the Book:</th>
  <td width="10%"> </td>
  <th>Description</th>
  <td width="10%"> </td>
  <th>Author of the Book:</th>
  </tr>
  <x:forEach select="$parsed//book">
  <tr>
  <td colspan><x:out select="title"/></td>
  <td width="10%"> </td>
  <td colspan><x:out select="description"/></td>
  <td width="10%"> </td>
  <td><x:out select="author"/></td>
  </tr>
  </x:forEach>
  </table>
</body>
</html>
```

In Listing 9.6, the details JSP page uses the <c:import> tag to import the books.xml file. The page also uses the <x:parse> tag to parse the books.xml file and store the result in a variable called parsed. After the XML document is parsed, the details JSP page is used to display the details of each book in a table format by using the <x:for-each> tag.

Configuring the XMLTagApp Application

Let's now configure the XMLTagApp Web application in the web.xml file and set the details JSP page in the<welcome-file> tag. The code for web.xml is shown in the Listing 9.7 (you can find the web.xml file on the CD in the code\JavaEE\Chapter9\XMLTagApp\WEB-INF folder):

Listing 9.7: Showing the Code for the web.xml File for the XMLTagApp Application

```
<?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">
<welcome-file-list>
  <welcome-file> details.jsp </welcome-file>
</welcome-file-list>
</web-app>
```

Defining the Directory Structure of the XMLTagApp Application

Now, after creating the required files for the XMLTagApp Web application, you need to arrange the files in a directory structure, as shown in Figure 9.4:

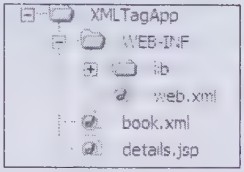


Figure 9.4: Showing the Directory Structure of the XMLTagApp Web Application

Packaging, Deploying, and Running the XMLTagApp Application

Perform the following steps to package, deploy, and run the XMLTagApp Web application:

1. Create the XmlTagApp.war file and deploy it on the GlassFish V3 application server.
2. Run the XMLTagApp Web application by using the `http://localhost:8080/XmlTagApp` URL. The details JSP page appears, as shown in Figure 9.5:

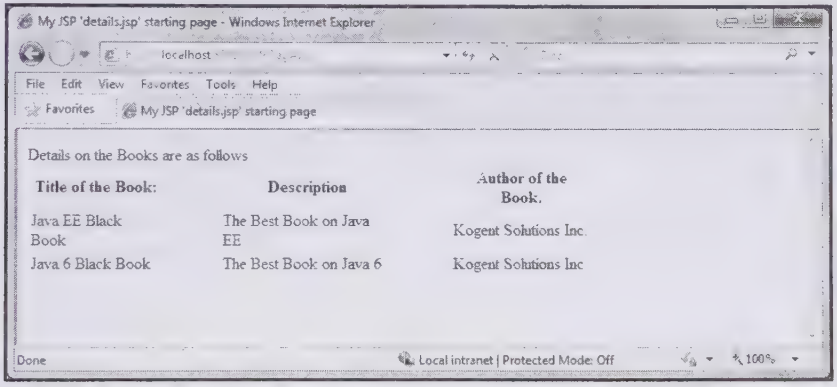


Figure 9.5: Showing the Book Details Extracted from an XML File in a JSP Page

With this, we complete our discussion on XML tags. Apart from covering different aspects of these tags, you have also learned how to implement these tags in a Web application.

Next, we discuss how to work with the internationalization tag library.

Working with the Internationalization Tag Library

The internationalization tags in JSTL are used to set a locale for a Web page, create localized messages, and format as well as parse data elements such as numbers, currencies, dates, and times in a localized or customized manner. Based on the locale of a client, the internationalization tags use a localization context containing a locale and a resource bundle to display the data to the client. The localization context is defined in the following two ways:

- Creating an instance of the LocalizationContext class
- Specifying a String value to the localization context parameter while deploying a Web application

Let's now explore the tags of the internationalization tag library.

Exploring the Tags of the Internationalization Tag Library

The internationalization tag library provides a set of tags that are categorized on the basis of their functionalities. Table 9.11 provides a list of the tags of the internationalization tag library along with their functions:

Table 9.11: Tags of the Internationalization Tag Library Categorized According to their Functions	
Function	**Tags**
Setting a locale	`setLocale` `requestEncoding`

Messaging	`bundle`
	`message`
	`param`
	`setBundle`

Formatting number and date	`formatNumber`
	`formatDate`
	`parseDate`
	`parseNumber`
	`setTimeZone`
	`timeZone`

Let's now explore the tags of the internationalization tag library on the basis of their functions.

Describing the Tags for Setting the Locale

A locale for JSP pages is set by the internationalization tag library. The language and formatting conventions of the locale are used to format the JSP page.

The internationalization tag library provides the following set of tags to set or format the locale in a JSP page:

- The <fmt:setLocale> tag
- The <fmt:requestEncoding> tag

Let's discuss each of these tags in detail.

The <fmt:setLocale> Tag

The <fmt:setLocale> tag sets the configuration of the locale variable by overriding the client-specific locale for a page. The syntax for using the <fmt:setLocale> tag is as follows:

```
<fmt:setLocale value="locale"
[variant="variant"]
[scope="{page|request|session|application}"]/>
```

In the preceding syntax, the value attribute is set to a locale value, which can be a String value, and which must be a language code followed by a country code.

Table 9.12 describes the attributes of the <fmt:setLocale> tag:

Table 9.12: Attributes of the <fmt:setLocale> Tag		
Name of the Attribute	**Type**	**Description**
`scope`	`String`	Returns the scope for a variable that contains the locale configuration information.
`value`	`String or Java.util. Locale`	Returns a value associated with a locale of the String type, which has two letters as language code (as defined by ISO-39) and must be in lowercase, and two letters as country code (as defined by ISO-3166), which must be in uppercase. There should be a hyphen (-) or (_) underscore between the language code and the country code.
`variant`	`String`	Returns a variant that is either browser or vendor specific.

Let's now discuss the <fmt:requestEncoding> tag.

The <fmt:requestEncoding> Tag

The <fmt:requestEncoding> tag is used to set the character encoding of a request. It is used to correctly decode the request parameter values if their encoding is different from that specified by ISO-8859-1.

The <fmt:requestEncoding> tag is needed because most browsers do not follow the HTTP-specification and do not include a Content-Type header in their requests.

The syntax of the <fmt:requestEncoding> tag is as follows:

```
<fmt:requestEncoding [value="charsetName"]/>
```

The <fmt:requestEncoding> tag provides the value attribute to specify the name of the character encoding to be applied while decoding the parameters of a request. This attribute accepts the String type value.

After learning about the tags used to format the locale, let's discuss the messaging tags of JSTL.

Exploring Messaging Tags

JSTL messaging tags are used to display content in a language identified by the locale set for a JSP page. The JSTL internalization tag library supports the following messaging tags:

- The <fmt:setBundle> tag
- The <fmt:bundle> tag
- The <fmt:message> tag

Now let's discuss each of these tags in detail.

The <fmt:setBundle> Tag

The <fmt:setBundle> tag is used to create a I18N localization context. The basename attribute is used to set the name of a resource bundle.

The I18N localization context is stored in the scoped variable defined by using the `var` attribute. If the name of a variable is not specified for the `var` attribute, the 118N localization context is stored in the `javax.servlet.jsp.jstl.fmt.LocalizationContext` configuration variable. This defines the default I18N localization context in the given scope.

The syntax of the <fmt:setBundle> tag is as follows:

```
<fmt:setBundle basename="basename"
[var="varName"]
[scope="{page|request|session|application}"]/>
```

Table 9.13 describes the attributes of the <fmt:setBundle> tag:

Table 9.13: Attributes of the <fmt:setBundle> Tag

Name of the Attribute	Type	Description
`basename`	`String`	Defines a name for the resource bundle. This is the fully qualified resource name of the bundle, which is as same as the fully qualified class name created in an application. You should note that the name specified for the basename attribute is separated from the package name by using a separator (.). In addition, the name does not contain any file type (such as .class or .properties) suffix.
`scope`	`String`	Defines the scope of the `var` attribute or a localization context configuration variable.
`var`	`String`	Defines the name of the scoped variable that contains the value of the object type, javax.servlet.jsp.jstl.fmt.LocalizationContext.

Let's now discuss the <fmt:bundle> tag.

The <fmt:bundle> Tag

The <fmt:bundle> tag creates a localization context and loads a resource bundle into this context. The `basename` attribute is used to specify the name of this resource bundle. The scope of this localized context is limited to the body of the <fmt:bundle> tag.

The syntax of the <fmt:bundle> tag is as follows:

```
<fmt:bundle basename="basename"
[prefix="prefix"]>
```

```
body content
</fmt:bundle>
```

Table 9.14 describes the attributes of the <fmt:bundle> tag:

Table 9.14: Attributes of the <fmt:bundle> Tag

Name of the Attribute	Type	Description
basename	String	Identifies a resource bundle to be set as the default bundle, which has been set by using the <fmt:setBundle> tag. This is the fully qualified resource name of the bundle.
prefix	String	Specifies the prefix to be appended to the value of the message key of any nested <fmt:message> action.

Let's now discuss the <fmt:message> tag.

The <fmt:message> Tag

The <fmt:message> tag contains a localized message corresponding to a given key.

The syntax of the <fmt:message> tag can be used in the following there ways:

- The<fmt:message> tag without body
- The <fmt:message> tag with body
- The <fmt:message> tag with message parameters

The <fmt:message> Tag without Body

The message key may be specified by the key attribute of the <fmt:message> tag.

The syntax of the <fmt:message> tag without body content is as follows:

```
<fmt:message key="messageKey"
[bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]/>
```

The <fmt:message> Tag with Body

When the <fmt:message> tag is used with body, the key attribute is not used and the message to be set is provided in the body of the tag.

The syntax of the <fmt:message> tag with body content is as follows:

```
<fmt:message [bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]>
Any Message
</fmt:message>
```

The <fmt:message> Tag with Message Parameters

In the presence of one or more <fmt:param> subtags, a localized text message is passed to the applyPattern() method, and the values of the <fmt:param> tags are collected in an Object[] array and passed to the format() method. The locale of the java.text.MessageFormat class is set to the appropriate localization context locale before the applyPattern() method is called.

If the message is compound (message that contains more than one variable) and no <fmt:param> subtags are specified, the applyPattern() method is not used. The <fmt:message> tag prints the output by using the current JSPWriter object, unless the var attribute is specified, in which case the result is stored in the named JSP attribute.

The syntax of the <fmt:message> with the message parameters specified by using the <fmt:param> tag is as follows:

```
<fmt:message [bundle="resourceBundle"]
[var="varName"]
[scope="{page|request|session|application}"]>
Key
```

```
optional <fmt:param> subtags</fmt:param>
</fmt:message>
```

The param subtag contains a single argument (for parametric replacements) that is passed to a compound message or provides a pattern or format in its parent message tag. The parameter values for the variables of a compound message may be specified by one or more <fmt:param> subtags (one for each parameter value). This procedure is known as parametric replacement.

In a compound message or pattern, at least one param tag must be specified for each variable. The following code snippet uses the <fmt:message>, <fmt:param>, and <fmt:formatNumber> tags to display the total number of athletes based on a specified client locale:

```
<fmt:message key="athletesRegistered">
<fmt:param>
<fmt:formatNumber value="${athletesCount}"/>
</fmt:param>
</fmt:message>
```

Depending on the locale, the output of the code snippet could be as follows:

```
french: Il y a 10 582 athletes enregistres.
english: There are 10,582 athletes registered.
```

If the <fmt:message> tag is nested inside a <fmt:bundle> tag, and the parent <fmt:bundle> tag contains a prefix attribute, the prefix specified to the prefix attribute is appended to a message key. The <fmt:message> tag uses a resource bundle of the I18N localization context identified to format the message according to the specified client locale.

Table 9.15 describes the attributes of the <fmt:message> tag:

Table 9.15: Attributes of the <fmt:message> Tag

Name of the Attribute	Type	Description
key	String	Specifies the message key used to locate a message
bundle	LocalizationContext	Specifies a localization context in which the message key looks up for a resource bundle
var	String	Specifies the name of an exported scoped variable, which stores a localized message
scope	String	Specifies the scope of the variable defined in the var attribute

Next, let's learn about formatting tags.

Describing Formatting Tags

JSTL provides formatting tags to parse and format locale-sensitive numbers and dates in a JSP page. The internationalization tag library provides the following formatting tags:

- The <fmt:formatNumber> tag
- The <fmt:formatDate> tag
- The <fmt:timeZone> tag
- The <fmt:setTimeZone> tag

The <fmt:formatNumber> Tag

The <fmt:formatNumber> tag is used to format a numeric value, such as a number, currency, or percentage, in a locale-sensitive or customized manner. The numeric value may be provided by the value attribute; if the value is not provided, then it is read from the body of the tag. Whether the given value is formatted as a number, currency, or percentage depends on the value of the type attribute.

The syntax of the <fmt:formatNumber> tag is as follows:

```
<fmt:formatNumber value="numericValue"
        [type="{number|currency|percent}"]
        [pattern="customPattern"]
        [currencyCode="currencyCode"]
```

```
[currencySymbol="currencySymbol"]
[groupingUsed="{true|false}"]
[maxIntegerDigits="maxIntegerDigits"]
[minIntegerDigits="minIntegerDigits"]
[maxFractionDigits="maxFractionDigits"]
[minFractionDigits="minFractionDigits"]
[var="varName"]
[scope="{page|request|session|application}"]/>
```

Table 9.16 describes the attributes of the <fmt:formatNumber> tag:

Table 9.16: Attributes of the <fmt:formatNumber> Tag

Name of the Attribute	Type	Description
currencyCode	String	Specifies the currency code (as defined by ISO 4217). The currencyCode attribute is applicable when currencies are to be formatted.
currencySymbol	String	Specifies the symbol of the currency. This attribute is only applicable when currencies need to be formatted (i.e., if type is equal to currency); otherwise, this attribute is ignored.
groupingUsed	boolean	Specifies whether or not any grouping separators, such as comma and semi colon are contained in the formatted output.
maxFractionDigits	int	Specifies the maximum number of digits in the fractional part of the formatted output.
maxIntegerDigits	int	Specifies the maximum number of digits in the integer part of the formatted output.
minFractionDigits	int	Specifies the minimum number of digits in the fractional part of the formatted output.
minIntegerDigits	int	Specifies minimum number of digits in the integer portion of the formatted output.
pattern	String	Specifies a custom formatting pattern, which is applied only when numbers are to be formatted. In other words, if the value of the type attribute is either not specified or is equal to number, then the specified pattern is applied. Otherwise, the specified custom pattern is ignored.
scope	String	Specifies the scope of the variable defined in the var attribute.
type	String	Specifies the type to which a value is to be formatted, such as number, currency, or percentage.
value	String or Number	Specifies the numeric value to be formatted.
var	String	Specifies the name of the scoped variable used to store the formatted result in the form of a String.

Let's now discuss the <fmt:formatDate> tag.

The <fmt:formatDate> Tag

The <fmt:formatDate> tag is used to format the date and time that is displayed in a JSP page. This tag depends on the value of the type attribute. You can format the time component, the date component, or both the components of a specified date. The date and time components are formatted by using one of the predefined formatting styles for date (specified by the dateStyle attribute) and time (specified by the timeStyle attribute) of a JSP page's formatting locale.

The syntax of the <fmt:formatDate> tag is:

```
<fmt:formatDate value="date"
[type="{time|date|both}"]
```

```
[dateStyle="{default|short|medium|long|full}"]
[timeStyle="{default|short|medium|long|full}"]
[pattern="customPattern"]
[timeZone="timeZone"]
[var="varName"]
[scope="{page|request|session|application}"]/>
```

Web designers may also apply a customized formatting style to time and date by specifying the `pattern` attribute, in which attributes, such as `type`, `dateStyle`, and `timeStyle` are ignored. The <fmt:formatDate> tag is used to format the current date and time, as shown in the following code snippet:

```
<JSP:useBean id="now" class="Java.util.Date" />
<fmt:formatDate value="${now}" />
```

If the date or time is in the form of a String and needs to be formatted, this String object must be parsed into the `Date` object by using the <fmt:parseDate> tag. The result is provided to the <fmt:formatDate> tag. The following code snippet shows how the <fmt:parseDate > tag is used to parse the date and the <fmt:formatDate> tag is used to format the parsed date:

```
<fmt:parseDate value="4/13/02" var="parsed" />
<fmt:formatDate value="${parsed}" />
```

Table 9.17 describes the attributes of the <fmt:formatDate> tag:

Table 9.17: Attributes of the <fmt:formatDate> Tag

Name of the Attribute	Type	Description
`value`	`java.util.Date`	Specifies the date and the time to be formatted.
`type`	`String`	Specifies whether time or date or both time and date need to be formatted.
`dateStyle`	`String`	Specifies an already defined formatting style for the date, which follows the semantics specified in the `java.text.DateFormat` class. The formatting style specified in the dateStyle attribute is applicable only when a date or both date and time need to be formatted. In other words, if the value of the type attribute is either not specified or is equal to `date`, then the specified formatting style is applied. Otherwise, the specified style is ignored.
`timeStyle`	`String`	Specifies an already defined formatting style for the date that follows the semantics specified in the `java.text.DateFormat` class. The formatting style specified in the dateStyle attribute is applicable only when a time or both date and time need to be formatted. In other words, if the value of the type attribute is either not specified or is equal to `time`, then the specified formatting style is applied. Otherwise, the specified style is ignored.
`pattern`	`String`	Specifies a custom formatting style used to format the date and time.
`timeZone`	`String` or `java.util.TimeZone`	Specifies the time zone in which to represent the formatted time.
`var`	`String`	Specifies the scoped variable that stores a String value as a result of formatting. By default, the scope of the variable assigned to the `var` attribute is false.

Time information on a page may be tailored to the preferred time zone of a client. This is useful when a server is accessed by different clients in different time zones. If information about the time to be formatted or parsed in a time zone is different from that in the JSP container, the `<fmt:formatDate>` and `<fmt:parseDate>` tags may

be nested inside the <fmt:timeZone> tag or provided with the timeZone attribute. We discuss the <fmt:timeZone> tag in the following section.

The <fmt:timeZone> Tag

The <fmt:timeZone> tag specifies the time zone in which time information is to be formatted or parsed. The syntax of the <fmt:timeZone> tag is as follows:

```
<fmt:timeZone value="timeZone">
body content
</fmt:timeZone>
```

In the following code snippet, the current date and time are formatted in the GMT+1:00 time zone:

```
<fmt:timeZone value="GMT+1:00">
  <fmt:formatDate value="${now}" type="both" dateStyle="full"
  timeStyle="full"/>
</fmt:timeZone>
```

The <fmt:timeZone> tag provides a single attribute, value, which is either of the String or TimeZone type. If the time zone is given as a String value in the <fmt:timeZone> tag, the value is parsed by using the getTimeZone() method.

Now, let's learn about the <fmt:setTimeZone> tag.

The <fmt:setTimeZone> Tag

The <fmt:setTimeZone> tag stores the specified time zone in a scoped variable or time zone configuration variable. The syntax of the <fmt:setTimeZone> tag is as follows:

```
<fmt:setTimeZone value="timeZone"
[var="varName"]
[scope="{page|request|session|application}"]/>
```

Table 9.18 describes the attributes of the <fmt:setTimeZone> tag:

Table 9.18: Attributes of the <fmt:setTimeZone> Tag

Name of the Attribute	Type	Description
scope	String	Specifies the scope of the variable assigned to the var attribute or time zone configuration variable
value	String or java.util.TimeZone	Defines the time zone ID for a region. For example, for America/Los_Angeles, the time zone ID used is GMT-8.
var	String	Defines the name of a scoped variable that stores the time zone as a java.util.TimeZone object.

If the var attribute is not given, the time zone is stored in the timeZone configuration variable, thereby making it as a new default time zone of the given scope. If time zone is defined as a String value, then the time zone is parsed by using the getTimeZone() method.

After exploring the internationalization tags, let's now learn how to use them in a Web application.

Using the Internationalization Tag Library in Web Applications

In this section, we implement the tags of the internationalization tag library. We create two separate Web applications named dateFormatApp and numFormatApp, to demonstrate the use of the <fmt:formatDate> and <fmt:formatNumber> tags, respectively.

Using the <fmt:formatDate> Tag in the dateFormatApp Application

Let's create a Web application named dateFormatApp, which implements the <fmt:formatDate> tag of the internationalization tag library.

The following are the broad-level steps to create the dateFormatApp Web application:

- Create a JSP page
- Configure the Web application
- Define the directory structure of the Web application

- Package, deploy, and run the Web application

Let's now perform these steps in the following sections.

Creating the formatDate JSP Page

The formatDate JSP page is the home page of the `dateFormatApp` Web application and displays the current date and time of most of the important cities of the world. It also displays the formatted current date and time of these cities. The `<fmt:formatDate>` tag is used in the formatDate JSP page to format the date in a specified pattern.

Listing 9.8 provides the code of the formatDate JSP page (you can find the formatDate.jsp file on the CD in the `code\JavaEE\Chapter9\dateFormatApp` folder):

Listing 9.8: Showing the Code for the formatDate.jsp File

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head>
        <title>CHECK TIME!!!</title>
        <link rel="stylesheet" type="text/css" href="mystyle.css">
  </head>
<body>
  <c:set var="now" value="<%=new java.util.Date()%>" />
  <c:set var="los" value="<%=TimeZone.getTimeZone("America/Los_Angeles")%>" />
  <c:set var="lon" value="<%=TimeZone.getTimeZone("Europe/London")%>" />
  <c:set var="rom" value="<%=TimeZone.getTimeZone("Rome ")%>" />
  <c:set var="pari" value="<%=TimeZone.getTimeZone("Paris")%>" />
  <c:set var="port" value="<%=TimeZone.getTimeZone("Portugal ")%>" />
  <c:set var="ind" value="<%=TimeZone.getTimeZone("IST")%>" />
    <c:set var="syd" value="<%=TimeZone.getTimeZone("Australia/Sydney")%>" />
<b>CURRENT TIME AT SOME OF THE POPULAR CITIES OF THE WORLD!!</b>
<BR/>
<BR/>
<table border="1" cellpadding="0" cellspacing="0"
  style="border-collapse: collapse" bordercolor="#111111"
  width="63%" id="AutoNumber2">
<tr>
  <td width="51%" colspan="2" bgcolor="yellow">Los Angeles:</td>
  <td width="49%" colspan="2" bgcolor="yellow">
  <fmt:timeZone value="$los}">
  <fmt:formatDate value="${now}" timeZone="${los}"
  type="both" />
  </fmt:timeZone>
  </td>
</tr>
<tr>
  <td width="51%" colspan="2" bgcolor="yellow">London:</td>
  <td width="49%" colspan="2" bgcolor="yellow">
  <fmt:timeZone value="$lon}">
  <fmt:formatDate value="${now}" timeZone="${lon}"
  type="both" />
  </fmt:timeZone>
  </td>
</tr>
<tr>
  <td width="51%" colspan="2" bgcolor="yellow">Rome:</td>
  <td width="49%" colspan="2" bgcolor="yellow">
  <fmt:timeZone value="$rom}">
  <fmt:formatDate value="${now}" timeZone="${rom}"
  type="both" />
  </fmt:timeZone>
```

```
    </td>
  </tr>
  <tr>
    <td width="51%" colspan="2" bgcolor="yellow">Paris:</td>
    <td width="49%" colspan="2" bgcolor="yellow">
    <fmt:timeZone value="$pari}">
    <fmt:formatDate value="${now}" timeZone="${pari}"
    type="both" />
    </fmt:timeZone>
    </td>
  </tr>
  <tr>
    <td width="51%" colspan="2" bgcolor="yellow">New- Delhi:</td>
    <td width="49%" colspan="2" bgcolor="yellow">
    <fmt:timeZone value="$ind}">
    <fmt:formatDate value="${now}" timeZone="${ind}"
    type="both" />
    </fmt:timeZone>
    </td>
  </tr>
  <tr>
    <td width="51%" colspan="2" bgcolor="yellow">Sydney:</td>
    <td width="49%"colspan="2" bgcolor="yellow">
    <fmt:timeZone value="$syd}">
    <fmt:formatDate value="${now}" timeZone="${syd}"
    type="both" />
    </fmt:timeZone>
    </td>
  </tr>
  </table>
  <BR>
  <BR>
  <BR>
  <BR>
  <B>FORMATTING DATE USING PATTERN:<B>
  <BR>
  <BR>
  <table border="1" cellpadding="0" cellspacing="0"
  style="border-collapse: collapse" bordercolor="#111111"
  width="63%" id="AutoNumber2">
  <tr>
    <td width="51%" colspan="2" bgcolor="yellow">Los Angeles:</td>
    <td width="49%" colspan="2" bgcolor="yellow">
    <fmt:timeZone value="$los}">
    <fmt:formatDate value="${now}" timeZone="${los}"
    type="both" pattern="dd-MMM-yyyy hh:mm:ss" />
    </fmt:timeZone>
    </td>
  </tr>
  <tr>
    <td width="51%" colspan="2" bgcolor="yellow">London:</td>
    <td width="49%" colspan="2" bgcolor="yellow">
    <fmt:timeZone value="$lon}">
    <fmt:formatDate value="${now}" timeZone="${lon}"
    type="both" pattern="dd-MMM-yyyy hh:mm:ss"/>
    </fmt:timeZone>
    </td>
  </tr>
  <tr>
    <td width="51%" colspan="2" bgcolor="yellow">Rome:</td>
    <td width="49%" colspan="2" bgcolor="yellow">
    <fmt:timeZone value="$rom}">
```

```
    <fmt:formatDate value="${now}" timeZone="${rom}"
    type="both" pattern="dd-MMM-yyyy hh:mm:ss"/>
    </fmt:timeZone>
    </td>
  </tr>
  <tr>
    <td width="51%" colspan="2" bgcolor="yellow">Paris:</td>
    <td width="49%" colspan="2" bgcolor="yellow">
    <fmt:timeZone value="$pari}">
    <fmt:formatDate value="${now}" timeZone="${pari}"
    type="both" pattern="dd-MMM-yyyy hh:mm:ss"/>
    </fmt:timeZone>
    </td>
  </tr>
  <tr>
    <td width="51%" colspan="2" bgcolor="yellow">New- Delhi:</td>
    <td width="49%" colspan="2" bgcolor="yellow">
    <fmt:timeZone value="$ind}">
    <fmt:formatDate value="${now}" timeZone="${ind}"
    type="both" pattern="dd-MMM-yyyy hh:mm:ss" />
    </fmt:timeZone>
    </td>
  </tr>
  <tr>
    <td width="51%" colspan="2" bgcolor="yellow">Sydney:</td>
    <td width="49%"colspan="2" bgcolor="yellow">
    <fmt:timeZone value="$syd}">
    <fmt:formatDate value="${now}" timeZone="${syd}"
    type="both" pattern="dd-MMM-yyyy hh:mm:ss"/>
    </fmt:timeZone>
    </td>
  </tr>
  </table>
  </body>
  </html>
```

In Listing 9.8, the time zone for the selected city is set in the `timeZone` variable and is retrieved by using the `TimeZone.getTimeZone()` method. The variable assigned to the `timezone` attribute is then used by the `<fmt:formatDate>` tag to set the time according to the time zone given as a parameter to the `timezone` attribute.

Configuring the dateFormatApp Application

Let's now configure the `dateFormatApp` application in the web.xml file. Listing 9.9 shows the code for the web.xml file (you can find this file on the CD in the code\JavaEE\Chapter9\dateFormatApp\WEB-INF folder):

Listing 9.9: Showing the web.xml File for the dateFormatApp Application

```
<?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">
<welcome-file-list>
  <welcome-file>formatDate.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

In Listing 9.9, the formateDate.jsp file is set as the welcome file.

Now let's understand the directory structure of the `dateFormatApp` Web application.

Defining the Directory Structure of the dateFormatApp Application

You can create a directory structure and arrange the files created for the `dateFormatApp` Web application, as shown in Figure 9.6:

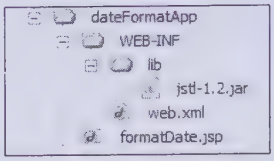


Figure 9.6: Showing the Directory Structure of the dateFormatApp Web Application

Packaging, Deploying, and Running the dateFormatApp Application

Perform the following steps to package, deploy, and run the dateFormatApp Web application:

1. Create the dateFormatApp.war file and deploy it on the GlassFish V3 application server.
2. Run the Web application by using the `http://localhost:8080/dateFormatApp` URL. The formatDate JSP page is displayed, as shown in Figure 9.7:

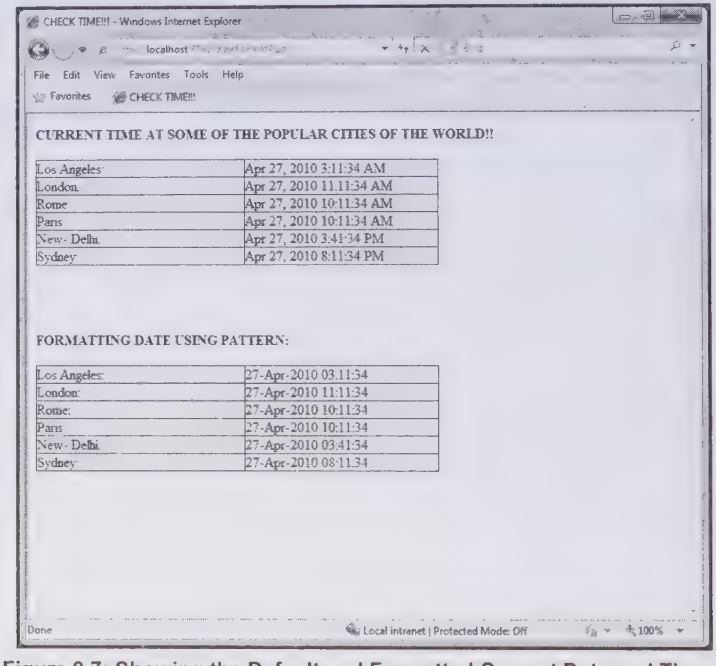


Figure 9.7: Showing the Default and Formatted Current Date and Times

After learning to use the `<fmt:formatDate>` tag in a Web application, let's learn to use the `<fmt:formatNumber>` tag.

Using the <fmt:formatNumber> Tag in the numFormatApp Application

Let's create a Web application named `numFormatApp`, which uses the `<fmt:formatNumber>` tag of the internationalization tag library. The home page of the Web application is index.html, which accepts a number from a user. The number is stored in the mynumber attribute that is passed to a JSP page named formatNum. The formatNum.jsp file formats the value of the mynumber attribute by using number format tags.

The following are the broad-level steps to create the `numFormatApp` Web application:

- Create the JSP and HTML pages
- Configure the Web application
- Define the directory structure of the Web application
- Package, deploy, and run the Web application

Let's now perform these steps in the following sections.

Creating the formatNum JSP and index HTML Pages

The index HTML page is the home page of the `numFormatApp` application. The code for the index.html file is shown in Listing 9.10 (you can find the index.html file on the CD in the `code\JavaEE\Chapter9\numFormatApp` folder):

Listing 9.10: Showing the index.html File

```
<html>
<head>
  <title>Format a number</title>
</head>
<body>
  <form method=post action="formatNum.jsp">
  <pre>
        <b>Number :  <input type="text" name="mynumber"/>
        <input type="submit" value="Format Number"/>
        </b>
  </pre>
  </form>
</body>
</html>
```

The `numFormatApp` Web application displays index as the home page, which contains the `Format Number` button and a text box to accept a number to be formatted. When a user clicks the `Format Number` button, the control is transferred to the formatNum JSP page. Listing 9.11 shows the code for the formatNum JSP page (you can find the formatNum.jsp file on the CD in the `code\JavaEE\Chapter9\numFormatApp` folder):

Listing 9.11: Showing the Code for the formatNum.jsp File

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
        <title>welcome!!</title>
        <link rel="stylesheet" type="text/css" href="mystyle.css">
  </head>
  <body>
        My Number:
        <b><c:out value="${param.mynumber}"/></b> <br/>
        Formatting My Number: <br/><br/>
        <fmt:setLocale value="fr"/>
        In FRENCH:
        <pre>
        <b>MY NUM: <fmt:formatNumber type="number" value="${param.mynumber}"/></b>
        Default pattern:<b>
        <fmt:formatNumber type="currency" value="${param.mynumber}"/></b>
        Using pattern (0,00,00.0000):
        <b>
        <fmt:formatNumber type="currency" value="${param.mynumber}"
        pattern="00,0,00.0000"/> </b>
        </pre>
        <br/>
        In US:
        <fmt:setLocale value="en_US"/>
        <pre>
        <b>MY NUM: <fmt:formatNumber type="number" value="${param.mynumber}"/></b>
        Default pattern:<b>
        <fmt:formatNumber type="currency" value="${param.mynumber}"/></b>
        Using pattern (0,00,00.0000):
        <b>
        <fmt:formatNumber type="currency" value="${param.mynumber}" currencySymbol="$"
        pattern="0,00,00.0000" var="fmt_mynumber"/>
        <c:out value="${fmt_mynumber}"/>
        </b></pre>
        <br/>
  </body>
</html>
```

Listing 9.11 retrieves the value of the mynumber attribute provided by the user in the home page of the Web application and formats it by using the `<fmt:formatNumber>` tag.

Now, let's configure the `numFormatApp` Web application in the web.xml file.

Configuring the numFormatApp Application

You need to configure the index HTML page of the `numFormatApp` application in the web.xml file. The code for the web.xml file is shown in the Listing 9.12 (you can find the web.xml file on the CD in the `code\JavaEE\Chapter9\numFormatApp\WEB-INF` folder):

Listing 9.12: Showing the web.xml File for the `numFormatApp` Application

```
<?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">
<welcome-file-list>
  <welcome-file> index.html </welcome-file>
</welcome-file-list>
</web-app>
```

In Listing 9.12, the index.html file is mapped as the welcome file in Deployment Descriptor (web.xml).

Defining the Directory Structure of the `numFormatApp` Application

After configuring the `numFormatApp` Web application, you create a directory structure to arrange the files of the application. Figure 9.8 shows the directory structure of the of the `numFormatApp` application:

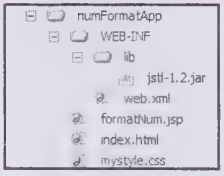


Figure 9.8: Showing the Directory Structure of the numFormatApp Application

After creating the directory structure of the numFormatApp Web application, we need to package, deploy, and run the application.

Packaging, Deploying, and Running the numFormatApp Application

Perform the following steps to package, deploy and run the `numFormatApp` application:

1. Create the numFormatApp.war file and deploy it on the GlassFish V3 application server.
2. Run the Web application by using the URL `http://localhost:8080/numFormatApp`. The index HTML page of the Web application appears, as shown in Figure 9.9:

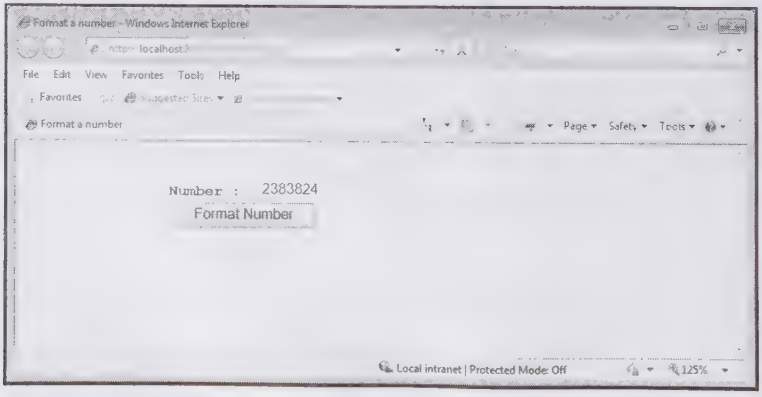


Figure 9.9: Displaying the Home Page of the numFormatApp Application

3. Enter a number in the Number text box in the index HTML page. In our case, we enter 2383824 and click the `Format Number` button. The control is passed to the formatNum JSP page to process the request.

   The formatNum JSP page formats the given number in French and US locale formats, with the default and given pattern, as shown in Figure 9.10:

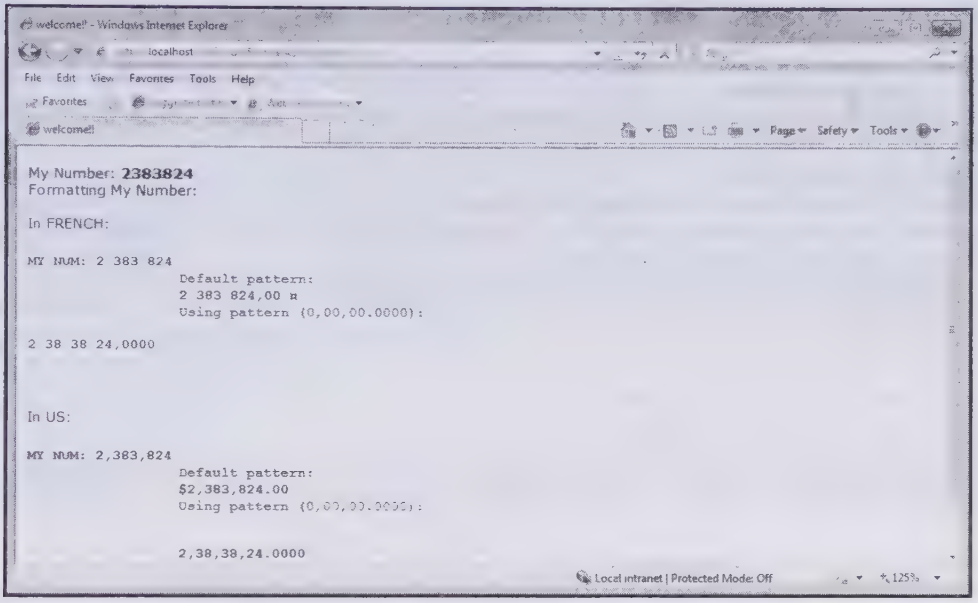


Figure 9.10: Showing a Number Formatted According to Various Locales

Let's now learn about the tags of the SQL tag library.

Working with the SQL Tag Library

The fourth JSTL tag library is the SQL tag library, which provides tags for interacting with relational databases. Interaction with databases is required to perform operations such as specifying data sources, creating queries, and performing updates. Web designers often need these SQL tags to access databases from JSP pages.

The SQL tag library depends on data sources to obtain connections. SQL statements are executed and the results are returned within the context of a connection retrieved from a data source. The data source is explicitly specified by the `dataSource` attribute of the `<sql:setDataSource>` tag.

Let's first explore the tags of the SQL tag library and then learn how to implement them in a Web application.

Exploring Tags of the SQL Tag Library

The tags of the SQL tag library are categorized based on the functionalities specified in Table 9.19:

Table 9.19: Functional Categorization and Associated Tags of the SQL Tag Library

Function	Tags
Getting a database connection	`setDataSource`

Accessing a database	`query`
	`dateParam`
	`param`
	`transaction`
	`update`

Now, let's discuss these tags under each functional category in detail.

The Tag for Getting a Database Connection

The SQL tag library provides the <sql:setDataSource> tag to allows an application to connect to databases. You can use the <sql:setDataSource> tag to get instances of a data source by either of the following ways:

- **Using the dataSource attribute**—Allows you to access a data source associated with a Java Naming and Directory Interfaces (JNDI) name. You can do this by passing the JNDI name as the value of the dataSource attribute. The following code snippet shows how to set a data source by setting the value for the dataSource attribute:

```
<sql:setDataSource dataSource="<Expression>" var="<Name>" scope="<scope>"/ >
```

- **Using the url attribute**—Allows you to set the URL for a Java Database Connectivity (JDBC) connection to the url attribute of the <sql:setDataSource> tag. Apart from the url attribute, you may also need to set the driver attribute, which is optional and specifies a class implementing a database server. If this is required, then user name and password also need to be specified to access a database. The syntax of the <sql:setDataSource> tag is as follows:

```
<sql:setDataSource url="<Expression>" driver="<Expression>" user="<Expression>"
  password="<Expression>" var="<Name>" scope="<Scope>"/>
```

JSTL provides tags that are useful in interacting with relational databases with the help of certain tags defined in a tag library called the SQL tag library. Table 9.20 describes the attributes of the <sql:setDataSource> tag:

Table 9.20: Attributes of the <sql:setDataSource> Tag

Table 9.20: Attributes of the <fmt:setTimeZone> Tag

Name of the Attribute	Type	Description
dataSource	String or javax.sql.DataSource	Specifies a data source. If specified as a String, this attribute can either be a relative path to a JNDI resource, or a JDBC parameter, String.
driver	String	Specifies the JDBC driver class name.
url	String	Specifies the JDBC URL used to connect to a database.
user	String	Specifies the user credentials on behalf of which a connection to the database is created.
password	String	Specifies a password for the user assigned to the user attribute with which the JDBC connection to a data source is established.
var	String	Specifies the name of the exported scoped variable for the specified data source.
scope	String	Specifies the scope of a variable for the specified data source. The <sql:setDataSource> tag exports the data source specified (either as a DataSource object or as a String) as a scoped variable.

A data source may be specified by either the dataSource attribute (as a DataSource object, JNDI relative path, or by providing the details for various JDBC attributes, such as driver, url, user, password.

Tags to Access a Database

The SQL tag library provides various tags that are used to access a database. These tags are as follows:

- The <sql:query> tag
- The <sql:update> tag
- The <sql:transaction> tag
- The <sql:param> tag
- The <sql:dateParam> tag

Now, let's discuss each of these tags in detail.

The <sql:query> Tag

The `<sql:query>` tag is used to query a database.

The syntax of the `<sql:query>` tag can be used in the following ways:

- The <sql:query> tag without body
- The <sql:query> tag with body
- The <sql:query> tag with optional query parameters

The <sql:query> Tag without Body

The syntax to use the <sql:query> tag without body content is as follows:

```
<sql:query sql="sqlQuery"
var="varName" [scope="{page|request|session|application}"]
[dataSource="dataSource"]
[maxRows="maxRows"]
[startRow="startRow"]/>
```

The <sql:query> Tag with Body

The syntax to use the <sql:query> tag with a body to specify query arguments is as follows:

```
<sql:query sql="sqlQuery"
var="varName" [scope="{page|request|session|application}"]
[dataSource="dataSource"]
[maxRows="maxRows"]
[startRow="startRow"]>
<sql:param> actions</sql:param>
</sql:query>
```

The <sql:query> Tag with Optional Query Parameters

The syntax to use <sql:query> tag with a body to specify query and optional query parameters is as follows:

```
<sql:query var="varName"
[scope="{page|request|session|application}"]
[dataSource="dataSource"]
[maxRows="maxRows"]
[startRow="startRow"]>
query
optional <sql:param> actions </sql:param>
</sql:query>
```

Table 9.21 describes the attributes of the <sql:query> tag:

Table 9.21: Attributes of the <sql:query> Tag

Name of the Attribute	Type	Description
`dataSource`	`javax.sql.DataSource` or `String`	Specifies the name of the variable assigned to the dataSource attribute of the <sql:setDataSource> tag, which is associated with the database used to execute a query. The relative path to a JNDI resource or the parameters for the JDBC DriverManager facility is represented by a String value.
`maxRows`	`int`	Specifies the maximum number of rows to be included in a query result. If the maximum number of rows is not specified, or set to -1, no limit on the maximum number of rows is enforced.
`scope`	`String`	Specifies the scope of the variable assigned to the `var` attribute for a query result.
`sql`	`String`	Specifies an SQL query statement, such as SELECT, UPDATE, and DELETE in a JSP page.
`startRow`	`int`	Specifies that a returned `Result` object includes the rows starting at the index specified for the `startRow` attribute. The first row of the original query resultset is at index 0. If the index is not specified, rows are included starting from the first row at index 0.
`var`	`String`	Specifies the name of an exported scoped variable for a query result.

The <sql:query> tag queries a database and gets back a single resultset containing rows of data. If the query produces no result, an empty Result object (of size zero) is returned. The SQL query statement may be specified by the sql attribute or from the body content of the <sql:query> tag.

A parameter marker (?) can be used in a query statement that represents the parameters of the JDBC PreparedStatement statement. You can provide values for these parameters by using nested parameter tags, such as <sql:param>. The <sql:query> tag implements the SQLExecutionTag interface, allowing parameter values to be provided by custom parameter actions.

The <sql:update> Tag

The <sql:update> tag executes an SQL INSERT, UPDATE, or DELETE statement. In addition, the <sql:update> tag can be used to execute SQL Data Definition Language (DDL) statements.

The syntax of the <sql:update> tag can be used in the following three ways:

- The <sql:update> tag without body
- The <sql:update> tag with body
- The <sql:update> tag with optional update parameters

The <sql:update> Tag without Body

The syntax of the <sql:update> tag without body content is as follows:

```
<sql:update sql="sqlUpdate"
[dataSource="dataSource"]
[var="varName"] [scope="{page|request|session|application}"]/>
```

The <sql:update> Tag with Body

The syntax to use the <sql:update> tag with a body to specify an update parameter is as follows:

```
<sql:update sql="sqlUpdate"
[dataSource="dataSource"]
[var="varName"] [scope="{page|request|session|application}"]>
<sql:param> actions </sql:param>
</sql:update>
```

The <sql:update> Tag with Optional Update Parameters

The syntax to use the <sql:update> tag to specify an update statement and optional update parameters is as follows:

```
<sql:update [dataSource="dataSource"]
[var="varName"] [scope="{page|request|session|application}"]>
update statement
optional <sql:param> actions</sql:param>
</sql:update>
```

Table 9.22 describes attributes of the <sql:update> tag:

Table 9.22: Attributes of the <sql:update> Tag

Name of the Attribute	Type	Description
dataSource	javax.sql.DataSource or String	Specifies the name of the variable assigned to the dataSource attribute of the <sql:setDataSource> tag. The data source associated with the variable is used to execute an update query. The relative path to a JNDI resource or the parameters for the JDBC DriverManager facility is represented by a String value.
scope	String	Specifies the scope of the variable assigned to the var attribute that is used for the result of the update SQL statement.
sql	String	Specifies an update SQL statement in a JSP page.
var	String	Specifies the name of the scoped variable used for the result of a database update. The type of this variable is java.lang.Integer.

The sql attribute or body content of an action represents the SQL update statement. Parameter markers (?) may exist in the update statement, representing JDBC PreparedStatement parameters. Similar to the <sql:query> tag, you can provide parameter values for the <sql:update> tag by using nested parameter actions, such as <sql:param>. The <sql:update> tag also implements the SQLExecutionTag interface.

The connection to a database is obtained in the same manner as described for the <sql:query> tag. The result of the <sql:update> tag is stored in a scoped variable defined by the var attribute, if the attribute has been specified. The result represents the number of rows affected by the update. Zero is returned if no rows are affected by the update statement.

The <sql:transaction> Tag

The <sql:transaction> tag establishes a transaction context for the <sql:query> and <sql:update> subtags.

The syntax of the <sql:transaction> tag is as follows:

```
<sql:transaction [dataSource="dataSource"]
[isolation=isolationLevel]>
<sql:query> and <sql:update> statements
</sql:transaction>
isolationLevel ::= "read_committed"
| "read_uncommitted"
| "repeatable_read"
| "serializable"
```

Table 9.23 describes the attributes of the <sql:transaction> tag:

Table 9.23: Attributes of the <sql:transaction> Tag

Name of the Attribute	Type	Description
dataSource	javax.sql.DataSource or String	Specifies the name of the variable assigned to the dataSource attribute of the <sql:setDataSource> tag. The data source associated with the variable is used to execute SQL transactions. The relative path to a JNDI resource or parameters for the JDBC DriverManager facility is represented by a String value.
isolation	String	Specifies the transaction isolation level. If not specified, this attribute represents the isolation level with which the data source is configured.

The <sql:transaction> tag groups nested <sql:query> and <sql:update> tags into a transaction. The transaction isolation levels are the ones defined by the java.sql.Connection interface. The tag handler of the <sql:transaction> tag must perform the following tasks in its life cycle by using the following methods:

- **doCatch()**—Calls the rollback() method of the Connection interface.
- **doEndTag()**—Calls the commit() method of the Connection interface.
- **doFinally()**—Enables the auto-commit mode by calling the setAutoCommit(true) method on the Connection object. The doFinally() method closes a JDBC connection when a transaction isolation level is saved. The connection is restored by using the setTransactionIsolation() method.
- **doStartTag()**—Specifies the transaction isolation level of Database Management System (DBMS). If the isolation level of transaction is set to TRANSACTION_NONE (i.e., transactions are not supported), an exception is thrown, which results in the failure of the transaction. For any other transaction isolation level, the auto-commit mode should be disabled by using the setAutoCommit (false) method of the Connection class. If the isolation attribute is specified, the current transaction isolation level is saved (and therefore can be restored later) and set to the specified level (by using the setTransactionIsolation() Connection method).

The Connection object is obtained and managed similar to the <sql:query> tag, and it can be never obtained from a parent tag. In other words, the <sql:transaction> tag cannot be nested to propagate a connection.

The <sql:transaction> tag commits and rollbacks a transaction (if it catches an exception) by calling the JDBC Connection commit() and rollback() methods, respectively. As the <sql:transaction> tag body does not support the execution of SQL statements by using the <sql:update> tag, the result is unpredictable. The behavior of the <sql:transaction> tag is undefined if it is executed in the context of a larger JTA user transaction.

To ensure database integrity, several updates to a database may be grouped into a transaction by nesting multiple <sql:update> tags inside a <sql:transaction> tag. For example, the following code snippet shows how the <sql:transaction> tag is used to transfer money between two accounts in one transaction, by using multiple updates:

```
<sql:transaction dataSource="${dataSource}">
<sql:update>
UPDATE account
SET Balance = Balance - ?
WHERE accountNo = ?
<sql:param value="${transferAmount}"/>
<sql:param value="${accountFrom}"/>
</sql:update>
<sql:update>
UPDATE account
SET Balance = Balance + ?
WHERE accountNo = ?
<sql:param value="${transferAmount}"/>
<sql:param value="${accountTo}"/>
</sql:update>
</sql:transaction>
```

Let's now discuss the <sql:param> tag.

The <sql:param> Tag

The <sql:param> tag is used as a subtag (such as <sql:query> and <sql:update>) of the SQLExecutionTag interface that is used to set the values of parameter markers (?) in an SQL statement.

The syntax of the <sql:param> tag can be used in either of the following ways:

- The <sql:param> tag without body
- The <sql:param> tag with body

The <sql:param> Tag without Body

The syntax of using the <sql:param> tag without body is as follows:

```
<sql:param value="value"/>
```

The <sql:param> Tag with Body

The sayntax of the <sql:param> tag with body content containing the value of a parameter is as follows:

```
<sql:param>
parameter value
</sql:param>
```

The <sql:param> tag provides a single attribute, value, which is used to provide a value to a parameter.

The <sql:param> tag substitutes a value for a parameter marker (?) in an SQL statement. Parameters are substituted in the order in which they are specified. The <sql:param> tag locates its nearest ancestor, which is an instance of the SQLExecutionTag interface, and calls its addSQLParameter() method. The specified parameter value is provided to the addSQLParameter() method to substitute the value in the parameter marker in the SQL statement.

The <sql:dateParam> Tag

The <sql:dateParam> tag is used as a subtag of the SQLExecutionTag interface to set the values of parameter markers (?) for values of the java.util.Date type.

The syntax of the <sql:dateParam> tag is as follows:

```
<sql:dateParam value="value" type="[timestamp|time|date]"/>
```

Table 9.24 describes the attributes of the <sql:dateParam> tag:

Table 9.24: Attributes of the <sql:dateParam> Tag

Name of the Attribute	Type	Description
value	java.util.Date	Specifies a parameter value for the DATE, TIME, or TIMESTAMP column in a database table.
type	String	Specifies a date, time or timestamp. By default, the timestamp is provided.

The <sql:dateParam> tag converts the specified Date object to objects such as java.sql.Date, java.sql.Time, or java.sql.Timestamp with the help of the value attribute.

If the specified Date object is an instance of the java.sql.Time, java.sql.Date, or java.sql.Timestamp class, the Date object is passed directly to a database without any type conversion.

Next, let's learn to use the tags of the SQL tag library in a Web application.

Using the SQL Tag Library in the SqlTagApp Application

In this section, we implement SQL tags with the help of a Web application named SqlTagApp.

The following are the broad-level steps to create the SqlTagApp Web application:

- Create the JSP page
- Create a book table in the Oracle database
- Configure the Web application
- Define the directory structure of the Web application
- Package, deploy, and run the Web application

Creating the bookDB JSP Page

Let's first create the bookDB.jsp file, which is the home page for the SqlTagApp Web application. This page shows the interaction with the Oracle database by using SQL tags, as shown in the code given in Listing 9.13 (you can find the bookDB.jsp file on the CD in the code\JavaEE\Chapter9\SqlTagApp folder):

Listing 9.13: Showing the Code of the bookDB.jsp File

```
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/sql_rt" prefix="sql" %>
<sql:setDataSource var="datasource"
  driver="oracle.jdbc.driver.OracleDriver"
  url="jdbc:oracle:thin:@192.168.1.123:1521:XE" user="scott" password="tiger"
  />
<sql:query var="books" dataSource="${datasource}">
  SELECT id, title, price FROM book
</sql:query>
<html>
  <head>
      <title>Accessing Database using JSTL</title>
  </head>
  <h2> Books Available in the Database</h2>
  <body>
      <table border="1">
      <tr>
      <td>id</td><td>title</td><td>price</td>
      </tr>
      <c:forEach varStatus="status" items="${books.rows}" var="row">
      <tr>
      <td><c:out value="${row.id}" /></td>
      <td><c:out value="${row.title}" /></td>
      <td><c:out value="${row.price}" /></td>
      </tr>
      </c:forEach>
```

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

In Listing 9.13, a connection to the Oracle database is created by using the `datasource` variable. In addition, the result of the SELECT statement is stored in the `books` variable. The value of a row is stored in the `rows` variable and is accessed by using the separator (.) along with the `rows` variable, followed by the name of the column. For example, the `rows.id` expression is used to retrieve the value of the id column.

Creating the Book Table in the Oracle Database

The `SqlTagApp` Web application establishes a connection with the book table in an Oracle database. The table structure of the book table for the Web application is shown in Table 9.25:

Table 9.25: Structure of the Book Table

Column Name	Data Type
ID	NUMBER
TITLE	VARCHAR2(30)
PRICE	VARCHAR2(30)

After defining the structure of the book table, let's move to the next step and configure the `SqlTagApp` application.

Configuring the SqlTagApp Application

We now configure the `SqlTagApp` application in the web.xml file. The code for web.xml is shown in Listing 9.14 (you can find the web.xml file on the CD in the `code\JavaEE\Chapter9\SqlTagApp\WEB-INF` folder):

Listing 9.14: Showing the web.xml File for the `SqlTagApp` Application

```
<?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">
<welcome-file-list>
   <welcome-file>bookDB.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

In Listing 9.14, the bookDB.jsp file is set as the welcome file for the SqlTagApp Web application.

Defining the Directory Structure of the SqlTagApp Application

You can create the directory structure for the `SqlTagApp` application, and arrange all the files of the application, as shown in Figure 9.11:

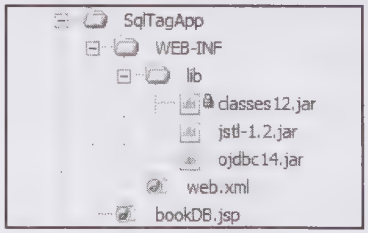


Figure 9.11: Showing the SqlTagApp Directory Structure

Packaging, Deploying, and Running the SqlTagApp Application

Perform the following steps to package, deploy, and run the `SqlTagApp` Web application:

1. Create the SqlTagApp.war file and deploy it on the GlassFish V3 application server.

2. Access the Web application by using the URL `http://localhost:8080/SqlTagApp`. The first page of the application is `bookDB.jsp`, as shown in Figure 9.12:

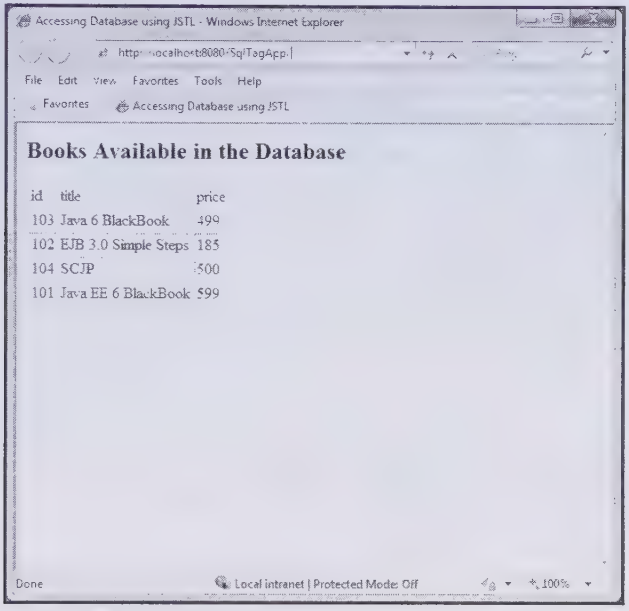


Figure 9.12: Showing the Use of SQL Tags

Let's now discuss the functions tag library.

Working with the Functions Tag Library

The functions tag library contains various functions to support tasks such as calculating the length of a collection or performing String manipulation. Let's learn about the functions available in the functions tag library in detail in the following sections.

Exploring the Functions Available in the Functions Tag Library

The functions tag library provides a set of functions that can be used with EL. The tags of the functions tag library use the `fn` prefix and allow you to perform String manipulations without using Java code within scriptlet tags.

Table 9. 26 shows the functions available in the functions tag library according to their category:

Table 9.26: Functions in the Functions Tag Library

Category	Functions
Collection length	`Length`

`String` manipulation	`toUpperCase, toLowerCase`
	`subString, subStringAfter, subStringBefore`
	`trim`
	`replace`
	`indexOf, startsWith, endsWith, contains, containsIgnoreCase`
	`split, join`
	`escapeXml`

Let's discuss each of these functional categories in detail.

Describing the Collection Length Function

The functions tag library provides a single collection length function called `fn:length`. The `fn:length` function can be applied to any collection to return the number of items in a collection. When applied to a String, the `fn:length` function returns the number of characters from the String.

The following code snippet shows the use of the `fn:length` function:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions"
prefix="fn" %>
<html>
<head><title>Hello</title></head>
<input type="text" name="username" size="25">
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>
<c:if test="${fn:length(param.username) > 0}" >
<%@include file="response.jsp" %>
</c:if>
</body>
</html>
```

In preceding code snippet, the `fn:length` function returns the value of the parameter named `username`. This value is checked against a condition specified by using the `<c:if>` tag. If the condition evaluates to true, another JSP page named response is included in the current JSP page.

Exploring the String Manipulation Functions

JSTL provides several useful functions, such as escapeXML and indexOf, to perform String manipulation in a JSP page. Table 9.27 shows various JSTL functions supporting String manipulation:

Table 9.27: JSTL Functions for String Manipulation

Function	Description
`escapeXml`	Returns the String values after escaping the characters that can be interpreted by using XML
`indexOf, startsWith, endsWith, contains, and containsIgnoreCase`	Verify whether or not a `String` contains another `String`
`Join`	Returns a String from an array that contains values separated by a separator, such as ; or ,

Let's now learn to implement the JSTL function in a Web application.

Using the JSTL Functions in the JSTLFunctionApp Application

Let's now create a Web application named `JSTLFunctionApp` to implement the tags of the functions tag library. The following are the broad-level steps to create the `JSTLFunctionApp` Web application:

- Create the JSP page
- Configure the Web application
- Define the directory structure of the Web application
- Package, deploy, and run the Web application

Creating the index JSP Page

Let's create a JSP page named index, which is a home page of the `JSTLFunctionApp` Web application. The index.jsp file demonstrates various JSTL functions, as shown in the code of Listing 9.15 (you can find the index.jsp file on the CD in the `code\JavaEE\Chapter9\JSTLFunctionApp` folder):

Listing 9.15: Showing the Code for the index.jsp File

```
<%@ page language="java" import="java.util.*" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
```

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head><title>example showing JSTL functions</title></head>
<body>
<h2>Using various JSTL 1.2 functions</h2>
<c:set var="var" value="Example showing usage of JSTL functions"/>
The length of the test String: ${fn:length(var)}<br />
Does the test String contain "JSTL"? ${fn:contains(var,"JSTL")}<br />
Putting the String into upper case using fn:toUpperCase(): ${fn:toUpperCase(var)}<br />
Splitting the String into a String array using fn:split(), and returning the array length:
  ${fn:length(fn:split(var," "))}<br />
</body>
</html>
```

Configuring the JSTLFunctionApp Application

Let's now configure the `JSTLFunctionApp` Web application in the web.xml file and set the index JSP page in the<welcome-file> tag. The code for the web.xml file is shown in Listing 9.16 (you can find the web.xml file on the CD in the `code\JavaEE\Chapter9\JSTLFunctionApp\WEB-INF` folder):

Listing 9.16: Showing the Code for the web.xml File for the `JSTLFunctionApp Application`

```
<?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">
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

Defining the Directory Structure of the JSTLFunctionApp Application

Create a directory structure as shown in Figure 9.13, and arrange the preceding files accordingly:

Figure 9.13: Showing the JSTLFunctionApp Directory Structure

Packaging, Deploying, and Running the JSTLFunctionApp Application

Perform the following steps to package, deploy, and run the JSTLFunctionApp application:

1. Create the JSTLFunctionApp.war file and deploy it on the GlassFish V3 application server.
2. Run the Web application using the URL `http://localhost:8080/JSTLFunctionApp`. The index JSP page of the application is shown in Figure 9.14:

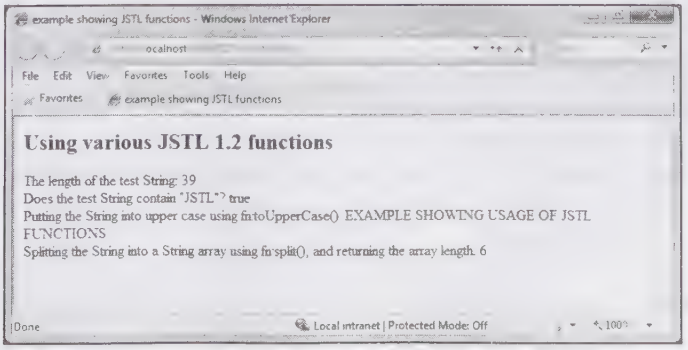


Figure 9.14: Showing an Output of a Test String

Figure 9.14 displays the length of the specified String and converts the text of the String in upper case by using tags of the functions tag library.

Let's now summarize the concepts discussed in the chapter.

Summary

The chapter has provided an introduction to JSTL and discussed its various features in detail. It has also discussed the various tag libraries supported by JSTL, including the core tag library, the XML tag library, the internationalization tag library, the SQL tag library, and the functions tag library. The chapter has also explained the process to create Web applications to demonstrate the use of these tag libraries.

In the next chapter, you will learn about the use of filters in Web applications.

Quick Revise

Q1. What is JSTL?

Ans. JSTL is a collection of tag libraries, which provides commonly needed functionalities, such as performing iteration, conditionals, database access, XML processing, and internationalization, in JSP pages.

Q2. The total number of libraries that JSTL provides is .....

A. 4
B. 3
C. 5
D. 6

Ans. The correct option is C.

Q3. What are JSTL core tags used for?

Ans. JSTL core tags are used to perform iteration and conditional processing, and to access URL-based resources.

Q4. How many flow control tags are available in the core tag library?

A. 2
B. 3
C. 4
D. 5

Ans. The correct option is D

Q5. What is the <x:transform> tag used for?

Ans. The <x:transform> tag applies a transformation to an XML document based on a specified XSLT.

Q6. What are JSTL XML tags used for?

Ans. JSTL XML tags are used to parse and transform the data used in a JSP page.

Q7. What are JSTL internationalization tags used for?

Ans. JSTL internationalization tags are used to format data such as dates, numbers, or time specifications in different domains.

Q8. What is the use of JSTL SQL tags?

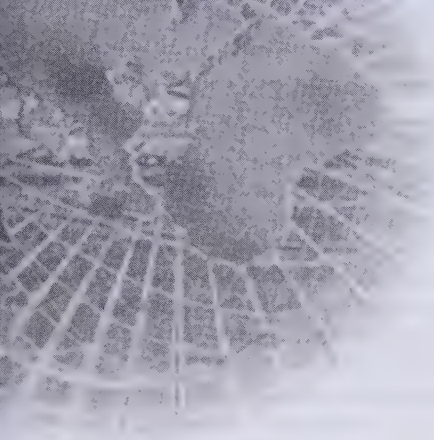
Ans. JSTL SQL tags are used to access the relational database specified in a JSP page.

Q9. What is the use of JSTL functions?

Ans. JSTL functions are used to support tasks such as calculating the length of a collection or performing String manipulation.

Q10. What is the fn:length function?

Ans. The `fn:length` function can be applied to any collection to return the number of items in a collection. When applied to a String, the `fn:length` function returns the number of characters from the String.



10

Implementing Filters

A filter is a Java class that is called for responding to the requests for resources, such as Java Servlet and JavaServer Pages (JSP). Filters dynamically change the behavior of a resource when a client requests the resource. In other words, the filter mapped to a resource, such as a Uniform Resource Locator (URL) or a servlet, is invoked when the resource is accessed. Filters intercept and process the requests before the requests are forwarded to servlets, and process the responses after the response has been generated by the servlet. Usually, a filter encapsulates and modifies the values of `request`, `response`, or header before or after the execution of the requested target resource. Filters need to be configured in order to serve client requests. Filters can also be put into a chain, where multiple filters can be invoked one after the other. A filter in a chain can either transfer the control to the next filter or redirect the request out of the chain to retrieve the requested resource.

Filters were introduced in Servlet 2.3 specification. The Servlet 3.0 specification defines APIs for filters ensuring that filters not only run on the application server that you develop it on, but also on any other application servers that follow the Servlet 3.0 specification.

In this chapter, you learn about filters and their use. Then, you learn to set up a development environment to create, configure, and test applications implementing filters. In addition, the chapter helps you to configure a filter using Deployment Descriptor as well as annotations. Moreover, you also learn how to define initializing parameters for filters. Next, the chapter helps you to manipulate the response using filters. Finally, the chapter discusses the issues raised by using threads with filters.

Exploring the Need of Filters

The need for implementing filters can be understood with the help of few examples. Let's take an example of a Web application that formats the data to be presented to clients in a specific format, say Excel. However, at a later point of time, the clients may require data in some other format, such as Hypertext Markup Language (HTML), Portable Document Format (PDF), or Word. In such a situation, instead of modifying the code every time to change the format of data, a filter can be created to transform data dynamically in the required formats.

Let's consider another example where a developer creates a Web application in which a servlet handles user logins. This implies that when a user submits his credentials, the servlet verifies the credentials against the user information. The servlet also creates a session for the user, so that other components in the application can also use the session details of the user. At a later point of time, the developer might require maintaining a login entry for each user login attempt in the application server's log system. In order to implement this, the developer would need to change the existing code or add additional code to the servlet and redeploy the Web application.

In such a situation, a servlet, besides fulfilling its primary objective that is to accept request and send responses to clients has to also implement additional functionalities. This additional load on the servlet reduces the efficiency of the application. To overcome this problem, filters were introduced that can implement these additional functionalities, such as verifying login credentials and maintaining the server log in a database. One of the most striking features of the filters is that they can be reused in other Web applications as well.

Some of the situations and tasks where filters can be used are as follows:

- Security verification
- Session validation
- Logging operations
- Internationalization
- Triggering resource access events
- Image conversion
- Scaling maps
- Data compression
- Encryption
- Tokenization
- Mime type changing

- Caching and XSL transformations of XML responses
- Debugging

Let's now learn how the filters work in a Web application.

Exploring the Working of Filters

When you send a request for a specific resource or a collection of resources, the request is intercepted by a filter. In order to properly intercept a request, a filter should have access to the HTTP request and response objects. The filter accesses the HTTP request and response objects by accessing the `javax.servlet.ServletRequest` and `javax.servlet.ServletResponse` objects. The filter also needs to access the list of chained filters that can be invoked in a sequence. To access the chained filters, the filter uses the `javax.servlet.FilterChain` object. After completing its work, a filter can call the next chained filter, block the request, throw an exception, or call up the originally requested resource.

After calling up the original resource, the control goes back to the last filter in the chain, which can inspect and modify the response headers and data, handle response, throw an exception, or call the filter before the last filter in the chain. The process carries on in the reverse order up through the filter chain.

Let's now discuss the Filter Application Programming Interface (API) that is used to create filters.

Exploring Filter API

The Filter API includes three interfaces:

- Filter
- FilterConfig
- FilterChain

These interfaces are present in the javax.servlet package. The following sections discuss each of these interfaces in detail.

The Filter Interface

A filter can be created in an application by implementing the `javax.servlet.Filter` interface, which is the basic interface defined in the Servlet 3.0 API.

The `Filter` interface calls the following methods during the life cycle of a filter:

- **The `init()` method**—Refers to the method that is invoked by the Web container only once when the filter is initialized. The servlet container passes the `FilterConfig` object as a parameter through the `init()` method.
- **The `doFilter()` method**—Refers to the method that is invoked each time a user requests a resource, such as a servlet to which the filter is mapped. When the `doFilter()` method is invoked, the servlet container passes the `ServletRequest`, `ServletResponse`, and `FilterChain` objects. The `FilterChain` object is passed to control the next object, if any.
- **The `destroy()` method**—Refers to the method that is invoked when the filter instance is destroyed.

Table 10.1 lists the syntax and working of all the preceding three methods of the `Filter` interface:

Table 10.1: Methods of the Filter Interface

Method	Syntax	Description
Init	public void init(FilterConfig filterConfig) throws ServletException	Initializes a filter and places it into service. A filter cannot be placed into service by the Web container if either a `ServletException` is thrown by the `init()` method or the method is not invoked within the time period specified by the Web container.
doFilter	public void doFilter(ServletRequest request, ServletResponse response,	Filters a request/response pair, whenever the pair is passed through a filter chain. The method takes an instance of the FilterChain interface as an argument,

Table 10.1: Methods of the Filter Interface

Method	Syntax	Description
	FilterChain chain)throws java.io.IOException, ServletException	which helps the filter to forward the request and response to the next filter in the chain. The doFilter() method encapsulates the actual logic of the filter. For example, this method can be implemented to perform the following tasks: • Examining the request • Carrying out input filtering by wrapping the request object with filter content or headers • Carrying out output filtering by wrapping the response object with filter content or headers • Invoking the next filter in the filter chain by calling the `FilterChain` object (`chain.doFilter()`) or blocking the request by not forwarding the request/response pair to the next filter in the filter chain • Setting the response headers can be done directly after invoking the next filter in the filter chain
destroy	public void destroy()	Removes the filter instance indicating that the filter is being moved out of service. This method is invoked only once for all threads within the doFilter() method. Once the destroy() method is called by the Web container, the doFilter() method cannot be called again on this filter instance. The destroy() method enables the filter to give up any resources being held, such as memory, file handles, or threads and ensures the synchronization of any persistent state with the current state of filter in memory.

The FilterConfig Interface

The `FilterConfig` interface is used to store the initialized data. The `init()` method of the `Filter` interface takes a filter configuration object as an argument, which is an instance of the `FilterConfig` interface. The filter receives filter configuration information during initialization from a servlet container through the `FilterConfig` object. The `getFilterName()`, `getInitParameter()`, `getInitParameterNames()`, and `getServletContext()` methods of the `FilterConfig` interface help in retrieving the filter name, initialization parameter values, and the reference to the `ServletContext`, respectively.

Table 10.2 describes the methods available in the `FilterConfig` interface:

Table 10.2: Methods of the FilterConfig Interface

Method	Syntax	Description
`getFilterName`	public java.lang.String getFilterName()	Returns the filter name as assigned in Deployment Descriptor.
`getInitParameter`	public java.lang.String getInitParameter(java.lang.String name)	Returns the named `initialization` parameter value as a `String`. A `null` value is returned if the parameter is not found. The `name` represents a `String` specifying the name of the `initialization` parameter.
`getInitParameterNames`	public java util.Enumeration getInitParameterNames()	Returns an `Enumeration` of `String` objects. The String objects in the Enumeration represent

Table 10.2: Methods of the FilterConfig Interface

Method	Syntax	Description
		the initialization parameter names for the filter. An empty Enumeration is returned, if no initialization parameters are found for the filter.
getServletContext	public ServletContext getServletContext()	Returns a reference of the ServletContext interface in which a filter is executed. It returns a ServletContext object, used by the filter to interact with the servlet container.

The FilterChain Interface

The FilterChain interface provides a mechanism to invoke a series of filters, which are specified in a filter chain, in an application. The filter chain instance is the instance of the class that implements the FilterChain interface.

The objects of the FilterChain interface are provided by the Web container to invoke the next filter in the chain of filters. In case the invoked filter is the last one in the chain, the target resource is invoked. The FilterChain interface specifies only the doFilter() method, which invokes the next filter or the targeted resource. The following code snippet shows the syntax of the doFilter() method:

```
public void doFilter(ServletRequest request,
ServletResponse response) throws java.io.IOException, ServletException
```

In the preceding syntax:

- The request parameter refers to the target resource request that is passed to the next filter in the chain
- The response parameter refers to the filter response that is forwarded to the target resource
- The doFilter() method causes the invocation of the next filter in the chain. The resource at the end of the chain is invoked in case the calling filter is the last chained filter

Let's now learn how to configure a filter.

Configuring a Filter

The web.xml file is used to configure filters to resources, such as servlet, JSP page, or Web application. This configuration helps to process the request and response objects. With the introduction of annotations in Servlet 3.0, filters can also be configured using annotations. Prior to the use of annotations, Deployment Descriptor was used for defining configuration and mapping for filters and servlets. The use of annotations eliminates the need of Deployment Descriptor. The mapping and configuration logic can be directly included within the filter class. In this section, you learn to:

- Configure a filter using Deployment Descriptor
- Configure a filter using Annotations

Configuring Filters Using Deployment Descriptor

You can configure filters as part of a Web application by using the application's web.xml Deployment Descriptor, which is located in the WEB-INF directory of the Web application. In Deployment Descriptor, you can declare and map the filter either to specific or multiple URL patterns or to servlets in the Web application. You can declare any number of filters and bind them with a specific pattern to any number of servlets or URL patterns. The <filter> and <filter-mapping> elements are used for configuring filters. The <filter> element must immediately succeed the <context-param> element and precede the <listener> and <servlet> elements in the web.xml file. The <filter> element declares a filter, defines a name for the filter, and specifies the Java class that executes the filter. One or more initialization parameters can also be specified inside the <filter> element by using the <init-param> element.

The following code snippet shows the code for adding a filter declaration in the `web.xml` file:

```
<filter>
  <filter-name>DemoFilter</filter-name>
  <display-name>demofilter</display-name>
  <description>This is my demo filter</description>
  <filter-class>com.kogent.DemoFilter</filter-class>
  <init-param>
        <param-name>InitParamName</param-name>
        <param-value>InitParamValue</param-value>
  </init-param>
</filter>
```

In the preceding code snippet, the `<filter-name>` element specifies the name of the filter and the `<filter-class>` element specifies the Java class that executes the filter. The description and display-name parameters are optional. The <init-param> element specifies the initialization parameters. `InitParamName` is the name of the initialization parameter specified by the <param-name> element and `InitParamValue` is the value of the initialization parameter specified by the <param-value> element. The filter class in a Web application can read the initialization parameters by using the `FilterConfig.getInitParameter()` or `FilterConfig.getInitParameterNames()` method.

The `<filter-mapping>` element defines the filter that is to be executed on the basis of the URL pattern or filter-name specified within the `<filter-mapping>` element. You need to specify the name of the filter and the URL pattern for creating a filter mapping using a URL pattern. The `<filter-mapping>` element must immediately succeed the `<filter>` element(s).

The following code snippet shows how to map a filter to a particular URL pattern:

```
<filter-mapping>
  <filter-name>DemoFilter</filter-name>
  <url-pattern>/myPattern/*</url-pattern>
</filter-mapping>
```

In the preceding code snippet, the DemoFilter filter is mapped to the requests containing the /myPattern/ URL pattern. According to Servlet 3.0 specification, the `<filter-mapping>` element supports multiple url patterns for a filter. The following code snippet shows mapping of a filter to multiple URL patterns:

```
<filter-mapping>
  <filter-name>DemoFilter</filter-name>
  <url-pattern>/myPattern/*</url-pattern>
  <url-pattern>/myDemoPatterns/*</url-pattern>
</filter-mapping>
```

In the preceding code snippet, the DemoFilter filter is mapped to the request containing /myPattern/ and /myDemoPattern/ URL patterns. Therefore, multiple URL patterns can be mapped to a single filter using the `<url-pattern>` element. Similarly, multiple filters can be mapped to a specific URL pattern. The filters can also be mapped to a specific servlet by mapping the filter to the name of the servlet that is registered in the Web application. The following code snippet shows the code for mapping a filter to the name of a servlet:

```
<filter-mapping>
  <filter-name>DemoFilter</filter-name>
  <servlet-name>DemoServlet</servlet-name>
</filter-mapping>
```

In the preceding code snippet, the DemoFilter filter is mapped to DemoServlet servlet. You can also map the filter to multiple servlets registered in a Web application, as shown in the following code snippet:

```
<filter-mapping>
  <filter-name>DemoFilter</filter-name>
  <servlet-name>DemoServlet</servlet-name>
  <servlet-name>MyServlet</servlet-name>
</filter-mapping>
```

In the preceding code snippet, the DemoFilter filter is mapped to the DemoServlet servlet and the MyServlet servlet. Apart from using Deployment Descriptor, filters can also be configured by using annotations.

Configuring Filters Using Annotations

With the introduction of annotations, the configuration code can be directly included in the code of the filter class. The @WebFilter annotation is used to mark a filter.

The following code snippet shows how a filter can be configured using annotations:

```
@WebFilter(filterName="DemoFilter", urlPatterns={"/myPattern/*"})
```

In the preceding code snippet, the filterName attribute defines the name of the filter and urlPatterns attribute defines the URL pattern which causes the invocation of the filter.

This completes the discussion about configuring filters. The next section focuses on the implementation of filters.

Creating a Web Application Using Filters

In this section, you learn how to implement and use filters by creating a simple Web application in which a login filter checks the login credentials entered by the user before the request for a servlet is made. Let's now create the FilterApp Web application in which the user logs in with a user name and password. The login credentials entered by the user are verified by a filter. If the user enters correct login credentials then the control is redirected to a servlet which displays a welcome message to the user; otherwise, an error message of invalid user credentials is displayed on the browser.

In *Chapter 5, Handling Sessions in Servlet 3.0*, you have created the `LoginApp` Web application, which contained a servlet to validate user credentials. In the `LoginApp` Web application, a client directly communicated with the `LoginServlet`, which was used for validating the user credentials. In the `FilterApp` Web application, the Web container invokes a filter to validate user credentials when the client requests for a resource. The Web container then invokes the target resource, such as a servlet if the user credentials are correct. A filter is used to ensure that the servlet is used only for handling requests and generating responses, while the validation is taken care of by the filter.

As discussed in the `Configuring a Filter` section of this chapter, filters can be configured by using Deployment Descriptor, as well as annotations. In this section, you learn to create the `FilterApp` Web application using both the ways. First, create the application using Deployment Descriptor to configure filters. Then, recreate the `FilterApp` Web application by using annotations to mark filters.

Using Deployment Descriptor to Configure a Filter

Let's now create the FilterApp Web application by using Deployment Descriptor for configuring filters used in the application. The `FilterApp` Web application includes filters for verifying user credentials, servlets for testing filters, and a home page for submitting user credentials.

Perform the following steps to create and run the FilterApp application:

- Create a generic filter
- Write the code for the first filter
- Creating the home page
- Creating a servlet to test the filter
- Configuring the filter and the servlet
- Running the FilterApp Web application

Creating a Generic Filter

A filter is created by implementing the `Filter` interface. However, instead of implementing the `Filter` interface for every filter that you create, you can create a generic filter for all the filters in a Web application. The generic filter can hold all the needed methods of the `Filter` interface. The other filters can access the methods of a generic filter by implementing the generic filter in an application.

Let's create the `MyGenericFilter` filter for the `FilterApp` Web application. The generic filter implements the `Filter` interface and defines all the necessary methods of the `Filter` interface. Instead of implementing the `Filter` interface, every time while creating a new filter, you can extend this generic filter and can overwrite the required methods. Apart from creating a filter, you also need to configure the filter in the web.xml file.

Listing 10.1 provides the code for the MyGenericFilter filter (you can find the MyGenericFilter.java file on the CD in the code\JavaEE\Chapter5\FilterApp\src\com\kogent\filter folder):

Listing 10.1: Showing the Code for the MyGenericFilter.java File

```
package com.kogent.filter;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MyGenericFilter implements Filter
{
   private FilterConfig filterconf = null;
   public void doFilter(final ServletRequest req, final ServletResponse res,
   FilterChain chain) throws IOException, ServletException
   {
         chain.doFilter(req,res);
   }

   public FilterConfig getFilterConfig()
   {
         return filterconf;
   }

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

   public void init(FilterConfig filterconf)
   {
         this.filterconf = filterconf;
   }

   public void destroy()
   {
         this.filterconf = null;
   }
}
```

In Listing 10.1, the MyGenericFilter class implements the Filter interface. Each instance of a generic filter has an instance variable that can hold the FilterConfig interface object to represent the configuration details of a filter. In the MyGenericFilter class, the filterconf variable holds the instance of the FilterConfig interface. As shown in Listing 10.1, the instance of the FilterChain interface is passed as an argument to the doFilter() method to call the next filter in the filter chain. When the Web container invokes the doFilter() method of the MyGenericFilter filter and if there are no more filters available in the chain, the Web container invokes the target resource. The getFilterConfig()and setFilterConfig() methods are utility methods that retrieve and set the filter configuration details, respectively. The init() method is invoked by the Web container during initialization of a filter. The destroy() method is called by the Web container when the current filter instance needs to be removed from service.

Save the MyGenericFilter.java file in the src\com\kogent\filter directory of the FilterApp Web application. Compile your generic filter. After compilation, the MyGenericFilter.class file should be located in the FilterApp\WEB-INF\classes\com\kogent\filter directory.

Writing Your First Filter

In order to create a filter, you first create a class, MyFilter, which is a Java class implementing the MyGenericFilter interface. As discussed in the Filter interface, the following methods must be implemented while writing a filter:

- public void init(FilterConfig config)
- public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
- public void destroy()

The `init()` method is called by the Web container to indicate that the `MyFilter` filter has been placed into service. The `doFilter()` method declared in the `MyFilter` class examines the request, verifies the user credentials, and finally invokes the targeted servlet. The `destroy()` method is invoked when the servlet is taken out of service. The `MyFilter` filter inherits the `init()` and `destroy()` methods from the `MyGenericFilter` Filter and overrides the `doFilter()` method.

Listing 10.2 shows the code for the `MyFilter` class (you can find the `MyFilter.java` file on the CD in the `code\JavaEE\Chapter10\FilterApp\src\com\kogent\filter` folder):

Listing 10.2: Showing the Code for `MyFilter.java` File

```
package com.kogent.filter;

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

public class MyFilter extends MyGenericFilter
{
  public void doFilter(ServletRequest req, ServletResponse res,
  FilterChain chain) throws IOException, ServletException
  {
                String password =
                ((HttpServletRequest) req).getParameter("password");
                if(password.equals("mypassword"))
        {
                        String uri = ((HttpServletRequest)req).getRequestURI();
                        chain.doFilter(req, res);
        }
        else
        {
                        res.setContentType("text/html");
                PrintWriter pw = res.getWriter();
                        pw.println("<html>");
                        pw.println("<head><title>Wrong Password</title></head>");
                        pw.println("<body>");
                        pw.println("<h3>Sorry, the password was incorrect.</h3>");
                        pw.println("</body>");
                        pw.println("</html>");
        }
  }
}
```

In Listing 10.2, the `doFilter()` method retrieves the password entered by the user and checks whether the password entered is `mypassword`. If the password entered is correct, the `MyFilter` filter invokes the targeted servlet; otherwise, an error message is displayed to the user indicating that the user has entered an incorrect password.

Save the `MyFilter.java` file at the `FilterApp\src\com\kogent\filter` location. Compile the `MyFilter.java` file. After compilation, the class file should be located in the `\FilterApp\WEB-INF\classes\com\kogent\filter\` directory.

Creating the Home Page

In the FilterApp application, let's create the `index.html` file used to access a servlet to test the MyFilter filter.

Listing 10.3 shows the code for the index.html file (you can find this file on the CD in the code\JavaEE\Chapter10\FilterApp folder):

Listing 10.3: Showing the Code for the index.html File

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
        <head>
```

```
        <title>Login Application Using Filters </title>
        <link rel="stylesheet" href="mystyle.css" type="text/css"/>
        </head>
<BODY>
        <h1>Login Application Using Filters </h1>
  <form action="/FilterApp/WelcomeServlet">
  <table>
  <tr>
        <td>User Name</td>
        <td>
        <input type="text" name="username"/>
        </td>
  </tr>
  <tr>
        <td>Password</td>
        <td><input type="password" name="password"/>
        </td>
  </tr>
  <tr>
        <td></td>
        <td><input type="submit" value="Submit"/></td>
  </tr>
  </table>
  </form>
</BODY>
</HTML>
```

After creating the index HTML page, save the index.html file in the root directory of the FilterApp application.

Let's now learn to create the WelcomeServlet servlet, which invokes the MyFilter filter during initialization.

Creating a Servlet to Test the Filter

The WelcomeServlet servlet receives users' requests and sends the desired responses. When the WelcomeServlet servlet is initialized, the MyFilter filter is invoked. The MyFilter filter verifies the user credentials and then invokes the target servlet, WelcomeServlet.

The WelcomeServlet servlet displays a welcome message to the user if the password entered is mypassword.

Listing 10.4 shows the code for the WelcomeServlet.java file (you can find this file on the CD in the code\JavaEE\Chapter10\FilterApp\src\com\kogent\servlets folder):

Listing 10.4: Showing the Code for WelcomeServlet.java File

```
package com.kogent.servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
public class WelcomeServlet extends HttpServlet
{
        @Override
  public void doGet(HttpServletRequest req,HttpServletResponse res)
  throws ServletException, IOException
  {
                res.setContentType("text/html");
                PrintWriter out = res.getWriter();
                String username = req.getParameter("username");
                out.println("<html><body>Welcome:<b>" + username + "<br/><br/>");
                out.println(new Date().toString());
                out.println("</b></body></html>");
  }
}
```

Save the WelcomeServlet.java file at the \FilterApp\src\com\kogent\servlets location. Now, compile the WelcomeServlet.java file. After compilation, the class file should be located in the \FilterApp\WEB-INF\classes\com\kogent\servlet\ directory.

Configuring the Filter and the Servlet

In order to execute the FilterApp application, you are required to provide a mapping for the filter in the web.xml file, which is located in the FilterApp\WEB-INF directory. Listing 10.5 provides the code for mapping the MyFilter filter (you can find the MyFilter.java file on the CD in the code\JavaEE\Chapter10\FilterApp\WEB-INF folder):

Listing 10.5: 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">

  <filter>
        <filter-name>MyFilter</filter-name>
        <filter-class>com.kogent.filter.MyFilter</filter-class>
  </filter>

  <filter-mapping>
        <filter-name>MyFilter</filter-name>
        <servlet-name>WelcomeServlet</servlet-name>
  </filter-mapping>

  <servlet>
        <servlet-name>WelcomeServlet</servlet-name>
        <servlet-class>com.kogent.servlets.WelcomeServlet</servlet-class>
  </servlet>

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

  <session-config>
        <session-timeout>
        30
        </session-timeout>
  </session-config>

</web-app>
```

In Listing 10.5, the MyFilter filter is mapped to the WelcomeServlet servlet class. Save the web.xml file in the WEB-INF directory of the FilterApp Web application.

Let's now discuss the directory structure of the FilterApp Web application.

Exploring the Directory Structure of FilterApp Application

All the files for the application are saved under a base directory named as FilterApp. Create the FilterApp directory under the chapter number directory. You should note that the FilterApp directory is located under the C:\JavaEE\Chapter10 folder. Create the remaining required folders as shown in the directory structure displayed in Figure 10.1. Place the respective files accordingly at proper locations in the directory structure as described by the following statements:

- All packages containing class files are placed in FilterApp\WEB-INF\classes folder
- The configuration file, such as web.xml is placed into FilterApp\WEB-INF\ folder
- All source files (.java files) for filters can be placed in FilterApp\src\com\kogent\filter folder
- All source files (.java files) for servlets can be placed in FilterApp\src\com\kogent\servlets folder

Figure 10.1 displays the directory structure of the FilterApp application:

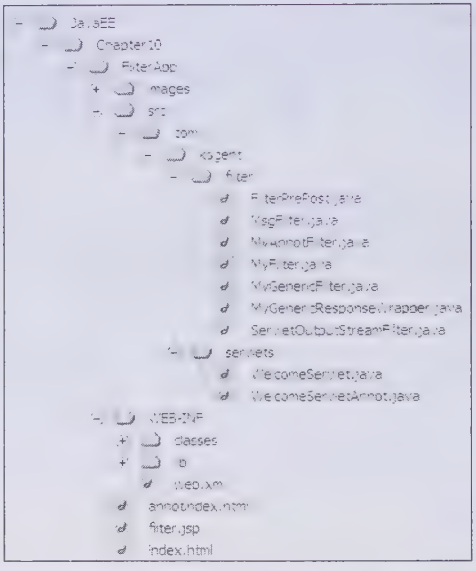


Figure 10.1: Displaying the Root Directory Structure for FilterApp Web Application

As shown in Figure 10.1, `FilterApp` is the root folder containing `WEB-INF` folder, `src` folder, and `index.html` file. The `WEB-INF` folder has two folders, `classes` and `lib`, and a file called `web.xml`. As discussed earlier, the package containing the .class files is placed at the WEB-INF\classes location under the `FilterApp` directory. The `src\com\kogent` is the optional folder containing source files. The configuration file, `web.xml`, is placed in the WEB-INF folder.

NOTE

The additional files, such as annotindex.html and filter.jsp as well as other Java source files have been created later in the chapter.

You also need to configure the compiler. Ensure that your compiler has all the necessary Java ARchive (JAR) files in its `CLASSPATH`, and that it can compile all Java files under \FilterApp\WEB-INF\classes directories.

Running the FilterApp Web Application

After creating all the required files, let's package and deploy the `FilterApp` application to test the filter. Now, navigate to the `http://localhost:8080/FilterApp/index.html` URL to view the output of the `FilterApp` application. The browser displays a login page as shown in Figure 10.2, in which the user needs to enter the username and password:

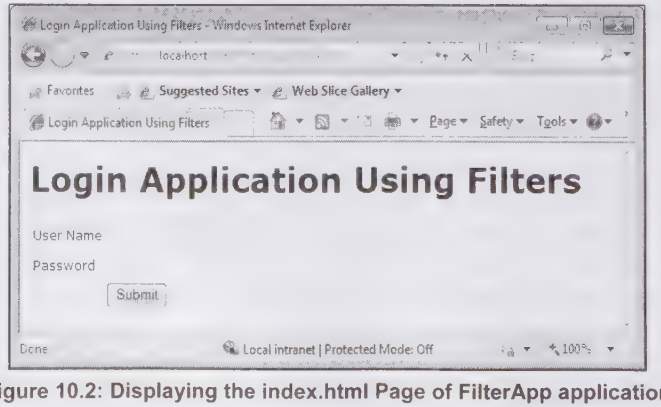


Figure 10.2: Displaying the index.html Page of FilterApp application

If the password entered by the user is `mypassword`, the Web container invokes the `WelcomeServlet` servlet and displays a welcome message to the user, as shown in Figure 10.3:

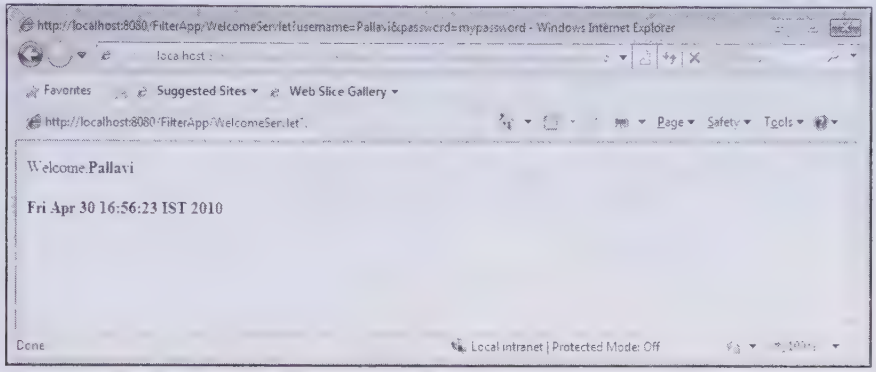


Figure 10.3: Displaying the Output of WelcomeServlet Servlet

In case the password is not correct, an error message is displayed, as shown in Figure 10.4:

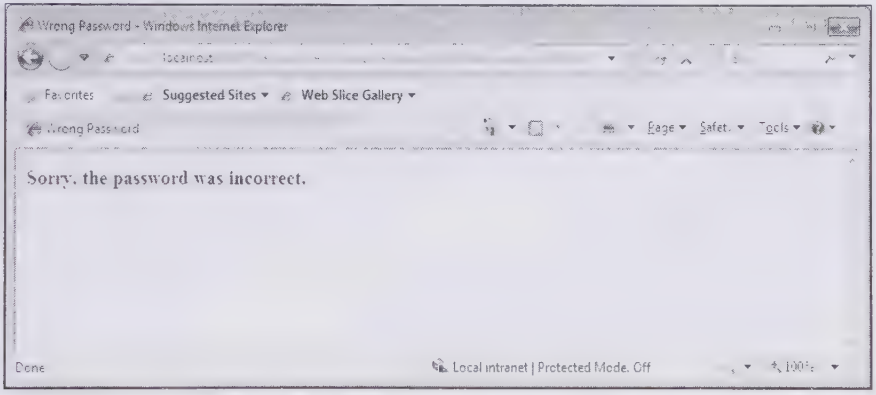


Figure 10.4: Displaying the Output of WelcomeServlet Servlet When Password is Incorrect

In this section, you learned to create a filter using Deployment Descriptor.

Let's now learn to create a filter using annotations.

Using Annotations to Configure a Filter

Starting from Servlet API 3.0, annotations can be used for marking filters rather than making an entry for mapping filters in Deployment Descriptor. Let's recreate the `MyFilter` filter created in the `FilterApp` application using annotations in place of Deployment Descriptor.

Creating a Filter using Annotations

To create a filter using annotations, you need to first create a class, `MyAnnotFilter`, which is a Java class implementing the `MyGenericFilter` interface. The `MyAnnotFilter` class is similar in functionality to the `MyFilter` class. The only difference between the two is that `MyAnnotFilter` class also includes the code for annotations.

Listing 10.6 shows the code for the `MyAnnotFilter` class (you can find the `MyAnnotFilter.java` file on the CD in the `code\JavaEE\Chapter10\FilterApp\src\com\kogent\filter` folder):

Listing 10.6: Showing the Code for `MyAnnotFilter.java` File

```
package com.kogent.filter;

import java.util.*;
import java.io.*;
```

```
import javax.servlet.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;

@WebFilter(filterName = "MyAnnotFilter", urlPatterns={"/WelcomeServletAnnot"})
public class MyAnnotFilter extends MyGenericFilter
{

   public void doFilter(ServletRequest req, ServletResponse res,
   FilterChain chain) throws IOException, ServletException
   {
                String password =
                ((HttpServletRequest) req).getParameter("password");
                if(password.equals("mypassword"))
        {
                        String file = ((HttpServletRequest)req).getRequestURI();
                        chain.doFilter(req, res);
        }
        else
        {
                        res.setContentType("text/html");
                PrintWriter pw = res.getWriter();
                        pw.println("<html>");
                        pw.println("<head><title>Wrong Password</title></head>");
                        pw.println("<body>");
                        pw.println("<h3>Sorry, the password was incorrect.</h3>");
                        pw.println("</body>");
                        pw.println("</html>");
        }
   }
}
```

In Listing 10.6, the `@WebFilter` annotation provides the required mapping between the filter and the `WelcomeServlet` servlet. Save the `MyAnnotFilter.java` file at the `FilterApp\src\com\kogent\filter` location and compile the `MyAnnotFilter` Java class.

Creating the Home Page

In order to test the `MyAnnotFilter` filter, create an HTML page that can access servlets. Let's name this page as the `annotindex.html`.

Listing 10.7 shows the code for the `annotindex.html` file (you can find this file on the CD in the `code\JavaEE\Chapter10\FilterApp` folder):

Listing 10.7: Showing the Code for annotindex.html Page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
        <head>
        <title>Login Application Using Filters </title>
        <link rel="stylesheet" href="mystyle.css" type="text/css"/>
        </head>
<BODY>
        <h1>Login Application Using Filters</h1>
  <p>The filters applied in this login application use annotations</p>
  <form action="/FilterApp/WelcomeServletAnnot">
  <table>
  <tr>
        <td>User Name</td>
        <td>
        <input type="text" name="username"/>
        </td>
  </tr>
  <tr>
        <td>Password</td>
```

```
            <td><input type="password" name="password"/>
            </td>
        </tr>
        <tr>
            <td></td>
            <td><input type="submit" value="Submit"/></td>
        </tr>
        </table>
        </form>
    </BODY>
    </HTML>
```

After creating the annotindex HTML page and save the annotindex.html file in the root directory, FilterApp. Let's now recreate the WelcomeServlet servlet and name it WelcomeServletAnnot, which invokes the MyAnnotFilter filter during initialization.

Creating a Servlet to Test the Filter

The WelcomeServletAnnot servlet receives users' requests and sends the desired responses. When WelcomeServletAnnot is initialized, the MyAnnotFilter filter is invoked, which verifies the user credentials and invokes the target servlet, WelcomeServletAnnot. The WelcomeServletAnnot servlet displays a welcome message to the user if the password entered is mypassword.

Listing 10.8 shows the code for WelcomeServletAnnot.java file (you can find this file on the CD in the code\JavaEE\Chapter10\FilterApp\src\com\kogent\servlets folder):

Listing 10.8: Showing the Code for WelcomeServletAnnot.java File

```
package com.kogent.servlets;

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

public class WelcomeServletAnnot extends HttpServlet
{
        @Override
    public void doGet(HttpServletRequest req,HttpServletResponse res)
    throws ServletException, IOException
    {
                res.setContentType("text/html");
                PrintWriter out = res.getWriter();
                String username = req.getParameter("username");
                out.println("<html><body>Welcome:<b>" + username + "<br/><br/>");
                out.println(new Date().toString() + "</b>");
                out.println("<p> This servlet invokes the filter that uses
                annotations in place of a deployment descriptor for
                marking the filter</p>");
                out.println("</body></html>");
    }
}
```

Save the WelcomeServletAnnot.java file at the FilterApp\src\com\kogent\servlets location and compile the WelcomeServletAnnot.java file.

You need to provide the mapping for the WelcomeServletAnnot Servlet to execute the Web application. Map the WelcomeServletAnnot servlet to the /WelcomeServletAnnot URL pattern in the web.xml file. The following code snippet shows the mapping for the WelcomeServletAnnot servlet:

```
<servlet>
    <servlet-name>
        WelcomeServletAnnot
    </servlet-name>
    <servlet-class>
        com.kogent.servlets.WelcomeServletAnnot
    </servlet-class>
</servlet>
```

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

Include the mapping shown in the preceding code snippet for the `WelcomeServletAnnot` servlet in the `web.xml` file for the `FilterApp` Web application. Apart from servlets, you also need to configure filters in the web.xml file. However, as we are using annotations, the mapping for the filter is not required to be included in the web.xml file.

To test the `MyAnnotFilter` filter, package and deploy the `FilterApp` application and navigate to the `http://localhost:8080/FilterApp/annotindex.html` URL.

The browser displays a login page as shown in Figure 10.5, in which the user needs to enter the username and password:

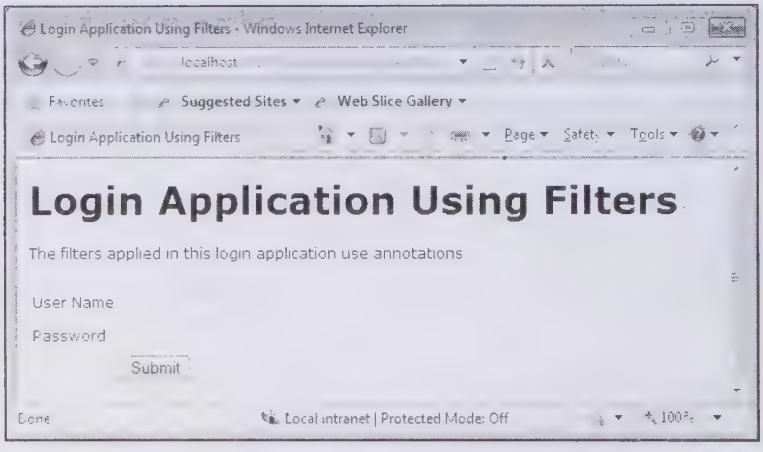


Figure 10.5: Displaying the annotindex.html Page

If the password entered by the user is `mypassword`, the Web container invokes the `WelcomeServletAnnot` servlet and displays a welcome message to the user, as shown in Figure 10.6:

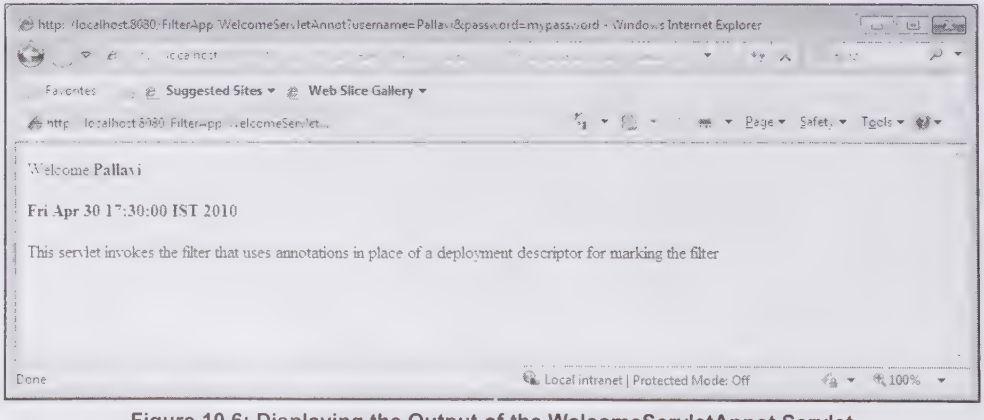


Figure 10.6: Displaying the Output of the WelcomeServletAnnot Servlet

This completes the discussion about creating filters using annotations. As you know, certain parameters can be provided for servlets that are initialized at the time of initialization of servlets. Such parameters are known as initializing parameters and are used to provide certain values to be used in an application. Similar to servlets, you can also provide initializing parameters to filters in the web.xml file. Let's now learn to create filters using initializing parameters in the next section.

Using Initializing Parameter in Filters

In this section, you learn to develop a simple filter that uses an initializing parameter. The name and value of an initializing parameter is specified in Deployment Descriptor. An initializing parameter helps a filter to retrieve a value to be used in that filter. You should note that the initializing parameter is defined during the initialization of the filter.

Let's create a filter, `MsgFilter`, which defines the `message` initializing parameter declared in the web.xml file of the `FilterApp` application. Further, you need to create the filter.jsp file to display the value of the `message` parameter. In order to invoke the `MsgFilter` filter with filter.jsp, you need to map the `MsgFilter` filter to `filter.jsp` in the `web.xml` file. Finally, you can run the `FilterApp` application to test the `MsgFilter` filter.

Creating the MsgFilter Filter

Let's create the `MsgFilter` filter that provides the value of the initializing parameter `message` to the `Request` scope before the target resource is called. Listing 10.9 shows the code for `MsgFilter.java` file (you can find this file on the CD in the `code\JavaEE\Chapter10\FilterApp\src\com\kogent\servlets` folder):

Listing 10.9: Showing the Code for `MsgFilter.java` File

```
package com.kogent.filter;

import javax.servlet.FilterChain;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.FilterConfig;

public class MsgFilter extends MyGenericFilter
{
  private FilterConfig filterConfig = null;

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

  public void doFilter(final ServletRequest req,
  final ServletResponse res, FilterChain chain) throws
  java.io.IOException, javax.servlet.ServletException
  {
        System.out.println("Entering Filter");
        JOptionPane.showMessageDialog(null," Entering Filter!");
        String message=filterConfig.getInitParameter("message");
        req.setAttribute("message",message);
        chain.doFilter(req,res);
        System.out.println("Exiting Filter");
        JOptionPane.showMessageDialog(null," Exiting Filter!");
        }
}
```

As shown in Listing 10.9, when the Web container invokes the `MsgFilter` filter, the `Entering Filter` message is displayed on the server console as well as in a dialog box. Then, the `filterConfig` instance retrieves the value of the `message` parameter, and the retrieved value is set to `message` attribute. Then, the `doFilter()` method forwards the request to the next filter in the chain. The `Exiting Filter` message is displayed in a message box as the filter exits. Save the `MsgFilter.java` file at the `FilterApp\src\com\kogent\filter` location. After saving the `MsgFilter.java` file, compile the `MsgFilter.java` file. The package containing the class file should be located at the `FilterApp\WEB INF\classes` folder.

Creating a JSP Page to Test the Filter

Let's now create a JSP page, `filter` and place it in the root directory of the `FilterApp` application. This JSP page is used to configure and test the `MsgFilter` filter.

Listing 10.10 provides the code for the filter.jsp file (you can find the filter.jsp file on the CD in the code\JavaEE\Chapter10\FilterApp folder):

Listing 10.10: Showing the Code for the filter.jsp File

```
<HTML>
  <HEAD>
        <TITLE>Implementing Filters</TITLE>
        <link rel="stylesheet" href="mystyle.css" type="text/css"/>
  </HEAD>
  <BODY>
        <HR>
        <P><%=request.getAttribute("message")%></P>
        <P>Check your console output!</P>
        <HR>
  </BODY>
</HTML>
```

In Listing 10.10, the request implicit object retrieves the value of the message attribute set in the MsgFilter class. The Web container invokes the MsgFilter filter when the filter JSP page requests the message attribute. The MsgFilter class retrieves the value of an initializing parameter and sets the value to the message attribute. Then, the request is sent back to the target resource, filter.jsp, and the value of the message attribute is displayed.

Configuring the Filter

In order to test the MsgFilter filter, you need to configure it first. So, let's now configure the MsgFilter filter and map it to the filter JSP page. The following code snippet shows the mapping for the MsgFilter filter to be added in the web.xml file:

```
<filter>
  <filter-name>messageFilter</filter-name>
  <filter-class>com.kogent.filter.MsgFilter</filter-class>
  <init-param>
        <param-name>message</param-name>
        <param-value>A message to you!</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>messageFilter</filter-name>
  <url-pattern>/filter.jsp</url-pattern>
</filter-mapping>
```

In the preceding code snippet, messageFilter is the name specified for the MsgFilter filter. The initializing parameter message is initialized with the parameter value A message to you!. When the Web container initializes the MsgFilter filter, the message parameter is initialized with its value and can be accessed by the MsgFilter filter. The MsgFilter filter is mapped to the filter JSP page. The Web container first invokes the filter named messageFilter for any requests for the filter JSP page.

The preceding code snippet shows the filter mapping in the web.xml file. Save the web.xml file after incorporating the necessary changes.

Let's now learn how to test a filter.

Testing a Filter

Let's now test the filter by running the FilterApp application. In order to test the filter, first ensure that the Glassfish V3 application server is running. Type the http://localhost:8080/FilterApp/filter.jsp URL in a browser. The result should be a page displaying the message, A message to you! accompanied by the instruction to view the console for any output. The MsgFilter filter is first invoked when the request is forwarded to the filter JSP page, and retrieves the initialization parameter value for setting the message attribute.

Figure 10.7 shows the output when http://localhost:8080/FilterApp/filter.jsp URL is entered in the browser:

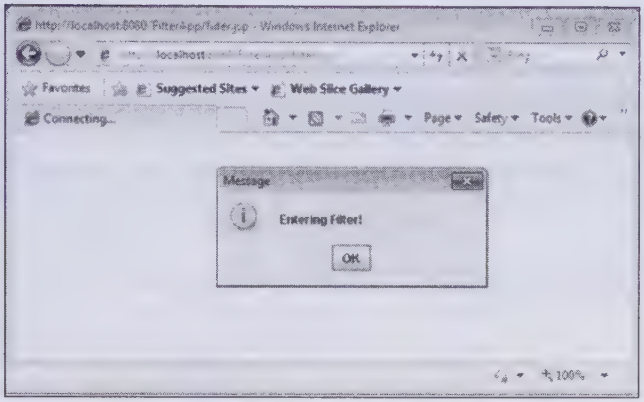


Figure 10.7: Showing the Output Indicating that Filter has Started Processing the Request

As you click the `OK` button, another dialog box is displayed indicating that the `MsgFilter` filter has completed its execution. The `Exiting Filter` message is displayed, as shown in Figure 10.8:

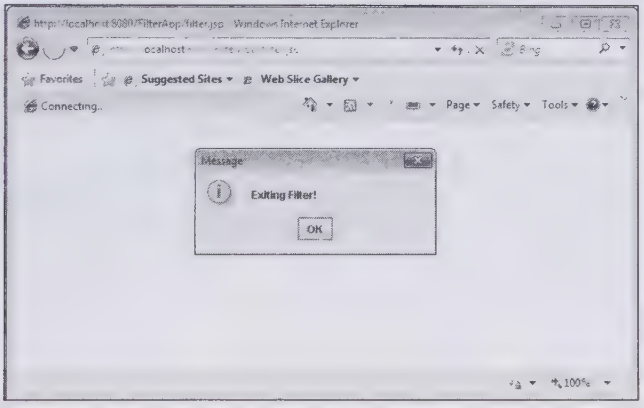


Figure 10.8: Showing the Output Indicating that Filter has Completed Processing the Request

Once the filter completes processing, the request is redirected to the `filter JSP` page. The `filter` JSP page retrieves the value of the `message` attribute and displays it on the browser, as shown in Figure 10.9:

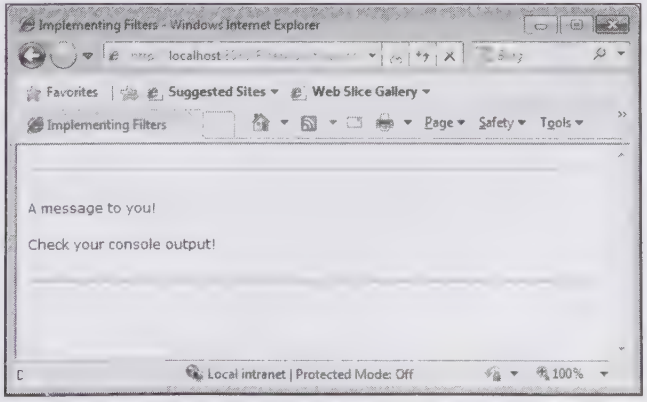


Figure 10.9: Showing the Output Indicating that Filter has Completed Processing the Request

This completes the discussion about creating filters.

Let's now discuss about manipulating responses in the next section.

Manipulating Responses

In this section, you learn to create a filter that can manipulate the responses to a request. In order to do so, you need the `HttpServletResponseWrapper` class, a custom `ServletOutputStream` class, and a filter. You also need a JSP page to attach the filter for testing it. You should then configure the filter, and finally test it.

The filter uses the `HttpServletResponseWrapper class` to wrap the `response` object before it is sent to the target resource. The `HttpServletResponseWrapper` object acts as a `wrapper` for the `ServletResponse` object. It helps in wrapping the response, so that the response can be modified after being delivered by the target resource.

The HttpServletResponseWrapper class uses a custom ServletOutputStream `class that` allows you to manipulate the response after the target resource has generated the response. The response generated cannot be modified if the `ServletOutputStream` has been closed. The ServletOutputStream class manipulates the response.

Creating the ServletOutputStreamFilter Class

Let's now create the `ServletOutputStreamFilter` class. Listing 10.11 shows the code for `ServletOutputStreamFilter` class (you can find the ServletOutputStreamFilter.java file on the CD in the `code\JavaEE\Chapter10\FilterApp\src\com\kogent\filter` folder):

Listing 10.11: Showing the Code for the ServletOutputStreamFilter Class

```
package com.kogent.filter;

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

public class ServletOutputStreamFilter extends ServletOutputStream
{
        DataOutputStream datastream;

  public ServletOutputStreamFilter(OutputStream out)
        {
        datastream=new DataOutputStream(out);
        }

  public void write(int num) throws IOException
        {
        datastream.write(num);
  }

        @Override
  public void write(byte[] num) throws IOException
        {
        datastream.write(num);
        }

        @Override
  public void write(byte[] num, int of, int ln) throws IOException
        {
        datastream.write(num, of, ln);
        }
}
```

In Listing 10.11, the `ServletOutputStreamFilter` class extends the `ServletOutputStream` class. To hold a `DataOutputStream that needs` to be written on the browser, the datastream instance variable is created. Whenever the constructor of the `ServletOutputStreamFilter` class is called, the `out` parameter is cast to a `DataOutputStream` and is stored in the `datastream` object. Compile the ServletOutputStreamFilter class. After compilation, the ServletOutputStreamFilter.class file should be located in the `\FilterApp\WEB-INF\classes\com\kogent\filter\`directory.

Now, after creating the ServletOutputStreamFilter class, you are going to create the MyGenericResponseWrapper class, which uses the ServletOutputStream class.

Creating the MyGenericResponseWrapper Class

To use the `ServletOutputStream` class, you need to implement a class that can act as a `response` object. The instance of the `MyGenericResponseWrapper` class is sent to the target resource instead of the original `response` object. You implement some utility methods in the `MyGenericResponseWrapper` class to retrieve the content type and content length for the content it holds.

Listing 10.12 shows the code for the `MyGenericResponseWrapper` class (you can find the MyGenericResponseWrapper.java file on the CD in the `code\JavaEE\Chapter10\FilterApp\src\com\kogent\filter` folder):

Listing 10.12: Showing the Code for the MyGenericResponseWrapper Class

```
package com.kogent.filter;

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

public class MyGenericResponseWrapper extends HttpServletResponseWrapper
{
   private ByteArrayOutputStream outstream;
         private int contentLen;
         private String contentTyp;

   public MyGenericResponseWrapper(HttpServletResponse res)
         {
         super(res);
         outstream = new ByteArrayOutputStream();
         }

         @Override
   public ServletOutputStream getOutputStream()
         {
         return new ServletOutputStreamFilter(outstream);
         }

   public byte[] getData()
         {
         return outstream.toByteArray();
         }

   @Override
   public PrintWriter getWriter()
   {
                return new PrintWriter(getOutputStream(), true);
   }

         @Override
   public void setContentType(String type)
         {
         this.contentTyp = type;
         super.setContentType(type);
         }

   @Override
   public String getContentType()
         {
         return this.contentTyp;
         }

   public int getContentLength()
         {
```

```
            return contentLen;
            }
            @Override
      public void setContentLength(int len)
            {
            this.contentLen=len;
            super.setContentLength(len);
            }
}
```

In Listing 10.12, the MyGenericResponseWrapper class extends the HttpServletResponseWrapper class and declares three instance variables: outstream, contentLen, and contentTyp. The outstream variable is an instance of the ByteArrayOutputStream class that holds any content written by the target resource, and the contentLen variable is an int type variable declared to hold the content length. The contentTyp variable is defined to hold the content type. The getOutputStream() method handles the binary content. The method will be used by the target resource when writing its binary response. The getData() method is declared to retrieve the content of the response. The getWriter() method handles the character content. This method is used by the target resource to write the character text response. The setContentType() and getContentType() methods handle the content type. The set and get methods are used to set or retrieve the content type of the response. These methods have been overridden in Listing 10.12 to make sure that the content type values can be obtained later. Similarly, the get and set methods are provided to set or retrieve the content-length of the response. Compile the MyGenericResponseWrapper.java file. After compilation, the MyGenericResponseWrapper.class file should be located in the \FilterApp\WEB-INF\classes\com\kogent\filter\ directory.

Creating the Filter

Let's create a filter that uses the MyGenericResponseWrapper class. This filter adds content to the response of the target resource before and after the target is invoked. Listing 10.13 shows the code for the FilterPrePost class (you can find the FilterPrePost.java file on the CD in the code\JavaEE\Chapter10\FilterApp\src\com\kogent\filter folder):

Listing 10.13: Showing the Code for the FilterPrePost.java File

```
package com.kogent.filter;

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

public class FilterPrePost extends MyGenericFilter
{
   public void doFilter(final ServletRequest req, final ServletResponse res,
   FilterChain chain) throws IOException, ServletException
   {
         OutputStream outstream = res.getOutputStream();
         outstream.write("<HR>PRE<HR>".getBytes());

         MyGenericResponseWrapper wrap =
         new MyGenericResponseWrapper((HttpServletResponse) res);
         chain.doFilter(req, wrap);
         outstream.write(wrap.getData());
         outstream.write("<HR>POST<HR>".getBytes());
         outstream.close();
         }
}
```

In Listing 10.13, FilterPrePost is a Java class that extends the MyGenericFilter class. The instance of the MyGenericResponseWrapper class is created which invokes the parameterized constructor. The instance of the MyGenericResponseWrapper class is passed to the next filter in the chain. Then the response received is written to the ServletOutputStream class by using the getData() wrapper method, which contains the response of the filter that is previously called. Compile the FilterPrePost.java file and place it in the \FilterApp\WEB-INF\classes\com\kogent\filter\ directory.

Creating the Servlet to Test the Filter

Let's now learn to create a servlet to which the FilterPrePost filter is attached. This servlet is then placed in the src\com\kogent\servlets directory of the FilterApp application. Listing 10.14 shows the code for the ResponseServlet.java file (you can find the ResponseServlet.java file on the CD in the code\JavaEE\Chapter10\FilterApp\src\com\kogent\filter folder:

Listing 10.14: Showing the Code for the ResponseServlet.java File

```
package com.kogent.servlets;

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

public class ResponseServlet extends HttpServlet
{
        @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
  {
            res.setContentType("text/html;charset=UTF-8");
        PrintWriter out = res.getWriter();
        out.println("This is a TEST SERVLET");
        out.close();
        }
}
```

In Listing 10.14, the ResponseServlet class extends the HttpServlet class. The This is a TEST SERVLET message is printed to test the filter. Save the ResponseServlet.java file in the \FilterApp\src\com\kogent\servlets directory and compile it. After compilation, the class file should be located at the \FilterApp\WEB-INF\classes\com\kogent\servlets directory.

Configuring the Filter

In order to configure the filter, you need to provide a mapping in the web.xml file. The following code snippet shows the code for mapping the filter:

```
<filter>
  <filter-name>prePost</filter-name>
  <filter-class>com.kogent.filter.FilterPrePost</filter-class>
</filter>
<filter-mapping>
  <filter-name>prePost</filter-name>
  <servlet-name>ResponseServlet</servlet-name>
</filter-mapping>
```

In the preceding code snippet, the prePost filter is configured to the FilterPrePost class that is defined in the com.kogent.filter package. The prePost filter is mapped to the ResponseServlet class. The following code snippet shows the mapping for the ResponseServlet servlet class:

```
<servlet>
        <servlet-name>ResponseServlet</servlet-name>
        <servlet-class>com.kogent.servlets.ResponseServlet</servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>ResponseServlet</servlet-name>
        <url-pattern>/ResponseServlet</url-pattern>
</servlet>
```

The Web container first invokes the filter named prePost for any requests to the ResponseServlet class.

Save changes to web.xml. Save web.xml at FilterApp\WEB-INF\ directory. Now after configuring the filter, let's test the filter to see the output.

Testing the FilterPrePost Filter

Let's now test the FilterPrePost filter to see the output. You should ensure that the Glassfish V3 application server is running. Now, browse the the `http://localhost:8080/FilterApp/ResponseServlet` URL.

Figure 10.10 displays the output of the `ResponseServlet` class:

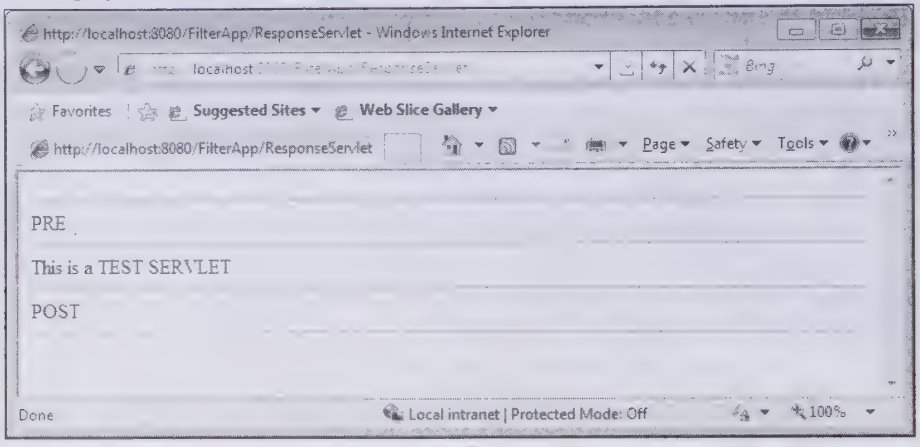


Figure 10.10: Displaying the Output of the ResponseServlet Class

Discussing Issues in Using Threads with Filters

Filters are used in an environment where threads need to collaborate with each other to carry out various functionalities provided by a Web application. All requests are handled by a single instance of the filter class. Therefore, a single instance may be used by more than one request at the same time. Each request will execute methods on the same filter instance in different threads; therefore, developers need to write filters in a threadsafe manner.

Developers, who are not familiar with threads, may make a common mistake of using object variables to store data, which applies only to a single thread. The variables for storing data used in a particular request should be created inside the method, so that each thread will have its own instance. On the other hand, if the variables are created outside the methods, then all threads will share the same instance. The following code snippet shows how to write a threadsafe filter class:

```
public void doFilter(final ServletRequest req, final ServletResponse          res,
  FilterChain chain) throws IOException, ServletException
{
  lHitCount++;
  chain.doFilter(req, res);
  System.out.println("This filter has been hit = " + lHitCount + " times");
}
```

The preceding code snippet shows the doFilter() method of a filter class which is used for counting the number of times the filter class is invoked. The code uses an integer object variable named `lHitCount` for storing the number of hits. The problem arises when two requests for this filter is made to the server at the same time. In this case, both threads may increment the `lHitCount` variable before either thread executes the line that prints the count to the log file. Therefore, both threads print the same value. The increment in `lHitCount` is done when the count value is printed to the log file, as shown by the following code snippet:

```
System.out.println("This filter has been hit = " + (++lHitCount) + " times");
```

However, this may also result in thread issues, as the JVM may not complete the two operations of incrementing and printing at once. The JVM may switch to another thread, which may lead to a thread issue. To avoid threading issues, a thread may be paused for few seconds, while the other thread can perform the operation at the same time. This would not lead to a conflict between the two threads.

In the following code snippet, the sleep() method is used to pause a thread for 5 seconds:

```
public void doFilter(final ServletRequest req, final ServletResponse res,
  FilterChain chain) throws IOException, ServletException
{
  lHitCount++;
  try
  {
        Thread.sleep(5000);
  }
  catch(InterruptedException e)
  {
        System.out.println("MyException:" + e);
  }
  chain.doFilter(req, resp);
        System.out.println("This filter has been hit = " + lHitCount + "
        times");
  }
```

The preceding code snippet prints the following output on the console of application server:

```
This filter has been hit = 2 times
This filter has been hit = 2 times
```

In addition, data can be shared among multiple threads. If one thread modifies the shared instance while the other thread is retrieving data using that instance, then inappropriate data would be retrieved. This would happen only if the instance being accessed is not threadsafe. So, to avoid such circumstances, threadsafe classes, such as `HashTable`, should be used instead of `HashMap` for sharing data.

Summary

The chapter introduced filters, described their relevance, and demonstrated how filters work in the context of a Web application. You have also learned how to configure filters using Deployment Descriptor as well as annotations. Then, the FilterApp application is developed in which a generic filter is created by using the `Filter` interface. In the `FilterApp` Web application, the MyFilter filter is used to verify the user credentials. Next, the `FilterApp` application is rebuild using annotations to configure filters. Further, you have also learned how to use initializing parameters with filters. Toward the end, the chapter also discussed how filters can manipulate the response of a target resource.

The next chapter discusses JavaServer Faces (JSF), an MVC framework, and creates an application using JSF tags.

Quick Revise

Q1. What is a filter?

Ans. A filter is a Java class that is invoked in response to requests for resources, such as Java Servlet or JavaServer Pages.

Q2. List the interfaces defined in the Filter API.

Ans. The Filter API defines the following three interfaces:

- Filter
- FilterConfig
- FilterChain

Q3. List the methods of the Filter interface.

Ans. The Filter interface provides the following three methods:

- init()
- doFilter()
- destroy()

Q4. Which class is used to manipulate responses in filters?

Ans. The HttpServletResponseWrapper class is a custom class that wraps the response object before it is sent to the target resource.

Q5. When should filters be used?

Ans. Filters are used to perform the following tasks:

- Security verification
- Session validation
- Logging operations
- Internationalization
- Triggering resource access events
- Image conversion
- Scaling maps
- Data compression
- Encryption

Q6. Explain the life cycle of a filter.

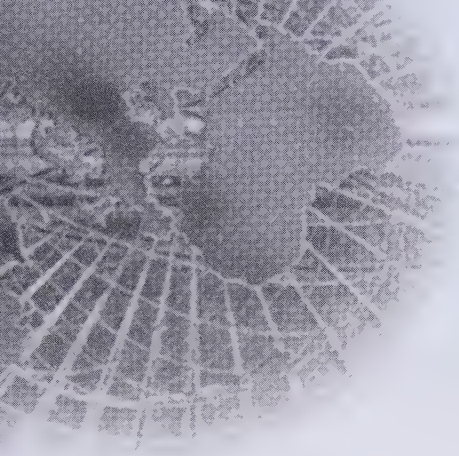
Ans. When a target resource is requested, the Web container invokes the init method of the filter class mapped to that target resource. The init method initializes the corresponding filter and invokes the doFilter() method. The doFilter() method forwards the request to the next filter in the chain. If the next filter in the chain is the last filter, the target resource is invoked. Finally, the destroy method is invoked to remove the filter instance from memory.

Q7. Which interface should be implemented while creating a filter?

Ans. While creating a filter, the filter class must implement the Filter interface to override the methods of this interface.

Q8. Where do we configure the filters?

Ans. We configure the filters in Deployment Descriptor (web.xml). The web.xml file is used to map a filter to a resource or to a URL pattern.



11

Working with JavaServer Faces 2.0

While developing a Web application, the developer designs a User Interface (UI) to interact with the clients, develops business logic to process data, and implements navigation rules to be followed when the clients access the application. Prior to the introduction of JavaServer Faces (JSF), the developer had to manually write the code to define the common tasks, such as validating user inputs and converting user input strings into specific Java objects to build Web applications. JSF is a Web application framework, which allows a developer to handle Web-based tasks easily with the help of its Application Programming InterfaceAPIs and tags and to design rich user interfaces with its components.

JSF can be defined as a framework that makes Web application development easy by supporting different rich, powerful, and ready-to-use UI components. JSF framework is based on Model-View-Controller (MVC) which is one of the most popular design patterns available for implementing separate layers, such as view, model, and controller to provide greater maintainability of an application.

JSF not only provides simple tags to design a UI component, such as label, text box, list box, but also enables you to perform the following tasks:

- Binding of UI components with some model data
- Handling different events on UI components (such as text change in a text box) on server side
- Validating user inputs and conveying all validation error messages to the client
- Defining navigation rules from one view to another according to the output of business process.

Creating a Web application using JSF is a simple and easy task from the perspective of a Web developer. The standard and powerful features of JSF, such as UI components and use of annotation make it a preferred choice among all available Java technologies used for developing a Web application.

JSF is a product of Java Community Process (JCP), which provides suggestions for new Java application programming interfaces and technology enhancements in the form of Java Specification Requests (JSR). The first specification for the initial release of JSF came as JSR 127. This specification was for both JSF 1.0 and JSF 1.1 versions. The latest version of JSF is JSF 2.0 that has been specified in JSR 314.

You need to download different JARs and TLD files to support the development of JSF-based Web applications for the versions prior to Java EE 5. Later, JSF 1.2 became a part of Java Platform, Enterprise Edition 5 (Java EE 5); therefore, while using Java EE 5 as development platform for your Web application, you can use JSF tags and other APIs, such as `java.util`, `javax.faces` for creating UI components, handling events, input validation, and defining page navigation. In the Java EE 6 platform, the latest version of JSF, JSF 2.0 is introduced with some new features, such as bookmarking for URLs, support for the AJAX requests, and use of annotations instead of the faces-config.xml file.

In the chapter you learn about JSF 2.0, its features, architecture, and elements. Then you learn about the JSF request processing life cycle. Next, you learn about JSF tag libraries, UI components, and backing beans. In addition, you learn about the implementation of input validation and type conversion in JSF application. Moreover, how to handle navigation among various JSF pages and implement internationalization in JSF pages is also described in the chapter. Towards the end, a Web application is created by using the JSF framework. In this application, you can add, edit, update, and delete the records of employees of an organization.

Introducing JSF

JSF technology includes a set of APIs, which represent different UI components and helps in managing their states. These APIs further help in handling events on the UI components and validate user inputs through the UI components. JSF framework provides the flexibility of creating simple as well as complex applications as this technology uses the most popular Java server technologies (Servlet and Java Server Page) and does not limit a developer to a specific markup language or client device. The UI component classes bundled with JSF APIs contain logic implementation for various component functionalities and do not have any client-specific presentation logic; therefore, the JSF UI components can be rendered for different client devices. Currently, JSF provides a custom renderer and Java Server Page (JSP) custom tag library for rendering UI components for an HTML client.

JSF is a robust Web application framework that implements an event programming model to handle different events and actions performed by the client on different UI components. To handle each event, a listener should

be registered on server side. While developing a Web application, a developer has to write navigation rules inside source code to navigate from one Web page to another. JSF provides a simple way to define navigation rules in a configuration file, and displays different error messages showing the real cause of errors to clients. These messages are generated while validating user inputs against some validation rule and can be displayed on the same page that contains the UI components.

There are different Web application frameworks that implement one or more forms of MVC design pattern. JSF is based on MVC2 pattern and this pattern is based on component type development. In this pattern, the developers have to concenterate only on their respective component. This introduces separate layers, such as model, view, and controller and helps the developer to concentrate on a single type of component by making a Web application easy to maintain. The different categories of UI components, such as model, view, and controller are created for different functionalities, such as use of TextField, DialogBox, SimpleLabel, and ColorChooser and can be used separately.

NOTE

JSF technology is based on the MVC architecture. It is used for separating presentation logic from business logic; therefore, if you have been practicing MVC, then it is very simple to work with JSF.

Explaning the Features of JSF

Java technology provides various frameworks to develop a Web application. Some of these frameworks, such as Struts, are more popular than JSF, but the rich yet simple features of JSF make it one of the preferred choices for designing and managing UI components in a Web application. The following are the various features of JSF:

- Provides an easy-to-use environment that is Integerated Development Environment (IDE) for developing Web applications with JSF UI components. It has extensive tool support from the companies, such as Sun, IBM, and Oracle.
- Facilitates creating complex UI components in a Web page through its own set of tags, which are provided as JSP custom tag library. Designing a UI component is easy with JSF as it is based on MVC design pattern, which clearly separates presentation and business logic.
- Provides a way to manage all UI components in a Web page. Managing UI components includes validation of user input, state of component, page navigation, and event handling.
- Provides extensible architecture, which means that you can add other functionalities over JSF and can easily customize and reuse JSF UI components.
- Supports multiple client devices. There are different renderers to make same UI component to be rendered or displayed for different client devices. Various component classes can be extended to create custom component tag libraries to support a particular type of client.
- Contains components that support internationalization and enable displaying localized messages according to the specified Locale.
- Supports a standard Rapid Application Development (RAD) Java Web application framework, which enables fast development of a powerful application with a set of reusable components.
- Provides the developer with a way to link visual components to the controller or model components without breaking the layering.
- Provides Expression Language (EL) for a JSF page. As JSF pages use JSP tags, it is difficult to embed separate ELs into one JSF page. One of the key concerns of Java EE specification is to keep its different Web tier technologies, such as JSP, JSF, and JSP Standard Tag Library (JSTL) aligned. This alignment resulted in the creation of a Unified EL, which integrates JSP 2.0 EL and JSF 1.1 EL. JSP 2.1 and JSF 1.2 support this Unified EL. In other words, you can use the JSTL tags with JSF components.
- Helps in building Web 2.0 applications that use Asynchronous JavaScript and XML technology (AJAX) and further reduces the complexities involved in creating UI components.
- Allows the configuration of application wide ResourceBundle to EL using <resource-bundle> element in the faces-config.xml file.

In addition to the features mentioned so far, JSF 2.0, the latest version of JSF, has various new features. The previous version, JSF 1.1, was developed to support JSP 1.2; therefore, could not work with new JSP versions, such as JSP 2.1. The new features added in JSF 2.0 are as follows:

- Provides the ViewDeclarationLanguage API that allows developers to integrate the view declaration languages with JSF runtime environment.
- Introduces Facelets as the non-JSP view declaration language. You learn about Facelets in detail in *Appendix I, Working with Facelets.*
- Introduces the composite component feature that simplifies the development of custom components. In JSF 2.0, you can define the composite User Interface (UI) components in the Extensible Markup Language (XML) file using Hyper Text Markup Language (HTML) contents as well as elements from the http://java.sun.com/jsf/composite namespace.
- Provides an integration of JSF with AJAX. Initially many JSF and AJAX integrated frameworks, such as ICEFaces, Ajax4jsf and many more were available; however, these frameworks lead to compatibility issues. The JSF users need to use the third party frameworks to add AJAX functionality in the JSF application; therefore, JSF 2.0 introduced the jsf.ajax.request() method to return the AJAX request to the view component. In addition to this method, the <f:ajax> tag was also used to send AJAX request back to the view component.
- Provides support for notifications of system-related events. In the previous version of JSF, JSF 1.2, the notifications of FacesEvents were provided to FacesListeners and the PhaseEvents notifications were provided to PhaseListeners. Therefore, the notifications of other events, such as system-related events were not provided. JSF 2.1 provides support for the notification of the SystemEvents to SystemEventListeners.
- Allows you to use annotations to define managed beans. In other words, instead of providing the XML configurations for managed beans, you can use annotations.

Now, let's explore the architecture of JSF.

Exploring the JSF Architecture

Similar to most of the popular Web application frameworks, JSF implements MVC design pattern. The implementation of MVC design pattern helps to design different components separately, and each of these components implements different type of logic. For example, you can create view component for presentation logic and model component for business logic implementation. A controller is another component, which controls the execution of various model methods and navigates from one view to another in a Web application. In JSF, you have a single front controller, which handles all JSF requests mapped to it. Figure 11.1 shows the MVC design pattern:

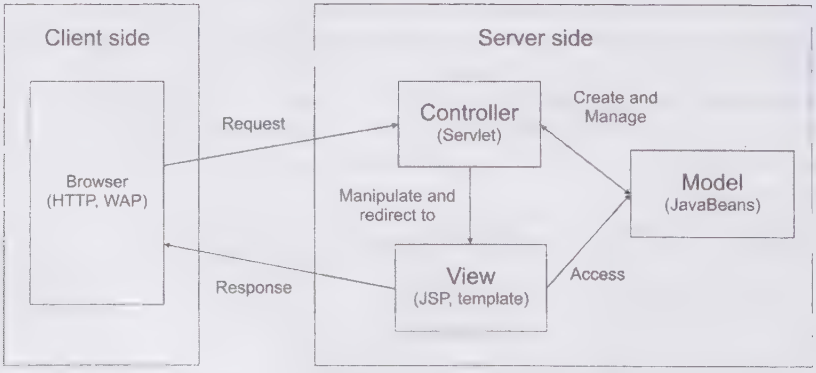


Figure 11.1: Displaying the MVC Design Pattern

The controller part of the JSF architecture consists of a Controller Servlet, that is, `FacesServlet`, a centralized configuration file, faces-config.xml, and a set of event handlers for the Web application. The Front Controller Servlet receives all the requests and manages the request processing life cycle of each request to generate

response for the client. In JSF, the FacesServlet is a Servlet that is responsible to manage the request processing life cycle of the Web application and is also used by JSF to design the user interface. Every JSF application contains the `web.xml` file, which specifies a Java Servlet of type `FacesServlet`. The application settings are also configured in FacesServlet through which all requests are channeled. On the basis of the results of component events, the flow of an application's pages is also controlled by the FacesServlet. This process of handling the flow of pages is represented in the FrontController design pattern. The FacesServlet also manages client requests by referencing the page mappings provided in the faces-config.xml configuration file. This configuration file contains a number of JSF configurations that we discuss in the Configuring JSF Application section later in this chapter. Event Listeners respond to component events, which perform some processing and generate an outcome that can be used by `FacesServlet`.

The Model in JSF architecture (Figure 11.2) is a set of server-side JavaBeans that retrieves the values from the model components, such as the database and defines methods on the basis of these values. These JavaBeans may further persist in a database through an underlying persistence layer. This layer may include Java Data Objects, Enterprise JavaBeans, or an Object-Relational Mapping implementation, such as hibernate. The view in JSF architecture comprises stateful UI components.

The UI Components are rendered in different ways according to the type of the client. The view delegates this task to separate the renderers, each taking care of one specific output type, such as HTML or Wireless Markup Language (WML). You can also attach various additional delegates such as validators and converters to specific components. Converters are used to validate and convert the values entered by the user. The client always submits strings as values for input fields that may need to be converted to a numerical data type, if the string values have to be used in a calculation. Validators check if the values delivered by the client are syntactically correct or are according to a specified format. The view also features resources, which may be used for localization of the Web application. Figure 11.2 shows the MVC architecture implemented in JSF:

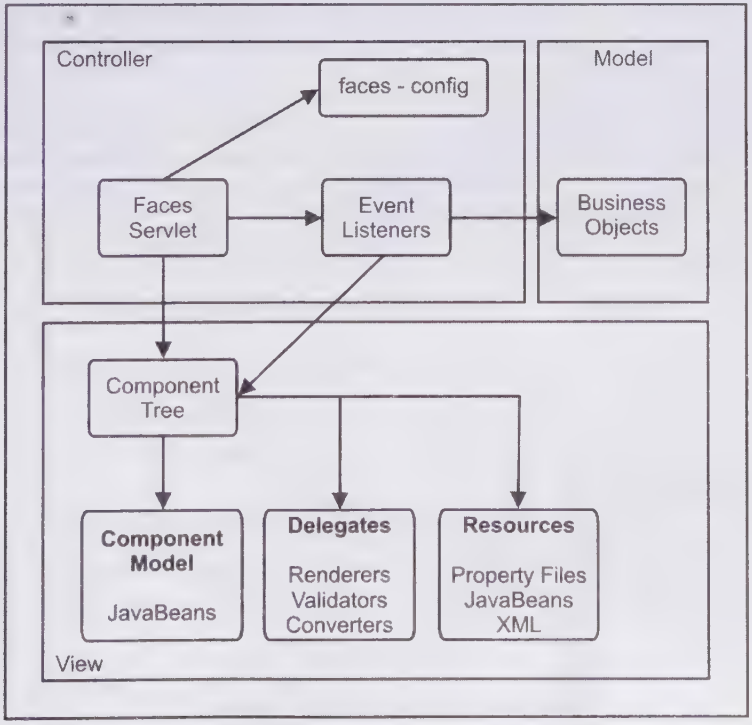


Figure 11.2: Displaying the MVC Architecture of JavaServer Faces

In Figure 11.2, various JSF elements as mapped with MVC architecture. Let's discuss different JSF elements, such as UI components, backing beans, and event listeners in the following section.

Describing JSF Elements

JSF technology has its own set of elements, such as, UI component, renderer, converter, and backing beans which altogether form the JSF framework. These elements provide the core features of JSF technology. In this section, we are talking about various JSF components (elements), such as UI components, validators, and renderers. Let's discuss all the JSF components and explain how they all are related to each other. The following are the elements of JSF:

- UI Component
- Renderer
- Validator
- Backing Beans
- Converter
- Events and Listeners
- Message
- Navigation

UI Component

UI components are the basic reusable components for developing UIs using the JSF framework. The developer can define UI components as stateful objects maintained on server side. The server communicates with the client through these UI components. These components are simple JavaBeans containing properties, methods, and events. The JSF UI components are also called the Web application UI components.

In a Web application, every interaction between a client and a server requires a complete HTTP request-response cycle to generate a message on client side to invalid data input or any other server side error. Each interaction between a client and a server needs to redisplay the request page with an additional error message for previously entered data in input fields. In JSF, no separate logic is required to redisplay the request page with past data values since the JSF UI components can remember their values. The JSF UI components are known as stateful UI components. All UI components are organized as a component tree and are usually displayed as a JSF page. The component tree, also known as view, is the internal presentation of a JSF page. The nested components in the component tree are presented as child components for the component that encloses it. Figure 11.3 shows an internal presentation of a component tree designed for a UI:

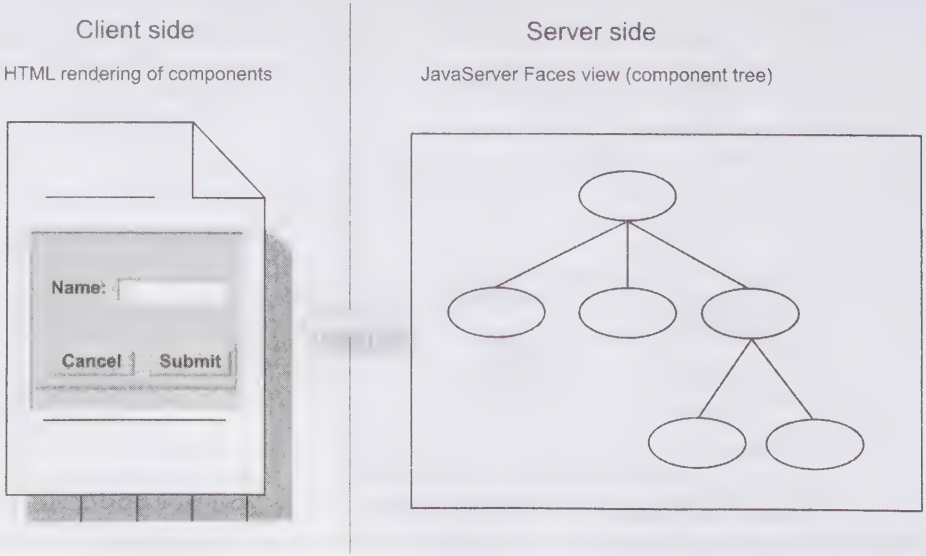


Figure 11.3: Displaying the UI Components that are being Managed as Component Tree

In Figure 11.3, you can see a form with one label, one text field, and a panel with two buttons. The rendering language other than HTML is used to display a form on client side and store the internal representation of the request page. The UI components provide only the functionality and do not involve any presentation logic for displaying the form. The element named renderer is used to ensure how a UI component appears to the user. The separation of UI components from presentation logic provides an easy way to display the responses for different client requests.

Each UI component in a view has its own component identifier. The component identifier can either be provided by the developer or can be generated automatically by JSF implementation at runtime.

To use the JSF UI components, instead of writing any code (HTML, JavaScript, and CSS), you just need to assemble and configure them. JSF provides various standard UI components, such as labels, text boxes, check boxes, radio buttons, panels, and data grids. All the UI components are discussed later in the chapter.

Renderer

Usually, Web applications send a response to the clients Web browser in HTML format. However, the client devices, such as mobile phones or PDAs do not provide the HTML based browser. Some of the formats supported by these handheld devices are WML and Extensive Hyper Text Markup Language (XHTML) basic. Since handheld technology is growing very fast, there arises a need to create a Web application that could respond to other markup languages as well. Earlier, without JSF, we had to incorporate many changes in the Web application to add support for different clients.

JSF introduces renderers, which completely separate the functionality of a component from the presentation logic on the client-side. A Web application does not require many changes in it due to the use of renderers. A JSF UI component is capable of rendering itself, which is known as direct rendering. JSF also supports another type of rendering, called as indirect/delegated rendering, in which a separate class handles the process of rendering. This separate class is known as renderer. A renderer class is responsible for handling the encoding, which is a process of creating the presentation of UI components in some markup language for the given client. The renderers are child components of UI components and used to set and get data from UI components. Let's create a simple HtmlInputText component using the following code snippet:

```
<h:inputText label="Name" id="name" required="true"/>
```

When a component is rendered and the response is created for the HTML client, the following HTML code lines are sent to the client browser, as shown in the following code snippet:

```
<input id="myForm:name" type="text" name="myForm:name" />
```

Renderers are organized into render kits. Each render kit has a specific type of output and all its registered renderers produce output in same format. For instance, JSF provides a standard render kit for HTML 4.0. You can use different render kits to support different types of clients. The following are the two different rendering models supported by JSF:

- **Direct Rendering Model** – Refers to the model in which the rendering logic is directly encapsulated into the UI components; therefore, there is no clear separation of functionality and presentation. This technique can be used if you are sure that the created UI component is used for a particular client. This results in a compact solution as the component is implemented in a single class. However, the reusability of component is poor.
- **Indirect/Delegated Rendering Model** – Refers to the model that uses a separate renderer for the component. In this model, the renderer, represented by `render` class works with the `encode()` and `decode()` methods. Rather, it replaces these methods by the render kit to get the output to be displayed in a format compatible with the client.

Validators

UI components enable clients to interact with server by giving some inputs. The data entered by the client must be validated for its correctness according to the defined range and format. The invalid data may cause application to produce wrong results and may leave the system with inconsistent data. We can use JavaScript or Java for writing validation logic, either on client side or on server side according to its complexity. While

working with JSF components, the Java code is not required to validate the input data; instead, JSF can handle the validation of data in the following three different ways:

- Implementing validation inside the UI component
- Using validator method in backing beans
- Using validator classes

The first way of handling validation is at UI component level where the component itself handles simple validation logic, which is specific to it only; therefore, cannot be used by other components. We can also implement validation logic in validator method of the backing bean that helps in validating one or more fields of a form, but again other components cannot use it. The third way to handle validation is to use Validator classes, also known as validators. The validators provide validation for generic cases, such as validating a field for allowed length of string or for an allowed number range.

We can attach validators with the component that needs to be validated for its data; however, more than one validator can be attached with one component. When some validation error is encountered, validators simply add error message in the existing message list. You can see in the following code snippet how a `Length` validator is associated with the `HtmlInputText` component:

```
<h:inputText label="Name" id="name" required="true">
        <f:validateLength minimum="2" maximum="25"/>
</h:inputText>
```

The preceding code snippet shows a text field whose id is name and validates the length of that field.

Backing Beans

JSF architecture follows MVC design pattern and consequently focuses on separating business logic and data from presentation logic. The model part in backing beans contain business logic and data, while the view consists of presentation logic with UI components.

In a JSF application, there are some JavaBean objects, which handle or store data between the business model and UI component at intermediate stage. These JavaBean objects are known as Backing Beans. Backing beans also play the role of a Controller by handling the interaction between the view and the Model. In other words, in JSF, the objects that handle the interaction between view and Model are called backing beans.

Generally, a single backing bean is created for a single view, but you can create a number of backing beans for a single view. Backing bean data model is usually a copy of the application model. All changes in the component model are often propagated back to the application model through the backing beans model.

Backing beans are normal JavaBeans that contain properties and event listener methods for storing and manipulating the user's data. The event listener methods can also manipulate the UI or execute the application logic in the backend. We can associate a UI component with a property of backing bean through EL value binding. We can also bind the method of backing beans with some UI action component (such as button) to process the logic. For example, we can enable the execution of backing bean method when a button is clicked to perform some action. The following code snippet shows the code to bind a UI component with a property of a given backing bean:

```
<h:inputText label="Name" id="name" value="#{student.name}" required="true"/>
```

We can also bind a backing bean instance directly with a component instance. This binding is useful when you want to manipulate a component using a Java code. We can bind a backing bean property with the component instance directly by using the binding property of a component's tag, as shown in the following code snippet:

```
<h:inputText label="Name" id="name" value="#{student.name}"
binding="#{student.inputName}" required="true"/>
```

The model objects are not directly bound to UI components and are manipulated by backing beans. In addition, you do not need to write codes for creation or instantiation of backing beans, because in JSF a declarative mechanism is used for the creation of backing beans. That is you can configure a backing bean in `faces-config.xml` file and define the name of the object, the class to be , and the scope of that object. The beans that are configured also known as `Managed Beans`. Therefore, all backing beans are managed beans. The following code snippet shows the code to configure a backing bean:

```
<managed-bean>
  <managed-bean-name>student</managed-bean-name>
  <managed-bean-class>com.kogent.Student</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

The preceding code snippet shows the configuration of a backing bean in the faces-config.xml file. Let's now discuss the converters in the next subsection.

Converters

In general, Web applications interact with the client through HTTP request and response. The data from client to server travels as HTTP request parameter, which can be a string only. We can enter only a string through a UI component, whereas in the backing bean or in the model layer the data can be any Java object. This is where we need a `Converter`. A converter is a translator, which can convert a Java object into a string and an input string into some Java object. JSF provides a number of standard converters for common types, such as date and numbers. We can also develop our own custom converters, which further support extendibility from third party.

We can associate a converter with any UI component through simple markup style, as shown in the following code snippet:

```
<h:inputText id="dob" value="#{student.dob}">
  <f:convertDateTime type="date" pattern="dd-MM-yyyy"/>
</h:inputText>
```

The renderers use converters during the process of encoding and decoding. JSF converters also support formatting and localization. For example, we have a DateTime converter to display Date object in different string formats (short and long) according to the client Locale.

Events and Listeners

JSF implements an event driven processing, where events generated on different UI components can be mapped to the execution of a method. A user can interact with the Web application by generating different events on different UI components. For example, a user can generate a click event on a command button or the value change event on some input text box. The JSF event model is based on simple JavaBean's event model. It means that JSF uses JavaBeans to handle events with event objects and event listeners. A UI component can generate one or more types of events, and we can register respective event listener with the component to respond to those events. The event listeners further invoke associated backing bean method to process the appropriate logic. Now, the developers do not need to work on stateless HTTP request/response model directly.

While writing JSF applications, we just need to assign listeners to the UI components for the generated events on these UI components. JSF supports an interesting way to create a simple method in your backing bean and assigns it as event listener to the UI component, as shown in the following code snippet:

```
<h:inputText id="empid" valueChangeListener="#{employee.valueChanged}"/>
```

In the preceding code snippet, the valueChanged() method of employee backing bean has been registered as listener for Value-change event of an HTMLInputText component.

JSF defines four different types of standard events, which are as follows:

- Action events
- Value-change events
- Data model events
- Phase events

In addition to these standard events defined in the JSF framework, this framework's UI components can be customized to support a number of other types of events. Let's discuss the standard JSF events.

Action Events

The UI component, which represents a command, can generate action events. A component that can generate an action event is also known as an action source. For example, HtmlCommandButton is an action source and can trigger an action event. For handling action events, you can associate an action listener with the UI component.

There are two types of action listeners, one that affects navigation, and the other does not. The action listener, which does not affect navigation executes the code, modifies backing bean or application model, and redisplays the current page. On the other hand, the action listener, which affects navigation, executes the logic and returns the output to navigation model to select the next page to be displayed to the client.

The action listener, which affects navigation, checks the output of action method configured with the component responsible for firing the action event. An action method is a backing bean method, which is invoked when an action event is fired. The string returned from the action method is dynamic in nature; therefore, can have different values. The following code snippet shows the code to configure an action method with the HtmlCommandButton component:

```
<h:commandButton type="submit" value="Add Employee" action="#{employee.addNew}"/>
```

The action listener shown in the preceding code snippet is addNew(). The method to be used as action method must return a string according to the logic performed. The `addNew()` method of backing bean, `employee` is shown in the following code snippet:

```
public class Employee {
  ...
  public String addNew() {
        if(...){
              return "success";
        }
        else{
              return "error";
        }
  }
}
```

In some cases we can directly put some string as action attribute instead of retrieving a string from an action method. The action listener returns the hard coded (or static) outcome string. The following code snippet shows how you can hard code a string instead of invoking a backing bean method:

```
<h:commandButton type="submit" value="Cancel" action="cancel"/>
```

In the preceding code snippet, the submit type command button is created and string type value is provided for the action attribute.

NOTE

The way an output string maps to another view, has been discussed later in this chapter.

If you do not want to affect navigation by an action event and just need to process some logic, you need to configure an action listener method instead of action method with the component. The action listener method again is a backing bean method, which returns void and takes an ActionEvent object as an argument. The action listener method has the access to action resource component. The following code snippet shows the code to configure an action listener method:

```
<h:commandButton type="submit" value="Update"
actionListener="#{employee.update}"/>
```

The backing bean method, update() must have the structure shown in the following code snippet:

```
public class Employee {
  ...
  public void update(ActionEvent e) {

  }
}
```

Value-change Events

We have different input UI components, such as the text box and text area that are used to take input from the client. Whenever, an end user enters a new value into these components, a value-change event is fired. We have a valueChangeListener attribute to store the reference of the method that handles the value-change event. In the same way as configuring backing bean methods as action event listener, we can set backing bean method as a

value-change listener. The following code snippet shows how to set a backing bean's method as `valueChangeListener`:

```
<h:inputText id="rollno" name="rollno"
 valueChangeListener="#{student.processValueChange}"/>
```

In the preceding code snippet, the processValueChange() method of the `student` backing bean takes an object of javax.faces.event.ValueChangeEvent class as argument. The structure of the processValueChange() method is shown in the following code snippet:

```
public class Student {
    ......
    public void processValueChange(ValueChangeEvent e){
          HtmlInputText comp=(HtmlInputText)e.getComponent();
          ... ...
    }
}
```

When a form is submitted with some value changes in the component, the value-change event is fired only after the associated validators validate the new value entered in the input component. Similar to an action listener, we can also register some event listener classes that implement an interface instead of configuring value-change event method.

Data Model Events

The data model events are associated with data-aware UI components. A data-aware component is a component that gives a list of items to be selected. The example of data-aware component is `HtmlDataTable`. This component cycles through rows in a data source and exposes individual rows to child components. The data model event is fired when data-aware components process a row. Data model event is different from other two events discussed earlier as we cannot register DataModelListener with the component itself in JSP, rather it is registered in the Java code.

The data model event is not fired by the data-aware component itself, rather it is fired by an object of javax.faces.model.DataModel. DataModel is an abstraction for different data binding technologies that are used to adapt a variety of data sources, such as arrays, lists, and ResultSets. The object of DataModel allows the registration of DataModelListener to handle DataModelEvent.

Phase Events

Every request in a JSF application has a life cycle, which is a set of six different stages or phases. These phases include getting request-based view, obtaining component values from request parameters, validate user input, update backing bean and model objects, invoking action listener and returning response back to the client. We have discussed all these life-cycle phases later in this chapter.

Each phase has its start point and end-point and a single event is fired when a phase either begins or ends. Both events, the event at start point and the event at end point, are known as phase events. Phase events are not fired by any UI component; rather, they are generated by JSF. Similar to implementing the data model events, we need to implement a Java interface to register an event listener. However, phase events are used internally and you can use them for implementing some specific logic in your application. The listener for a phase event is registered directly with the instance of `javax.faces.lifecycle.LifeCycle`, which represents the entire life cycle of request processing.

Message

Messages are an integral part of the JSF framework. Generally, a message is used to inform the user about different errors that can occur while processing the logic. A message may be associated with some component or it can be an application level message; therefore, we have two categories of messages, application errors (business logic) and user input errors (empty text field, invalid input text). Different validators can add messages to `FacesContext` and also provide different validations. The converters can convert the messages into application or user input errors. In addition to error messages, we can also have informational messages, such as the messages conveying information to the user about successful addition of a new record in the database.

HtmlMessage and HtmlMessages are two components used to display messages. HtmlMessage is used to display a single message associated with a specific component, as shown in the following code snippet:

```
<h:message for="name" style="color:red"/>
```

All messages associated with FacesContext can be displayed at a single location using HtmlMessages component, as shown in the following code snippet:

```
<h:messages style="color:red"/>
```

JSF supports Internationalization; therefore, all messages can be localized to support client Locale. Messages provide an easy way to communicate different errors and other processing details to the user. The standard validators and converter automatically generates an error message if it finds any incorrect data. We can also create messages in our validator method, action method, and action listener methods, as shown in the following code snippet:

```
FacesContext fc=FacesContext.getCurrentInstance();
fc.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_WARN,
"Your Error Message", ""));
```

You can create an object of FacesMessage with its severity level, message summary, and message detail. The created message needs to be added in FacesContext and can be displayed on any page using <h:messages/> tag.

Navigation

A Web application is a collection of Web pages linked together in a particular sequence to solve a particular purpose, such as shopping. The clients need to go through these pages to perform the desired task and the Web developer has to implement this navigation logic. Earlier, the navigation logic was implemented in Servlet or JSP code to dispatch the client's request to an appropriate view. However, if the Web application is small (consisting two or three pages) it is a simple task to dispatch the client's request to an appropriate view ; but it becomes complex to keep track of all the paths when the size of an application (number of Web pages) increases to enterprise level.

JSF provides an efficient yet simple way to configure all navigation rules declaratively in a single location, that is, in faces-config.xml file. You can define a set of navigation rules to navigate from one view to another view, which are used by navigation handler. It is the responsibility of navigation handler to load the JSF page according to the outcome (return value) of action method. For every page, you can define a navigation rule with a number of navigation cases for different outcomes. The example of a navigation rule for a page with two navigation cases is shown in the following code snippet:

```
<navigation-rule>
  <from-view-id>/login.jsp</from-view-id>
  <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/welcome.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
        <from-outcome>failure</from-outcome>
        <to-view-id>/login.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

In the preceding code snippet, a navigation rule is defined for login.jsp page. The two cases are provided for two different outcomes, `success` and `failure`. This approach of defining navigation rules in a centralized configuration file helps you to maintain all logical paths in an application easily.

You have learned about JSF elements, their functionalities, and their integration. All these elements, such as all UI components, management of backing beans, implementation of standard validators and converters, and configuration of navigation rules in a JSF Web application are discussed later in this chapter.

Exploring the JSF Request Processing Life Cycle

The JSF framework processes a request in predefined steps similar to other Web application frameworks. You can also develop a Web application in JSF, without knowing the request processing details. This reduces the

efforts in the enhancements of the existing JSF application. While discussing about the phases of the JSF request process life cycle, the request processing details are also explored.

The request-processing life cycle starts as soon as a request is submitted to the JSF controller Servlet and ends when the client receives a response for the request. Following are the six different phases of JSF request processing life cycle and are given in the order they are executed by the JSF framework:

- The Restore View Phase
- The Apply Request Values Phase
- The Process Validations Phase
- The Update Model Values Phase
- The Invoke Application Phase
- The Render Response Phase

The execution of all these phases is not required for a given request. The request processing life cycle can be terminated in any phase by executing `FacesContext.responseComplete()` method, which skips the other phases of the life cycle. For example, when any invalid input is found in the Process Validation Phase, the current page/view is to be redisplayed and some of the phases, such as Invoke Application Phase, may not execute.

These phases altogether make sure that the processing of a request follows a logical path. For example, no application logic is executed before the form is validated for the input data. Further, implementation of these life cycle phases helps a developer in concentrating on business logic instead of form validation and data conversion. The basic idea behind implementing the life cycle is to make sure that before executing any business logic invoked as a consequence of some action, we have a fully populated backing bean with all validated data.

If you want to create custom JSF UI components, you need to concentrate on first and last phases of JSF request processing life cycle. Figure 11.4 represents the JSF request processing life cycle:

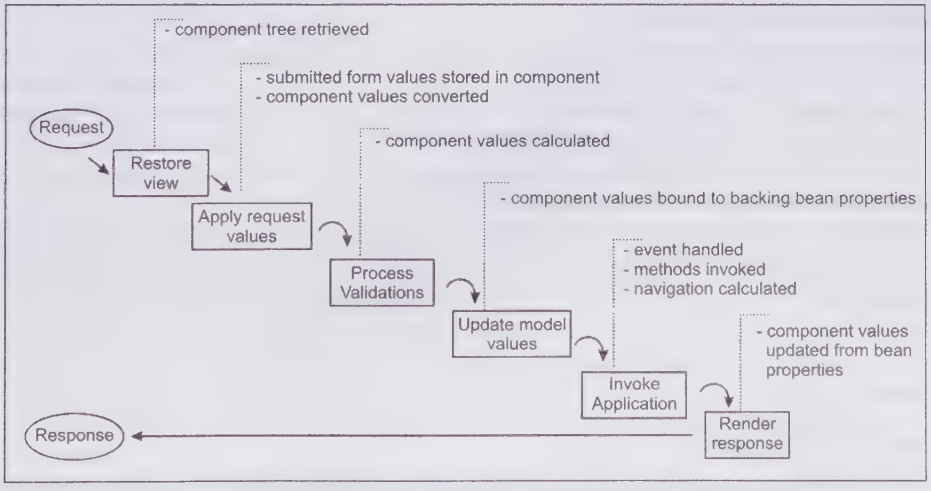


Figure 11.4: Displaying the JSF Request Processing Life Cycle

In a JSF-based application, whenever a request is received by the controller Servlet, an instance of the `javax.faces.context.FacesContext` class is created. The `FacesContext` object contains the state information related to the current JSF request. The `FacesContext` object is accessed and modified in almost all of the request life cycle phases. Let's discuss all six phases of the JSF request processing life cycle with the description of all the functions performed by the framework.

The Restore View Phase

A view can be defined as a collection of components in a request page. All the components in a request page or view are grouped together as a tree; therefore, every view is a tree of components having a unique identification,

that is view ID. The JSF controller Servlet finds the view ID from the request, which can be determined by the requested JSP page name. The controller can load the existing view or can create a new view if it does not exist.

The Restore View phase can handle three different views, new view, initial view, and postback. In case of a new view, this phase creates the view of the request page and associates event handlers and validators to the components in the view. This phase also stores the view as root component using setViewRoot() method on `FacesContext` instance. When the JSF page is requested for the first time, the view needs to be loaded for the first time (initial view) and the controller creates an empty view. In case of initial view, the framework directly moves to render the response page.

If the user requests a page accessed earlier (postback), the controller needs to restore the view with the current state of the view. In case of postback, the next phase after Restore View phase is the Apply Request Values phase.

The Apply Request Values Phase

There are a set of JSF UI components that receive the input as request data from the user and then processes it. In the Apply Request Values phase, the JSF implementation iterates the component objects in the component tree and requests every component to decode itself. Decoding is the process in which the framework sets the submitted values for the components based on the request parameters. The component value can be validated in this phase if the immediate property is true for the component. In case of validation error, an error message is added and the render response phase starts. If the immediate property of the component is set to false, then the submitted string values are converted into the desired data type. At the end of this phase, each component stores recent values submitted through current request. After the completion of this phase, the existing events are broadcasted for the associated event listeners.

The action events can fire in this phase but are handled in the invoke application phase. In this phase, they are created and added to the FacesContext instance.

The Process Validations Phase

In this phase, the values of all the components are validated against some validation rules. We can associate a given component with some JSF standard validator or can define our own validators. The validators are registered with the components of the JSF page. In case, the value of a given component is found invalid, a validation error message is added in the FacesContext and the render response phase starts to display the current view with the validation message. The following code snippet shows an HtmlInputText component created with a validator to validate its value for its length:

```
<h:inputText label="Name" id="name" value="#{student.name}" required="true"
  valueChangeListener="#{student.change}">
  <f:validateLength minimum="2" maximum="25"/>
</h:inputText>
```

All the value change events associated with different components are fired and used by the appropriate listener in this phase. If all the submitted values are found valid, the framework enters into the update model values phase of the life cycle.

The Update Model Values Phase

The Process Validation phase ensures that all component values are valid. After validating component values, all values of the backing beans or model objects are updated with the component values to save the state of components so that these values can be used in subsequent requests. Every component can be associated with some backing bean property and the backing bean is updated with the current value of the component. The following code snippet shows the code to associate the `name` property of `student` backing bean with an `HtmlInputText` component:

```
<h:inputText label="Name" id="name" value="#{student.name}" required="true">
</h:inputText>
```

In the preceding code snippet, backing bean properties are associated with a component through a value binding expression, such as `#{sudent.name}`. When these expressions are evaluated, the `FacesServlet` searches for the appropriate backing beans in request, application, or session scope. By the end of this phase, we have all

component values set with all validated user inputs and updated backing beans. Now, we are ready to execute our business logic in the invoke application phase.

The Invoke Application Phase

In the previous phases, we have not executed any business logic. We have just updated component model and backing bean model with the current and valid data. In the Apply Request Values phase, we have added all action events in the FacesContext for their handling in invoke application phase. In invoke application phase, the JSF framework broadcasts all the added action events, which are further handled by the registered action event listeners. We can register some backing bean methods as action listeners or action methods. The action method may contain the logic for different operations, such as rendering responses and adding messages. Action methods are also integrated with the navigation model and helps in determining the next view or page to be displayed.

After the completion of all phases and handling of all events by the respective listeners, the framework is ready with the next view for the user.

The Render Response Phase

The Render Response phase is the last phase in the JSF request-processing life cycle. This phase is responsible for sending responses to the client. The state of the view (component tree) is saved in this phase before the response is rendered for the client. The state of the view can be stored on the server side using client session or can be saved on the client side using hidden fields. Saving the state of a view is important, so that it can be restored in the Restore view phase in case of postback. In case of postback, all error messages that are added in Apply Request Values phase, Process Validations phase, or Update Model Values phase are also displayed along with the current state of the components in the view.

JSF framework is not bound to a particular display technology to render the response to the client. We can have different approaches to render the responses. We can use the output of encoding method with the markup code, which is either stored in some template resource or generated by another application for the client. The JSF implementations use a different approach, that is, it uses the output of decoding methods with some dynamic resources such as JSP page.

During Render Response phase, all component values are converted into a string to be displayed. When this phase completes, the Web container sents the response to the client as a response page in browser.

We have discussed all six phases of the JSF request processing life cycle that explain how a JSF request (read it Faces request) is changed into JSF response (read it Faces response). A brief explanation of Faces and Non-Faces request and responses are as follows:

- **Faces Response**—Refers to the Servlet response constructed after the execution of Render Response phase of request processing life cycle.
- **Non-Faces Response**—Refers to the Servlet response, which is not constructed by the execution of Render Response phase, for example a JSP page that does not have JSF components.
- **Faces Request**—Refers to the Servlet request sent from previously generated faces response, for example, a form submitted from a JSF UI component, where request URI recognizes JSF component tree for processing the request.
- **Non-Faces Request**—Refers to the Servlet request sent to an application component, such as Servlet of JSP page, rather than directed to the JSF component tree.

These four request and response types lead to four combinations, which define the request and response scenarios that may occur in a Web application based on JSF. In case of Non-Faces request or Non-Faces response, the JSF request processing life cycle is not followed.

Exploring JSF Tag Libraries

We can design pages in a JSF application by using different JSF UI components. The various UI components can be created easily for the page by using tags defined in the JSF tag library. In addition to laying out different

components in the page, designing a page in JSF application includes making provision in the page to use backing beans, validators, converters, and other backend objects associated with the components.

JSF 1.2 provides two different tag libraries, JSF HTML tag library and JSF core tag library. These tag libraries provide tags to create pages in JSF application with a rich set of UI components supporting event handling and input validation. You must include the following standard tag libraries to use JSF UI components in a JSP page:

- **JSF HTML tag library** – Defines the tags that represent general HTML UI components. These HTML components include `HtmlInputText`, `HtmlInputTextarea`, `HtmlForm`, and `HtmlCommandButton`.
- **JSF core tag library** – Defines the tags that perform core actions. This tag library does not depend upon specific render kit.

We can use these tag libraries simply by providing a taglib directive in our JSP page with proper uri and prefix, as shown in the following code snippet:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
```

The code lines in the preceding code snippet must be the first two lines in a JSP page in a JSF based Web application.

Let's discuss all JSF HTML tags and JSF core tags. All these tags have been fully described with their syntax and behavior.

JSF HTML Tags

The JSF HTML tag library provides JSF component tags for all UI components and HTML RenderKit. It means that all the tags discussed for JSF HTML tag library are rendered to some HTML response for the client browser. Each tag in JSF HTML tag library corresponds to a specific class of UI components present in the `javax.faces.component.html` package.

All HTML UI components created using the tags defined in the JSF HTML tag library render to different HTML tags. For example <h:form> tags render to the HTML <form> element. All JSF HTML tags support common HTML tag attributes, such as alt, style, tabindex, and width. In addition, Dyanamic Hyper Text Markup Language (DHTML) event attributes, such as onclick, ondbclick, onmouseover are also supported by JSF HTML tags. All JSF HTML tags have some common attributes, which are listed in Table 11.1:

Table 11.1: Representing the Basic HTML Tag Attributes

Attribute	Description	Component Type
id	Takes a string, which is a component identifier and must be unique within the closest parent component.	UI components
binding	Binds a component with a backing bean property.	
rendered	Takes a boolean value. By default this value is true, which indicates rendering of component. If this value is set to false, it suppresses rendering of component, which means the component is not rendered at all.	
styleClass	Refers to the class name of the Cascading StyleSheet (CSS).	
value	Sets the value of a component. Typically used for value binding.	Input, Output, and Command type of components
valueChangeListener	Binds a method, which can respond to value change events.	Input component
converter	Sets converter class name.	Input and Output components
validator	Sets Validator class or Validator method for a component.	Input component

Table 11.1: Representing the Basic HTML Tag Attributes		
Attribute	**Description**	**Component Type**
required	Takes a boolean value. If the value is true, then it requires a value to be entered in the associated field, otherwise it is not necessary to enter the value in associated field.	Input component

Now, let's discuss various JSF HTML tags and their behavior with examples.

The <h:commandButton> Tag

The `<h:commandButton>` tag is used to create a command button in the JSP page. This represents an HtmlCommandButton component. This tag is rendered to HTML <input> element with a type attribute, `submit` or `reset`; therefore, this tag can be used to create a submit button which when clicked would send the form data to the server and would fire an action event. The most commonly used attributes of the <h:commandButton> tag are shown in Table 11.2:

Table 11.2: Showing the CommonAttributes of <h:commandButton > Tag	
Attribute	**Description**
action	Accepts a string or a method binding expression as its value. The navigation handler determines the JSF page to be loaded next according to the string or the outcome of the action method.
actionListener	Accepts a method binding expression as its value. You should note that the signature of this method must be void methodName(ActionEvent object).
image	Accepts the context relative path of an image as its value, which would be displayed on the button.
immediate	Accepts a boolean value. The default value of this attribute is false, in which the action and action listener methods are invoked at the end of request life cycle.
type	Accepts the type of the rendered input element. This can be set as a button, submit or reset. The default type is submit.
value	Accepts a string as its value, which would be displayed on the button. We can provide string or value expression for the value attribute.

Let's consider the following code snippet that uses the <h:commandButton> tag:

```
<h:commandButton value="Login" action="#{user.login}"/>
<h:commandButton value="Check Availability" actionListener="#{user.check}" />
```

In the preceding code snippet, the first `<h:commandButton>` tag displays the `Login` submit button and binds a `login` method of the `user` backing bean with the button. The method of backing bean (`user`) assigned to the `action` attribute in method binding expression returns a string that determines navigation flow. The second `<h:commandButton>` tag displays the Check Availability (a submit) button and registers an action listener method (`check`) from backing bean (`user`) that performs an operation named Check Availability.

The <h:commandLink> Tag

Normally, we use a button to submit the data of a form or to invoke an action. However, sometimes we may want to perform the same thing using HTML hyperlink. The `<h:commandLink>` tag renders an HTML hyperlink, which is capable of submitting form data and firing some action. It also supports sending of parameters to an action method or action listener method using UIParameter component. Table 11.3 shows the most commonly used attributes of the `<h:commandLink>` tag:

Table 11.3: Showing the CommonAttributes of the <h:commandLink> Tag	
Attribute	**Description**
action	Accepts a string ór a method binding expression as its value. The navigation handler determines the JSF page to be loaded next according to the string or the outcome of the action method. The signature of the action method should be String methodName().
actionListener	Accepts a method binding expression as its value. You should note that the signature of the method must be void methodName(ActionEvent object).
charset	Accepts the character encoding of the linked reference as its value.
immediate	Accepts a boolean value. If the value is set to false, the action or action listener methods are invoked in the end of request life cycle. The default value of the `immediate` attribute is false.
type	Accepts the content type of the linked resource as its value. For example, text/html and image/gif.
value	Accepts a string or an expression referencing a value. For example, the label displayed by the link.

Let's consider the following code snippet that uses the <h:commandLink> tag:

```
<h:commandLink action="#{student.delete}" value="Delete">
  <f:setPropertyActionListener target="#{student.roll}" value="#{user.roll}"/>
</h:commandLink>
```

In the preceding code snippet, the `roll` property of the backing bean, `student` is set with the value of roll property of user bean before the action invoke phase, that is, before the `delete()` method of student backing bean is executed. You learn about the <f:setPropertyActionListener> tag in detail later in the *JSF Core Tags* section.

The **<h:dataTable>** Tag

This tag is used to create an HtmlDataTable component and helps to create a data grid. The HtmlDataTable component automatically populates with data sets. The data grid is a necessary component for any UI framework; therefore, it has been developed as a standard data grid. The `<h:dataTable>` tag renders to an HTML `<table>` element but its rows are automatically created according to the data set stored in the table. The columns of this table can be defined using the UIColumn components. Table 11.4 shows the most commonly used attributes of the <h:dataTable> tag:

Table 11.4: Showing the Common Attributes of <h:dataTable> Tag	
Attribute	**Description**
bgcolor	Specifies the background color for the table.
border	Accepts an integer value as the width of table border.
cellpadding	Accepts an integer value that specifies how much padding exist around table cells.
cellspacing	Specifies spacing between table cells.
columnClasses	Accepts comma-separated list of CSS classes for columns of tables.
first	Accepts the index of the first row shown in the table.
footerClass	Specifies the CSS class for the table footer.
frame	Specifies the values of the sides of the frame surrounding the table. The allowed values are none, above, below, hsides, vsides, lhs, rhs, box, and border.
headerClass	Specifies the CSS class for the table header.
rowClasses	Accepts comma-separated list of CSS classes for rows.

Table 11.4: Showing the Common Attributes of <h:dataTable> Tag	
Attribute	**Description**
rules	Specifies the lines to be drawn between cells. The allowed values are groups, rows, columns, and all.
summary	Sets the summary of the table depicting the purpose as well as structure of the table. The value of the `summary` attribute is mainly used for non-visual feedback such as speech.
var	Accepts a string value, which is created by the data table and used to represent the current item. This value is associated with the `var` attribute.

The following code snippet shows the use of the <h:dataTable> tag:

```
<h:dataTable value="#{applicationScope.students}" var="student" cellspacing="1">
<h:column>

  <f:facet name="header">
        <h:outputText value="Roll No."/>
  </f:facet>
  <h:outputText value="#{student.roll}"/>
  </h:column>

        <h:column>
        <f:facet name="header">
              <h:outputText value="Student Name"/>
        </f:facet>

        <h:outputText value="#{student.name}"/>
  </h:column>
</h:dataTable>
```

In the preceding code snippet, the column component tags represent the UIData and UIColumn components, respectively. We have created columns for standard data grid using the <h:column> element, which represents the UIColumn component. The HtmlDataTable component supports data binding to a set of data objects represented by the DataModel instance. The DataModel instance holds the data model of a component and can be instantiated, saved in a separate context from the component definition, and can be used by different components simultaneously. As the UIData component iterates over the rows of the data represented by the DataModel instance, it also processes the UIColumn component for each row.

The <h:form> Tag

The `<h:form>` tag is used to create an HTML form element on the JSF page. The `<h:form>` contains other required input fields, such as name, age, address, and zip . You should place all the input UI components within the <h:form> tag, because the form is concerned with user input. The HTML form created by this tag behaves as a normal form element, which can be submitted to the server with data to be processed.

We can have more than one `<h:form>` tags in a single page. Only one form, which contains the click command button, is submitted to the server.

While using the `<h:form>` tag, you do not need to provide method and action attributes as they are required in an HTML form element. The default method used to provide method and action attributes in an HTML form element is post and action would be the result obtained from the `getActionURL()` method of ViewHandler for the application. Table 11.5 shows the commonly used attributes of the <h:form> tag:

Table 11.5: Showing the Common Attributes of the <h:form> Tag	
Atrribute	**Description**
id	Accepts a string that is a component identifier and must be unique within the closest parent component

Table 11.5: Showing the Common Attributes of the <h:form> Tag	
Atrribute	**Description**
binding	Accepts a value expression, which binds it with some backing bean property
rendered	Accepts a boolean value indicating that whether the component will be rendered or not
styleClass	Sets the style class to be applied when the component is rendered

Let's consider the following code snippet that uses the <h:form> tag:

```
<h:form id="loginForm">
  //Other UI components
</h:form>
```

The <h:form> tag in the preceding code snippet is used to encapsulate UI components into a form named loginForm.

The <h:graphicImage > Tag

The `<h:graphicImage>` tag is used to display images on a Web page. In JSF pages, the HtmlGraphicImage component is created to handle the images embedded in the designed page. The `<h:graphicImage>` tag creates a simple HTML `<img>` element. The src property of `<img>` tag is obtained from the uri property of the `<h:graphicImage>` tag. Table 11.6 shows the most commonly used attributes of the `<h:graphicImage>` tag:

Table 11.6: Showing the Common Attributes of the <h:graphicImage > Tag	
Attribute	**Description**
id	Accepts a string, which is component identifier and must be unique within the closest parent component.
binding	Accepts a value expression, which binds this attribute with a backing bean property.
rendered	Accepts a boolean value indicating whether or not the component will be rendered.
styleClass	Sets the style class to be applied when the component is rendered.
value	Accepts the image file name with full path as its value. This value is rendered as the value for the `src` attribute of the `<img>` element.
alt	Sets some textual description of the image.
height	Sets the height of the image.
style	Specifies the CSS style to be applied.
url	Accepts the context relative path for the resource image. It works as an alias for the `value` attribute.
width	Sets the width of the image.

Let's consider the following code snippet that uses the `<h:graphicImage>` tag:

```
<h:graphicImage value="images/animal.jpg" width="300"/>
```

The `<h:graphicImage>` tag in the preceding code snippet displays a graphic image (animal.jpg) located in images directory.

The <h:inputHidden> Tag

The `<h:inputHidden>` tag is used to create an input element, which is not displayed to the user of the page. Using this tag a hidden field can be added to the page, which need not be displayed on the page, but required to process logic. The `<h:inputHidden>` tag corresponds to the `HtmlInputHidden` component and renders to a simple `<input>` element with its type set to hidden. Table 11.7 shows the most commonly used attributes of the `<h:inputHidden>` tag:

Table 11.7: Showing the Common Attributes of the <h:inputHidden> Tag	
Attribute	**Description**
immediate	Accepts a boolean value. If the value of this attribute is set to true the value of the component is validated immediately in the apply request phase instead of waiting for validation phase.
required	Accepts a boolean value. If the value of this attribute is set to true the user is required to enter some value for this input component.
valueChangeListener	Accepts a method expression to define a listener to handle the value change event for this component

Let's consider the following code snippet that uses the `<h:inputHidden>` tag:

```
<h:inputHidden id="roll" value="#{student.roll}"/>
```

The preceding code snippet shows the use of the <h:inputHidden> tag. When you use this tag inside the <h:form> tag, the value of the roll parameter is sent to the server as a hidden parameter.

The <h:inputSecret> Tag

The `<h:inputSecret>` tag creates an HtmlInputSecret component and renders an HTML `<input>` element with type set as password. The text entered into this secret field cannot be read, as the characters are displayed using asterisk or other character. Table 11.8 shows the most commonly used attributes of the `<h:inputSecret>` tag:

Table 11.8: Showing the Common Attributes of the <h:inputSecret> Tag	
Attribute	**Description**
immediate	Accepts a boolean value. If the value of this attribute is set to true the value of the component is validated immediately in the apply request phase instead of waiting for validation phase.
redisplay	Accepts a boolean value. When the value of the `redisplay` attribute is set to true, the value of input field is redisplayed on reloading the Web page.
required	Accepts a boolean value and setting this value to true ensures that the user is required to enter some value for this input component.
valueChangeListener	Accepts a method expression to define a listener to handle the value change event for this component.

The following code snippet shows the use of the <h:inputSecret> tag:

```
<h:inputSecret redisplay="false" value="#{user.password}"/>
```

The preceding code snippet converts the password entered by a user into encoded form such as ****.

The <h:inputText> Tag

The `<h:inputText>` tag is used to create a single line text box, which is the most common component to take input from the user. The `<h:inputText>` tag corresponds to the `HtmlInputText` class and renders to simple `<input>` element with type, text. As other input UI components, this tag can also be attached with some validator or converter. Table 11.9 shows the most commonly used attributes of the <h:inputText> tag:

Table 11.9: Showing the Common Attributes of the <h:inputText> Tag	
Attribute	**Description**
immediate	Accepts a boolean value. If this value is set to true the value of a component is validated immediately in the apply request phase instead of waiting for validation phase.
required	Accepts a boolean value. If this value is set to true the user is required to enter some value for this input component.
valueChangeListener	Accepts a method expression to define a listener to handle the value change event for this component.

Let's consider the following code snippet that uses the <h:inputText> tag:

```
<h:inputText id="uname" label="User Name" value="#{user.fullName}" required="true"/>
```

In the preceding code snippet, the `HtmlInputText` component has been bound with the `fullName` property of the `user` backing bean with a value expression. Setting all the required attributes to `true` ensures that the user must enter some string in the text field.

The <h:inputTextarea> Tag

The `<h:inputTextarea>` tag is used to create a multiline text input control. This tag corresponds to `Html rich text box` and renders to HTML `<textarea>` element. The `<h:inputTextarea>` tag has the properties to set column (that is width in characters) as well as row (that is height in characters) of text area. Table 11.10 shows the most commonly used attributes of the `<h:inputTextarea>` tag:

Table 11.10: Showing the Common Attributes of the <h:inputTextarea> Tag

Attribute	Description
cols	Sets the number of characters in the text area.
immediate	Accepts the boolean value. This value is set to true to process validation early in the request processing life cycle.
required	Accepts a boolean value. If this value is set to true the user is required to enter some value for this input component.
rows	Sets the number of rows.
valueChangeListener	Accepts a method expression to define a listener to handle the value change event for this component.

Let's consider the following code snippet that uses the `<h:inputTextarea>` tag:

```
<h:inputTextarea id="address" cols="20" rows="3" value="#{student.address}"/>
```

The `HtmlInputTextarea` component in the preceding code snippet creates a multiline text box, which has been bound with the `address` property of the student backing bean.

The <h:message> Tag

The JSF provides a simple way of displaying messages, which have been added to FacesContext by some validator, converter, or some other method for a specific UI component. A message can be displayed using the `<h:message>` tag. The associated UI component can be referred by setting the `for` attribute with the id of that particular component. Table 11.11 shows the most commonly used attributes of the `<h:message>` tag:

Table 11.11: Showing the Common Attributes of the <h:message> Tag

Attribute	Description
for	Specifies the ID of the component whose message is displayed as applicable only to h:message.
errorClass	Applies the specified CSS class to error messages.
errorStyle	Applies the provided CSS style to error messages.
fatalClass	Applies the specified CSS class to fatal messages.
fatalStyle	Applies the provided CSS style to fatal messages.
infoClass	Applies the specified CSS class to information based messages.
infoStyle	Applies the provided CSS style to information based messages.
showDetail	Accepts a boolean value specifing whether or not the message details should be displayed.
showSummary	Accepts a boolean value specifying whether or not the summary part of the message should be displayed. By default, the value of this attribute is false for the <h:message> tag.

Table 11.11: Showing the Common Attributes of the <h:message> Tag	
Attribute	**Description**
tooltip	Accepts a boolean value specifying whether or not the summary message should be displayed in tooltip.
warnClass	Applies the specified CSS class to warning message.
warnStyle	Applies the provided CSS style to warning message.

Let's consider the following code snippet that uses the `<h:message>` tag:

```
<h:message for="uname"></h:message>
or
<h:message for="uname" errorClass="error"/>
```

The first `<h:message>` tag displays an error message for the component named, `uname`. The second tag additionally applies the CSS style class (error) to display same error message in a particular format.

The <h:messages> Tag

There may be a number of messages that can be added during the request processing process. In addition, there can be different application level messages, which are not associated with any particular component. The `<h:messages>` tag is used to display all the messages added into FacesContext at once.

Table 11.12 shows the most commonly used attributes of the <h:messages> tag:

Table 11.12: Showing the Common Attributes of the <h:messages> Tag	
Attribute	**Description**
errorClass	Applies the specified CSS class to the error messages.
errorStyle	Applies the provided CSS style to the error messages.
fatalClass	Applies the specified CSS class to fatal messages.
fatalStyle	Applies the provided CSS style to fatal messages.
globalOnly	Accepts a boolean value specifying whether or not to display only global messages. By default, its value is false.
infoClass	Applies the specified CSS class to information based messages.
infoStyle	Applies the provided CSS style to information based message.
layout	Specifies the layout for a message, which can either be a list or a table. If you do not use this attribute then the layout is displayed in the form of a list.
showDetail	Accepts a boolean value specifying whether the message details should be provided or not. By default this value is false.
showSummary	Accepts a boolean value specifying whether the message summaries should be displayed or not. By default this value is true.
tooltip	Accepts a boolean value specifying whether the message details are rendered in a tooltip or not; if the showDetail and showSummary attributes are true then the tooltip is rendered.
warnClass	Applies the CSS class for warning message
warnStyle	Applies the CSS style for warning message

Let's consider the following code snippet that uses the <h:messages> tag:

```
<h:messages errorClass="error" warnClass="warn"/>
```

The <h:messages> tag in the preceding code snippet displays all error and warning messages in their respective formats.

The <h:outputFormat> Tag

The `<h:outputFormat>` tag works similar to the `<h:outputText>` tag but it also formats the compound messages. The `<h:outputFormat>` tag allows a page author to display the concatenated messages as a

message format pattern. This tag is used to display parameterized text. The value attribute of the <h:outputFormat> tag specifies a message format pattern and the substation parameters are specified with the <f:param> tag. The associated parameters should be defined in the same order as they appear in the message format pattern. Table 11.13 shows the most commonly used attribute of the <h:outputFormat> tag:

Table 11.13: Showing the Common Attribute of the <h:outputFormat> Tag	
Important Attribute	
escape	Accepts a boolean value. If set to true, escapes <, >, and & characters. The default value of the escape attribute is false.

Let's consider following code snippet to understand the use of the <h:outputFormat> tag.

```
<h:outputFormat id="msg" value="Hello {0}, Your Roll number is {1}.">
  <f:param value="#{student.name}"/>

  <f:param value="#{student.rollno}"/>
</h:outputFormat>
```

The <h:outputFormat> tag in the preceding code snippet displays a command message, which automatically contains the name and roll number of the student.

The <h:outputLabel> Tag

The <h:outputLabel> tag is used to create the HtmlOutputLabel component on a JSF page. The HtmlOutputLabel component is used as a label for a given component. It displays a label for a component, which is associated with the component through the given id. Table 11.14 shows the most commonly used attribute of the <h:outputLabel> tag:

Table 11.14:Showing the Common Attribute of the <h:outputLabel> Tag	
Attribute	**Description**
for	Specifies the ID of the component to be labeled.

Let's consider the following code snippet that uses the <h:outputLabel> tag:

```
<h:outputLabel for="msg">Your Message</h:outputLabel>
<h:inputText id="msg" value="#{user.message}"/>
```

The <h:outputLabel> tag in the preceding code snippet displays the label, Your Message before the text field component named msg.

The <h:outputLink> Tag

The <h:outputLink> tag is used to create an HTML anchor on a Web page that helps you to navigate through the current Web page to other locations. Unlike the HtmlCommandLink component, the HtmlOutputLink component does not generate any action event, as the URL is already specified using the value attribute. The <h:outputLink> tag renders to the HTML <a> element, and the href attribute of this anchor element is set with the value set for value attribute of the <h:outputLink> tag. Table 11.15 shows the most commonly used attributes of the <h:outputLink> tag:

Table 11.15: Showing the Common Attributes of the <h:outputLink> Tag	
Attribute	**Description**
Id	Accepts a string, which is component identifier and must be unique within the closest parent component.
binding	Accepts a value expression, which binds this attribute with some backing bean property.
Layout	Accepts the type of layout markup used while rendering the component group. If this attribute is set to block, then the HTML element is produced; otherwise, is produced.

Table 11.15: Showing the Common Attributes of the <h:outputLink> Tag	
Attribute	**Description**
rendered	Accepts boolean value indicating whether or not the component will be rendered.
style	Applies the specified CSS style when the component is rendered.
styleClass	Applies the CSS style class when the component is rendered.
value	Accepts the linked resource name with its path. The value set for the value attribute is set as the value for the href attribute of the rendered HTML anchor tag.

Let's consider the following code snippet that uses the <h:outputLink> tag:

```
<h:outputLink value="login.faces">Login</h:outputLink>
or
<h:outputLink value="login.faces">
  <h:outputText value="Login"/>
</h:outputLink>
```

The `<h:outputLink>` tag in the preceding code snippet displays a hyperlink (named Login) on a Web page. When a user clicks this hyperlink, request goes to the login.faces page.

The <h:outputText> Tag

The `<h:outputText>` tag is used to display the single line output string. The string to be displayed can be defined giving JSF expression for value attributes. The `<h:outputText>` tag corresponds to `HtmlOutputText` component. Table 11.16 shows the most commonly used attributes of the `<h:outputText>` tag:

Table 11.16: Showing the Common Attribute of the <h:outputText> Tag	
Attribute	**Description**
escape	Defines whether or not special characters, such as <,>, and & is to be escaped. For escaping these characters the value should be true but by default the value is false.

Let's consider following code snippet that uses the <h:outputText> tag:

```
<h:outputText value="#{user.name}"/>
or
<h:outputText value="Welcome"/>
```

The first <h:outputText> tag in the preceding code snippet reads the value of the name property of (user) backing bean and displays it on browser. The second tag displays the Welcome string (specified in its value attribute) on browser.

The <h:panelGrid> Tag

The `<h:panelGrid>` tag corresponds to the `HtmlPanelGrid` component and renders to the HTML `<table>` element. An entire table can be presented using the `<h:panelGrid>` tag. You can also define header and footer using the `<f:facet>` tag, which renders the `<thead>` and `<tfoot>` elements. The number of columns can be set by using the columns property of the `<h:panelGrid>` tag. Table 11.17 shows the most commonly used attributes of the `<h:panelGrid>` tag:

Table 11.17: Showing the Common Attributes of the <h:panelGrid> Tag	
Attribute	**Description**
bgcolor	Specifies the background color of a table
border	Specifies the width of table border
cellpadding	Specifies the padding space to be provided around the table cells
cellspacing	Specifies the space to be provided between table cells

Table 11.17: Showing the Common Attributes of the <h:panelGrid> Tag	
Attribute	**Description**
columnClasses	Specifies the lists of CSS classes (separated by comma) for columns
columns	Specifies the total number of columns in a table
footerClass	Specify the CSS class for the footer of the table
frame	Specifies the frame surrounding the table that needs to be drawn; valid values for this purpose are: none, above, below, hsides, vsides, lhs, rhs, box, and border
headerClass	Specifies the CSS class for the header of the table
rowClasses	Specifies the lists of CSS classes (separated by comma) for rows
rules	Specifies the lines drawn between cells; valid values are: groups, rows, columns, and all
summary	Provides the purpose and structure of the table, that is basically used for non-visual feedback such as speech

The following code snippet shows the use of the <h:panelGrid> tag:

```
<h:panelGrid id="panel" columns="2" border="1">
  <h:outputLabel for="uid">User ID</h:outputLabel>
  <h:inputText id="uid" value="#{user.userName}"/>
  <h:outputLabel for="pwd">Password</h:outputLabel>
  <h:inputText id="pwd" value="#{user.password}"/>
</h:panelGrid>
```

In the preceding code snippet, a table with two columns and two rows is created by using the <h:panelGrid> tag. The number of rows created depend on the number of other components enclosed in the `<h:panelGrid>` tag.

The <h:panelGroup> Tag

The `<h:panelGroup>` tag can encapsulate a set of other components to make a single entity. This tag creates HtmlPanelGroup components that do not render any output, but are used as a place holder. The `<h:panelGroup>` tag is used to create a blank cell inside a table rendered by the `<h:panelGrid>` element. Table 11.18 shows the most commonly used attributes of the <h:panelGroup> tag:

Table 11.18: Showing the Common Attributes of the <h:panelGroup> Tag	
Attribute	**Description**
id	Accepts a string, which is a component identifier and must be unique within the closest parent component
binding	Accepts a value expression, which binds this attribute with some backing bean property
layout	Allows you to set the type of layout markup to be used when rendering the component group. If this attribute is set to block, the HTML element is produced; otherwise, is produced
rendered	Accepts a boolean value indicating whether the component should be rendered or not
style	Allows you to set the CSS style to component when the component is rendered
styleClass	Allows you to set the CSS style class to component when the component is rendered

The following code snippet shows the use of the `<h:panelGroup>` tag:

```
<h:panelGroup id="panel1">
  <h:commandButton value="Login" action="#{user.login}"/>
```

```
    <h:commandButton value="New User" action="#{user.newUser}"/>
</h:panelGroup>
```

The <h:panelGroup> tag in the preceding code snippet displays the Login and New User buttons as one group.

The <h:selectbooleanCheckbox> Tag

The `<h:selectbooleanCheckbox>` tag is used to create a `HtmlSelectbooleanCheckbox` component, which represents single boolean value. The `<h:selectbooleanCheckbox>` tag renders to an HTML `<input>` element with type checkbox. Table 11.19 shows the most commonly used attributes of the <h:selectbooleanCheckbox> tag:

Table 11.19: showing the Common Attributes of the <h:selectbooleanCheckbox> Tag

Attribute	Description
binding	Accepts a value expression, which binds the value (entered by user) with some backing bean property.
converter	Allows you to set the converter instance registered with the component.
Id	Accepts a string, which is a component identifier and must be unique within the closest parent component.
immediate	Allows you to set the process validation early in the life cycle of JSF, by default it is true.
styleClass	Allows you to set the CSS style class to be applied when the component is rendered.
required	Allows you to set a boolean value to a field and setting it to true ensures that the user is required to enter some value for this input component.
rendered	Accepts a boolean value indicating whether the component should be rendered or not.
validator	Accepts a method expression and represents the validator method to be invoked to validate the current value of the component.
value	Allows you to set the current value of a component. A value expression can be used to bind the current value with a backing bean property.
valueChangeListener	Allows you to set the method expression to define a listener to handle the value change event for this component

The following code snippet shows the use of the <h:selectbooleanCheckbox> tag:

```
<h:selectbooleanCheckbox value="#{student.status}"/>
```

The preceding code snippet, code is written to display a checkbox, which takes the value of status property of student backing bean.

The <h:selectManyCheckbox> Tag

As the name suggests, the `<h:selectManyCheckbox>` tag renders a group of checkboxes to be selected. This tag corresponds to `HtmlSelectManyCheckbox` component that renders to the HTML `<table>` element and `<input>` elements of type checkbox. The items to be selected can be encapsulated within the `<h:selectManyCheckbox>` tag by using the `<f:selectItem>` or `<f:selectItems>` tag (discussed later in this chapter). Table 11.20 shows the most commonly used attributes of the <h:selectManyCheckbox> tag:

Table 11.20: Showing the Common Attributes of the <h:selectManyCheckbox> Tag

Attribute	Description
binding	Accepts a value expression, which binds the attribute with some backing bean property.
converter	Allows you to set the converter instance registered with the component.

Table 11.20: Showing the Common Attributes of the <h:selectManyCheckbox> Tag	
Attribute	**Description**
disabledClass	Accepts a CSS style class to be applied on disabled options.
enabledClass	Accepts a CSS style class to be applied on enabled options.
id	Accepts a string, which is a component identifier and must be unique within the closest parent component.
immediate	Allows you to set the value of process validation early in the life cycle of JSF. By default this value is true.
layout	Allows you to set the orientation of components for the Web page that needs to be displayed. The allowed values are pageDirection and lineDirection. The default value is lineDirection and its options are rendered horizontally.
styleClass	Allows you to set the CSS style class to be applied when the component is rendered.
required	Allows you to set the boolean value and setting it to true ensures that the user is required to enter some value for this input component.
rendered	Accepts a boolean value indicating whether the component will be rendered or not.
validator	Accepts a method expression and represents the validator method to be invoked to validate the current value of the component.
value	Allows you to set the current value of the component. A value expression can be used to bind the current value with a backing bean property.
valueChangeListener	Allows you to set the method expression to define a listener to handle the value change event for this component.

The following code snippet shows the use of the <h:selectManyCheckbox> tag:

```
<h:selectManyCheckbox id="country" value="#{user.country}">
  <f:selectItem itemLabel="India" itemValue="ind"/>
  <f:selectItem itemLabel="Pakistan" itemValue="pak"/>
  <f:selectItem itemLabel="Sri Lanka" itemValue="srl"/>
</h:selectManyCheckbox>
```

The `<h:selectManyCheckbox>` tag displays a group of checkboxes with id (equals to country). This tag represents each item of a group using the <f:selectItem> tag.

The <h:selectManyListbox> Tag

The `<h:selectManyListbox>` tag corresponds to `HtmlSelectManyListbox` component, which renders to the HTML `<select>` element. All the child components of the `<f:selectItem>` tag are rendered to the HTML `<option>` element. The `<h:selectManyListbox>` tag provides a list box of choices, which allows the selection of multiple options from the list. The size attribute of this tag defines the total number of options that can be displayed at a time. Table 11.21 shows the most commonly used attributes of the `<h:selectManyListbox>` tag:

Table 11.21:Showing the Common Attributes of <h:selectManyListbox> Tag	
Attribute	**Description**
binding	Accepts a value expression, which binds this attribute with some backing bean property.
converter	Allows you to set the converter instance registered with a component
id	Accepts a string, which is a component identifier and must be unique within the closest parent component.

Table 11.21:Showing the Common Attributes of <h:selectManyListbox> Tag	
Attribute	**Description**
immediate	Allows you to set the process validation early in the life cycle of the JSF.
styleClass	Allows you to set the CSS style class to be applied when the component is rendered.
required	Specifies whether a value is required for input component or not, and setting it to true ensures that the user is required to enter some value for this input component.
rendered	Accepts a boolean value indicating whether the component should be rendered or not.
validator	Accepts a method expression and represents the validator method to be invoked to validate the current value of the component.
value	Allows you to set the current value of the component. A value expression can be used to bind the current value with some backing bean property
valueChangeListener	Allows you to set a method expression to define a listener to handle value change event for this component

The following code snippet shows the use of <h:selectManyListbox> tag:

```
<h:selectManyListbox id="country" value="#{user.country}" size="2">
  <f:selectItem itemLabel="India" itemValue="ind"/>
  <f:selectItem itemLabel="Pakistan" itemValue="pak"/>
  <f:selectItem itemLabel="Sri Lanka" itemValue="srl"/>
</h:selectManyListbox>
```

The preceding code snippet uses the <h:selectManyListbox> tag to display a listbox. The `<f:selectItem>` tag is used to specify different options to be selected in this listbox. These options are rendered as the `<option>` element.

The <h:selectManyMenu> Tag

The `<h:selectManyMenu>` tag creates a multiselect menu, which corresponds to the `HtmlSelectManyMenu` component. The `HtmlSelectManyMenu` component is similar to the `HtmlSelectManyListbox` component, except that the number of options displayed using this component is always one. As the `<h:selectManyListbox>` tag, the `<h:selectManyMenu>` tag also renders to the HTML `<select>` element with all child elements rendered to its `<option>` elements. Table 11.22 shows the most commonly used attributes of the `<h:selectManyMenu>` tag:

Table 11.22: Showing the Common Attributes of the <h:selectManyMenu> Tag	
Attribute	**Description**
binding	Accepts a value expression, which binds this attribute with some backing bean property.
converter	Allows you to set the converter instance registered with the component.
id	Accepts a string, which is a component identifier and must be unique within the closest parent component.
immediate	Allows you to set a boolean value to process validation early in the life cycle of the JSF.
styleClass	Allows you to set the CSS style class to be applied when the component is rendered.
required	Specifies whether a value is required for input component or not, and setting it to true ensures that the user is required to enter some value for the input component.

Table 11.22: Showing the Common Attributes of the <h:selectManyMenu> Tag	
Attribute	**Description**
rendered	Accepts a boolean value indicating whether the component should be rendered or not.
validator	Accepts a method expression and represents the validator method to be invoked to validate the current value of the component.
value	Allows you to set the current value of a component. A value expression can be used to bind the current value with some backing bean property.
valueChangeListener	Allows you to set a method expression to define a listener to handle the value change event for the menu component.

The following code snippet shows the use of the `<h:selectManyMenu>` tag:

```
<h:selectManyMenu id="country" value="#{user.country}">
  <f:selectItem itemLabel="India" itemValue="ind"/>
  <f:selectItem itemLabel="Pakistan" itemValue="pak"/>
  <f:selectItem itemLabel="Shri Lanka" itemValue="srl"/>
</h:selectManyMenu>
```

In the preceding code snippet, the <h:selectManyMenu> tag is created, whose id is country and contains three items, India, Pakistan, and Srilanka. This tag can be accessed from request processing page by using its id, country. Let's now discuss the <h:selectOneListbox> tag in the next subsection.

The <h:selectOneListbox> Tag

The `<h:selectOneListbox>` tag corresponds to the `HtmlSelectOneListbox` component and is rendered similar to `HtmlSelectManyListbox` tag. The only difference between these tags is that the `<h:selectOneListbox>` tag does not allow the selection of more than one option from a list box. You can set the size of a list box to set the number of options to be displayed at a time. Table 11.23 shows the most commonly used attributes of the `<h:selectOneListbox>` tag:

Table 11.23: Showing the Common Attributes of the <h:selectOneListbox> Tag	
Attribute	**Description**
binding	Accepts a value expression, which binds this attribute with some backing bean property.
converter	Allows you to set the converter instance registered with the component.
id	Accepts a string, which is a component identifier and its value should be unique within the ancestor component.
immediate	Allows you to set a boolean value that specifies the phase during which the value change event of JSF life cycle would occur.
styleClass	Allows you to set the CSS style class to be applied when a component is rendered.
required	Allows you to set a boolean value and if it is set to true the user is required to enter some value for this input component.
rendered	Accepts a boolean value indicating whether or not the component will be rendered.
validator	Accepts method expression and represents the validator method to be invoked to validate the current value of the component.
value	Allows you to set the current value of a component. A value expression can be used to bind the current value with some backing bean property.

Table 11.23: Showing the Common Attributes of the <h:selectOneListbox> Tag	
Attribute	**Description**
valueChangeListener	Allows you to set a method expression to define a listener to handle the value change event for this component that is listbox.

The following code snippet shows the use of the `<h:selectOneListbox>` tag:

```
<h:selectOneListbox id="country" value="#{user.country}" size="3">
  <f:selectItem itemLabel="France" itemValue="frc"/>
  <f:selectItem itemLabel="America" itemValue="usa"/>
  <f:selectItem itemLabel="Germany" itemValue="grm"/>
  <f:selectItem itemLabel="England" itemValue="eng"/>
</h:selectOneListbox>
```

The preceding code snippet uses the <h:selectOneListbox> tag to display a listbox on browser. In this listbox, you can select only one option out of various options given in it.

The <h:selectOneMenu> Tag

The `<h:selectOneMenu>` tag represents the `HtmlSelectOneMenu` component , which provides a list of options and out of the list of options only one option is allowed to be selected. This tag displays a drop-down list box on browser. Table 11.24 shows the most commonly used attributes of the `<h:selectOneMenu>` tag:

Table 11.24: Showing the Common Attributes of the <h:selectOneMenu> Tag	
Attribute	**Description**
binding	Accepts a value expression, which binds this attribute with some backing bean property.
converter	Allows you to set the converter instance registered with a component.
id	Accepts a string, which is a component identifier and its value should be unique within the ancestor component.
immediate	Allows you to set a boolean value that specifies in which phase of the JSF life-cycle the value change event should occur.
styleClass	Allows you to set the CSS style class to be applied when the component is rendered.
required	Specifies whether a value is required for input component or not, and if the value is set to true the user is required to enter some value for this input component.
rendered	Accepts a boolean value indicating whether the component will be rendered or not.
validator	Accepts a method expression and represents the validator method to be invoked to validate the current value of the component.
value	Allows you to set the current value of the component. A value expression can be used to bind the current value with some backing bean property.
valueChangeListener	Accepts an expression for method binding. This expression represents a method of value change listener. This method is invoked when a new value is set for the menu component.

The following code snippet shows the use of the `<h:selectOneMenu>` tag:

```
<h:selectOneMenu id="country" value="#{user.country}">
  <f:selectItem itemLabel="France" itemValue="frc"/>
  <f:selectItem itemLabel="America" itemValue="usa"/>
  <f:selectItem itemLabel="Germany" itemValue="grm"/>
</h:selectOneMenu>
```

The preceding code snippet uses the `<h:selectOneMenu>` tag to display the drop-down listbox. You can select only one option from this drop-down list box.

The <h:selectOneRadio> Tag

The `<h:selectOneRadio>` tag is rendered similar to the `<h:selectManyCheckbox>` tag. In JSF an HTML `<table>` element is rendered that contains input elements of radio type. The only difference is that from the HTML <table> table element when you select an option then previously selected option will be deselected automatically. The `<h:selectOneRadio>` tag represents the `HtmlSelectOneRadio` component. Table 11.25 shows the most commonly used attributes of the `<h:selectOneRadio>` tag:

Table 11.25: Showing the Common Attributes of the <h:selectOneRadio> Tag

Attribute	Description
binding	Accepts a value expression, which binds this attribute with some backing bean property.
converter	Allows you to set the converter instance registered with a component.
disabledClass	Specifies a CSS style class that should be applied on disabled option.
enabledClass	Specifies a CSS style class that should be applied on enabled option.
id	Accepts a string, which is a component identifier and the value should be unique within the ancestor component.
immediate	Allows you to set a boolean value that specifies the phase of the JSF life-cycle, the value change event should occur.
layout	Allows you to set the orientation of the list of options on the JSF page. The allowed values are pageDirection and lineDirection. The default value is lineDirection and all options are rendered horizontally.
styleClass	Allows you to set the CSS style class to be applied when the component is rendered.
required	Allows you to set a boolean value and if this value is set to true, the user is required to enter some value for this input component.
rendered	Accepts a boolean value that specifies whether the component is rendered or not.
validator	Accepts a method expression and represents the validator method to be invoked to validate the current value of the component.
value	Allows you to set the current value of the component. A value expression can be used to bind the current value with some backing bean property.
valueChangeListener	Allows you to set a method expression to define a listener to handle the value change event for the radio button component.

The following code snippet shows the use of `<h:selectOneRadio>` tag:

```
<h:selectOneRadio id="sex" value="#{student.sex}">
   <f:selectItem itemValue="Male" itemLabel="Male" />
   <f:selectItem itemValue="Female" itemLabel="Female"/>
</h:selectOneRadio>
```

The preceding code snippet uses the `<h:selectOneRadio>` tag to display radio buttons labeled as Male and Female respectively.

The <h:column> Tag

The `<h:column>` tag is used with the `<h:dataTable>` tag to define the columns provided in a data grid. You can set CSS classes for any header and footer generated for the table using the headerClass and footerClass attributes. Table 11.26 shows the most commonly used attributes of the `<h:column>` tag:

Table 11.26: Showing the Common Attributes of the <h:column> Tag	
Attribute	**Description**
binding	Accepts a value expression, which binds this attribute with some backing bean property.
id	Accepts a string, which is a component identifier and the value should be unique within the ancestor component.
rendered	Accepts a boolean value that specifies whether the component is rendered or not.
footerClass	Allows you to set the CSS style class for the footer generated for a column in a table.
headerClass	Allows you to set the CSS style class for the header generated for a column in a table.

The following code snippet shows the use of the <h:column> tag:

```
<h:dataTable value="#{school.students}" var="stud">
  <h:column>
        <f:facet name="header">
              <h:outputText value="Roll No."/>
        </f:facet>
        <h:outputText value="#{stud.roll}"/>
  </h:column>
  <h:column>
        <f:facet name="header">
              <h:outputText value="Student Name"/>
        </f:facet>
        <h:outputText value="#{stud.name}"/>
  </h:column>
</h:dataTable>
```

The preceding code snippet uses the `<h:column>` tag inside the <h:dataTable> tag to define two columns named Roll no. and Student Name, respectively. The `<h:column>` tag uses the `<h:facet>` tag, (discussed later in this chapter).

JSF Core Tags

The JSF core Tag library contains tags for different JSF custom actions and these tags do not depend on any particular render kit. The JSF core tags can be categorized according to their usage in an application. For example, core tags support event handling to implement standard JSF validators and converters, and create items and items in a list. Table 11.27 shows various tag types and their respective tags that the JSF core tag library contains:

Table 11.27: Showing the JSF Core Tag Types and Tags under These Types	
Tag types	**Tags**
Event handling tags	f:actionListener
	f:valueChangeListener
Container for form tags	f:view
Component attributes Configuration tag	f:attribute
Data conversion tags	f:converter
	f:convertDateTime
	f:convertNumber
Facet tag	f:facet
Localization tag	f:loadBundle
Parameter substitution tag	f:param

Table 11.27: Showing the JSF Core Tag Types and Tags under These Types	
Tag types	**Tags**
Tags for representing items in a list	f:selectItem
	f:selectItems
Container tag	f:subview
Validator tags	f:validateDoubleRange
	f:validateLength
	f:validateLongRange
	f:validator
Output tag	f:verbatim

The <f:actionListener> Tag

The `<f:actionListener>` tag adds an action listener to a component. It registers an `ActionListener` instance with the closest parent component. However, you can use the actionListener attribute of the component tag, but this tag registers only a single action listener method with the component. The `<f:actionListener>` tag is used to register more than one action listener with a single action event generated. All action listener classes used in JSF must implement the `javax.faces.event.ActionListener` interface. Table 11.28 describes the attributes of `<f:actionListener>` tag:

Table 11.28: Describing the Attributes of the <f:actionListener> Tag	
Attribute	**Description**
type	Accepts a fully qualified name of the class that has implemented the ActionListener interface and defines the processAction() method.
binding	Accepts a value as a binding expression, which evaluates an object. This object should implement javax.faces.event.ActionListener interface.

The following code snippet shows the use of the <f:actionListener> tag:

```
<h:commandButton value="Add Student">
        <f:actionListener type="com.kogent.Listener1" />
        <f:actionListener type="com.kogent.Listener2" />
    </h:commandButton>
```

The preceding code snippet uses the <f:actionListener> tag inside the <h:commandButton> tag to register Listener1 and Listener2 classes with the Add Student button.

The <f:valueChangeListener> Tag

The `<f:valueChangeListener>` tag is used to register a value change listener for the closest parent component. Using this tag, you can register more than one value change listeners with a single component. All listener classes registered using `<f:valueChangeListener>` tag must implement javax.faces.event.ValueChangeListener interface. Table 11.29 lists the most commonly used attributes of the `<f:valueChangeListener>` tag:

Table 11.29: Showing the Common Attributes of the <f:valueChangeListener> Tag	
Attribute	**Description**
type	Takes a fully qualified name of the class that has implemented the ValueChangeListener interface and defines the processValueChange() method.
binding	Accepts a value as a binding expression, which evaluates an object. This object should implement `javax.faces.event.ValueChangeListener` interface.

The following code snippet shows the use of the <f:valueChangeListener> tag:

```
<h:inputText id="rollno" immediate="true" onchange="submit()">
  <f:valueChangeListener type="com.kogent.SomeListener"/>
</h:inputText>
```

The preceding code snippet shows the use of the `<f:valueChangeListener>` tag to register the `SomeListener` class with the value change event that will occur in rollno text field.

The <f:view> Tag

The `<f:view>` tag encloses all other JSF components on a Web page. As you have learned earlier, all the components of a Web page are grouped into a component tree, also known as view. There is a `UIViewRoot` component at the root of every view; otherwise, a view cannot exist. The `<f:view>` tag is used to create the `UIViewRoot` on a Web page; therefore, all components of a JSF page should be enclosed within the `<f:view>` and `</f:view>` tags. Table 11.30 lists the important attributes of the <f:view> tag:

Table 11.30: Showing the Common Attributes of the <f:view> Tag

Attribute	Description
locale	Allows you to set a Locale, which needs to be supported by a Web page. If the expression is given, it must evaluate to `java.util.Locale` or a string that needs to be converted to `Locale`.
renderKitId	Allows you to set the identifier for the RenderKit to be used.
beforePhase	Accepts a method binding expression and binds the expression to a method that is called before every phase except the Restore View phase. The method should take `javax.faces.event.PhaseEvent` object and return void.
afterPhase	Accepts a method binding expression and binds the expression to a method that is called after every phase except the Restore View phase. The method should take the `javax.faces.event.PhaseEvent` object and return void.

The following code snippet shows the use of the `<f:view>` tag:

```
<f:view>
  ..........Other JSF components of the JSF page.........
</f:view>
```

The <f:view> tag may contain other component tags, such as <h:form>, <h:commandButton> and many more. Let's now discuss the <f:attribute> tag in the next subsection.

The <f:attribute> Tag

The `<f:attribute>` tag is used to add an attribute, which is a key/value pair, to the closest parent component. The name attribute takes the name for the nearest parent component and the value attribute takes the value that needs to be set for the component attribute. Table 11.31 shows the most commonly used attributes of the `<f:attribute>` tag:

Table 11.31: Showing the Common Attributes of <f:attribute> Tag

Attribute	Description
Name	Specifies the name of the nearest parent component attribute
Value	Specifies the value of the nearest parent component attribute

The following code snippet shows the use of the <f:attribute> tag:

```
<h:commandButton id="cmd1">
  <f:attribute name="value" value="Add New"></f:attribute>
</h:commandButton>
```

The preceding code snippet uses <f:attribute> tag to add attribute (value) to the HtmlCommandButton component.

The <f:converter> Tag

The `<f:converter>` tag is used to register a converter instance for a component. There are various ways to register converters for a component, such as using the <f:converter> tag. The <f:converter> tag can be nested within a component by specify the name of the class, which implements the javax.faces.convert.Converter interface as a value of the converterId attribute of the <f:converter> tag. Table 11.32 shows the most commonly used attributes of the <f:converter> tag:

Table 11.32: Displaying the Common Attributes of the <f:converter> Tag

Attribute	Description
converterId	Specifies an ID for the Converter class, which is used during the custom conversion.
binding	Accepts a value expression that must evaluate to an object, which implements `javax.faces.convert.Converter` interface.

The following code snippet uses the <f:converter> tag:

```
<h:inputText value="#{student.rollno}" >
  <f:converter converterId="javax.faces.Integer"/>
</h:inputText>
```

In the preceding code snippet, the <f:converter> tag registers a converter instance.

The <f:convertDateTime> Tag

The `<f:convertDateTime>` tag is used to register datetime converter for a component. This represents a standard JSF validator that converts a string into the date object. Table 11.33 shows the most commonly used attributes of the <f:convertDateTime> tag:

Table 11.33: Showing the Common Attributes of the <f:convertDateTime> Tag

Attribute	Description
dateStyle	Allows you to set the predefined formatting style for the date string that needs to be displayed. The dateStyle style is applied only when the `type` attribute is set to either `date` or `both`. The allowed values for the `dateStyle` attribute are default, short, medium, long, and full.
locale	Allows you to set the Locale whose predefined style needs to be used while formatting and parsing.
pattern	Allows you to set custom formatting style as defined in the java.text.SimpleDateFormat class. This attribute defines how the date can be formatted.
timeStyle	Allows you to set the predefined formatting style for the time component of the date string. This is applied only when the type attribute is set to time or both (date or time). The allowed values for the timeStyle attribute are default, short, medium, long, and full.
timeZone	Allows you to set the time zone in which date and time can be formatted. By default it is GMT.
type	Specifies whether only date or date and time needs to be formatted. Its default value is date and it can take either `date` or `both`.
binding	Accepts a value expression, which evaluates to an instance of the javax.faces.convert.DateTimeConverter class.

The following code snippet shows the use of the <f:convertDateTime> tag:

```
<h:inputText id="dateofbirth" value="#{student.birthDate}" required="true">
  <f:convertDateTime type="date" pattern="dd-MM-yyyy"/>
</h:inputText>
```

In the preceding code snippet, the `<f:convertDateTime>` tag registers the date-time converter with the dateofbirth text field component.

The <f:convertNumber> Tag

The `<f:convertNumber>` tag is used to register a `NumberConverter` instance for the closest parent component. You can customize the input component using the `<f:convertNumber>` tag to take the specified number in a specific format type for currency, simple number, and percentage. Table 11.34 shows the commonly used attributes of the <f:convertNumber> tag:

Table 11.34:Showing the Common Attributes of the <f:convertNumber> Tag

Attribute	Description
currencyCode	Allows you to set the currency code as defined in ISO 4217, whenever you need to convert the currency into your desired format.
currencySymbol	Defines the currency symbol to be used.
groupingUsed	Accepts a boolean value, which indicates whether the formatted output contains grouping separators or not.
integerOnly	Accepts a boolean value, which indicates whether parsing and formatting of a value could be done on integer part.
locale	Defines the Locale whose predefined styles for numbers will be used.
maxFractionDigits	Allows you to set the maximum number of digits that needs to be formatted in the output's fractional portion.
maxIntegerDigits	Allows you to set the maximum number of digits that needs to be formatted in the output's integer portion.
minFractionDigits	Allows you to set the minimum number of digits that needs to be formatted in the output's fractional portion.
minIntegerDigits	Allows you to set the minimum number of digits that needs to be formatted in the output's integer portion.
pattern	Defines the custom formatting pattern.
type	Specifies the formatting style of the number. The allowed values are number, currency, and percentage. The default value is number.
binding	Accepts a value expression that evaluates to an instance of javax.faces.convert.NumberConverter class.

The following code snippet uses the <f:convertNumber> tag:

```
<h:inputText value="#{book.price}" >
  <f:convertNumber type="currency" currencySymbol="$"/>
</h:inputText>
```

The preceding code snippet uses the `<f:convertNumber>` tag to register the `NumberConverter` instance with price property of the book backing bean.

The <f:facet> Tag

The `<f:facet>` tag adds facet to a component and identifies a nested component that has a special relationship with its enclosing tag. This tag registers a named facet on the closest parent component. Table 11.35 shows the most commonly used attributes of the `<f:facet>` tag:

Table 11.35: Showing the Common Attribute of the<f:facet> Tag

Attribute	Description
name	Specifies the name of the facet to be created

The following code snippet shows the use of the `<f:facet>` tag:

```
<h:panelGrid id="panel" columns="2" border="1" rules="rows">
  <f:facet name="header">
      <h:outputText>Login Form</h:outputText>
  </f:facet>
  <f:facet name="footer">
      <h:outputText>Some footer Text.</h:outputText>
  </f:facet>
```

```
        <h:outputLabel for="uid">User ID</h:outputLabel>
        <h:inputText id="uid" value="#{user.userName}"/>
        <h:outputLabel for="pwd">Password</h:outputLabel>
        <h:inputText id="pwd" value="#{user.password}"/>
    </h:panelGrid>
```

The preceding code snippet uses the `<f:facet>` tag inside the `<h:panelGrid>` tag to add header and footer to a panel grid, named panel.

The <f:loadBundle> Tag

The `<f:loadBundle>` tag loads the resource bundle and stores it in the request scope. The key/value pairs from the resource bundle are stored as `Map`. You can directly use the name of the variable assigned to resource bundle to access the localized messages in other component tags used in the JSF page. Table 11.36 shows the most commonly used attributes of the `<f:loadBundle>` tag:

Table 11.36: Showing the Common Attributes of the <f:loadBundle> Tag

Attribute	Description
basename	Defines the base name of the resource bundle to be loaded.
var	Specifies the reference name of a request scope attribute, and under that reference name the resource bundle will be exposed as a Map.

The following code snippet uses the `<f:loadBundle>` tag:

```
<f:loadBundle basename="com.kogent.ApplicationMessages" var="msg"/>
   .... .... ....
        <h:commandLink action="addnew" value="#{msg.addnew}"/>
```

The preceding snippet uses the `<f:loadBundle>` tag to display the text messages written inside the `ApplicationMessages.properties` file on JSF UI components.

The <f:param> Tag

At times, you may need to specify additional parameters for a component. This can be done by using the `UIParameter` component, which is created by using the `<f:param>` tag. The parameters are added to enclosing component using a name/value pair created by the `<f:param>` tag. Table 11.37 shows the most commonly used attributes of the <f:param> tag:

Table 11.37: Showing the Common Attributes of the <f:param> Tag

Attribute	Description
binding	Accepts a value binding expression to link the specified value to the backing bean property and this value is later bound to the component of the UI component created by the current custom action.
id	Defines the component identifier.
name	Defines the name of the parameter to be created.
value	Allows you to set the value of a parameter.

The following code snippet shows the use of the <f:param> tag:

```
<h:outputLink value="edit.faces">
   <h:outputText value="Edit"/>
   <f:param name="roll" value="#{student.rollno}"/>
</h:outputLink>
```

The preceding code snippet displays a hyperlink (named Edit) on browser. When the user clicks the hyperlink, roll parameter specified using <f:param> tag is also passed as a parameter to the URL request to edit.faces Web page.

The <f:selectItem> Tag

The `<f:selectItem>` tag is a `UISelectItem` component and provides a single option to select one or more components. The `UISelectItem` component is not displayed by default. Its rendering depends on the enclosing parent component. The `<f:selectItem>` tag element may render to checkbox, radio button, or items in the list box or the drop-down list. Table 11.38 shows the most commonly used attributes of the `<f:selectItem>` tag:

Table 11.38: Showing the Common Attributes of the <f:selectItem> Tag

Attribute	Description
binding	Accepts a value binding expression to a backing bean property bound to the component.
id	Defines the component identifier.
itemDescription	Allows you to provide the description of the selectItem instance.
itemDisabled	Accepts a boolean value indicating whether the option created by this tag will be disabled or not.
itemLabel	Specifies the label that needs to be displayed for the selectItem instance.
itemValue	Allows you to set the value, which needs to be submitted to the server if an option is selected from the `selectItem` instance.
value	Accepts a value binding expression, which points to the SelectItem instance.

The following code snippet shows the use of the `<f:selectItem>` tag:

```
<h:selectOneMenu id="department" value="#{employee.department}" required="true">
  <f:selectItem itemValue="prod" itemLabel="Production"/>
  <f:selectItem itemValue="test" itemLabel="Testing"/>
  <f:selectItem itemValue="sal" itemLabel="Sales"/>
</h:selectOneMenu>
```

The preceding code snippet uses the `<f:selectItem>` tag to represent each option of the `HtmlSelectOneMenu` component.

The <f:selectItems> Tag

The `<f:selectItems>` tag works similar to the `<f:selectItem>` tag, but this tag provides a list of options to select one or more components. The `<f:selectItems>` tag corresponds to the UISelectItems component, which is used to configure multiple choices in one request. Table 11.39 shows the most commonly used attributes of the `<f:selectItems>` tag:

Table 11.39: Showing the Common Attributes of the <f:selectItems> Tag

Attribute	Description
binding	Accepts a value binding expression to a backing bean property bound to the component.
id	Defines the component identifier.
value	Accepts a value binding expression, which points to a List or array of SelectItem instances.

The following code snippet shows the use of the `<f:selectItems>` tag:

```
<h:selectOneRadio id="cities" value="#{student.city}">
  <f:selectItems value="#{citybean.options}"/>
</h:selectOneRadio>
```

In the preceding code snippet, the `<f:selectItems>` tag represents all the cities mentioned as options in the `HtmlSelectOneRadio` component.

Let's create a managed bean named citybean in the faces-config.xml file. The following code snippet shows the code to create the CityBean.java file:

```
package com.kogent;
import java.util.ArrayList;
import java.util.List;
import javax.faces.model.SelectItem;

public class CityBean{
  private List cities;
  public CityBean(){
        cities = new ArrayList();
        SelectItem city = new SelectItem("del", "Delhi");
        cities.add(city);
        city = new SelectItem("fdb", "Faridabad");
        cities.add(city);
        city = new SelectItem("lck", "Lucknow");
        cities.add(city);
  }

  public void setOptions(List cities){
        this.cities =cities ;
  }
  public List getOptions(){
        return this.cities;
  }
}
```

In preceding code snippet, the CityBean class has a getter and setter method corresponding to the <f:selectItems> tag. Let's discuss the <f:subview> tag in next subsection.

The <f:subview> Tag

The `<f:subview>` tag creates a subview of a view. It works as a container for all JSF core and custom components that are used in a nested page included by using the `<jsp:include>` tag or JSTL's `<c:import>` tag. Table 11.40 shows the most commonly used attributes of the <f:subview> tag:

Table 11.40: Showing the Common Attributes of the <f:subview> Tag

Attribute	Description
binding	Accepts a value binding expression to a backing bean property bound to a component
id	Defines component identifier
rendered	Allows you to set a boolean value, that specifies whether a component will be rendered or not

The following code snippet shows the use of the `<f:subview>` tag:

```
<f:subview id="footer">
  <jsp:include page="footer.jsp">
</f:subview>
```

The `<f: subview>` tag represents footer as a subview of a Web page on a browser. Let's discuss the <f:validateDoubleRange> tag in the next subsection.

The <f:validateDoubleRange> Tag

The `<f:validateDoubleRange>` tag is used to register the `DoubleRangeValidator` instance on the input UI component. The `DoubleRangeValidator` validator validates the current value of a component for the given range of specified value. Table 11.41 shows the most commonly used attributes of the `<f:validateDoubleRange>` tag:

Table 11.41: Showing the Common Attributes of the <f:validateDoubleRange> Tag	
Attribute	**Description**
maximum	Sets the maximum value allowed for this component
minimum	Sets the minimum value allowed for this component
binding	Accepts a value expression, which must evaluate to an instance of DoubleRangeValidator

The following code snippet shows the use of the <f:validateDoubleRange> tag:

```
<h:inputText id="num" value="#{bean.number}" required="true">
  <f:validateDoubleRange minimum="2.0" maximum="10.0"/>

</h:inputText>
```

Let's learn about the <f:validateLength> tag.

The <f:validateLength> Tag

The `<f:validateLength>` tag is used to register the `LengthValidator` instance on the input UI component. Using the `<f:validateLength>` tag with an input component specifies the length of the input string. Table 11.42 shows the commonly used attributes of the `<f:validateLength>` tag:

Table 11.42: Showing the Common Attributes of the <f:validateLength> Tag	
Attribute	**Description**
maximum	Sets the maximum length specified for this component
minimum	Sets the minimum length specified for this component
binding	Accepts a value expression, which must evaluate to an instance of LengthValidator

The following code snippet shows the use of the `<f:validateLength>` tag:

```
<h:inputSecret id="pwd" label="Password" value="#{user.password}" required="true">
  <f:validateLength minimum="4" maximum="10"/>

</h:inputSecret>
```

Let's learn about the <f:validateLongRange> tag.

The <f:validateLongRange> Tag

The `<f:validateLongRange>` tag is used to register `LongRangeValidator` on a component. Using the `<f:validateLongRange>` tag with the input component specifies the range of the input value. Table 11.43 shows the commonly used attributes of the `<f:validateLongRange>` tag:

Table 11.43: Showing the Common Attributes of the <f:validateLongRange> Tag	
Attribute	**Description**
maximum	Sets the maximum value specified for this component
minimum	Sets the minimum value specified for this component
binding	Accepts a value expression, which must evaluate to an instance of LongRangeValidator

The following code snippet shows the use of the `<f:validateLongRange>` tag:

```
<h:inputText id="num" value="#{bean.number}">
  <f:validateLongRange maximum="30" minimum="15"/>

</h:inputText>
```

Let's learn about the <f:validator> tag.

The <f:validator> Tag

The `<f:validator>` tag is used to register a custom validator on an input UI component. You need to create a custom validator class implementing the `javax.faces.validator.Validator` interface and configure it in the `faces-config.xml` file, so that the custom validator class can be used by the `<f:validator/>` tag. Table 11.44 shows the commonly used attributes of the `<f:validator>` tag:

Table 11.44: Showing the Common Attributes of the <f:validator> Tag

Attribute	Description
validatorId	Specifies the validator identifier of the Validator class that needs be created and registered on the component.
binding	Accepts a value expression, which must evaluate to an object that implements javax.faces.validator.Validator interface

The following code snippet shows the use of the `<f:validator>` tag:

```
<h:inputText id="email" required="true" value="#{student.email}">
  <f:validator validatorId="MyEmailValidator" />

</h:inputText>
```

The preceding code snippet registers a custom validator class (MyEmailValidator) with a text field component (email). You need to manually create the `MyEmailValidator` class to implement the `javax.faces.validator.Validator` interface. The `MyEmailValidator` class should to be configured in the `faces-config.xml` file using the `validator`, `validator-id`, and `validator-class` elements.

The <f:verbatim> Tag

The `<f:verbatim>` tag is used to create as well as register the `UIOutput` child component on the `UIComponent` parent component. All the components inside the <f:view> tag should be provided in JSF component tree; therefore, the HTML and JSP tags can be nested with <f:verbatim> tag. It solves the problem occurred when a page contains both the JSF elements as well as the HTML elements and you can not control the HTML elements programmatically. You cannot use any JSF components within the `<f:verbatim>` tag. Table 11.45 shows the commonly used attributes of the `<f:verbatim>` tag:

Table 11.45: Showing the Common Attributes of the <f:verbatim> Tag

Attribute	Description
escape	Accepts a boolean value indicating whether the generated markup should be escaped or not. The default value of the `escape` attribute is false.
rendered	Accepts a boolean value indicating whether the component will be rendered or not. The default value of the `rendered` attribute is true.

The following code snippet shows the use of the `<f:verbatim>` tag:

```
<f:verbatim>
  <h3>Some Heading</h3>

</f:verbatim>
```

In the preceding code snippet, the <f:verbatim> tag is used to provide the head in the HTML page. Let's now discuss the <f:phaseListener> tag.

The <f:phaseListener> Tag

The `<f:phaseListener>` tag is used to register a `PhaseListener` instance on the `UIViewRoot` component in which it is enclosed. It registers a phase listener for the current page or view. Table 11.46 shows the most commonly used attributes of the <f:phaseListener> tag:

Table 11.46: Showing the Important Attributes of the <f:phaseListener> Tag	
Attribute	**Description**
type	Accepts a fully qualified name of the phase listener class that needs to be created and registered for the current Web page.
binding	Accepts a value expression that should bind to an object, which implements the javax.faces.event.PhaseListener interface.

The following code snippet shows the use of the `<f:phaseListener>` tag:

```
<f:view>
  <f:phaseListener type = "com.kogent.MyPhaseListener"/>
</f:view>
```

In the preceding code snippet, a phase listener class is used. This class can be created by implementing the `javax.faces.event.PhaseListener` interface. The structure of the `MyPhaseListener` class is shown in the following code snippet:

```
package com.kogent;

import javax.faces.event.PhaseEvent;

import javax.faces.event.PhaseId;

import javax.faces.event.PhaseListener;

public class MyPhaseListener implements PhaseListener

{
  public void beforePhase(PhaseEvent event){

  }
  public void afterPhase(PhaseEvent event){

  }

  public PhaseId getPhaseId(){
        return PhaseId.ANY_PHASE;
  }

}
```

In the preceding code snippet, the methods, such as beforePhase(), afterPhase(), and getPhaseId() of the PhaseListener interface are implemented.

The <f:setPropertyActionListener> Tag

The `<f:setPropertyActionListener>` tag creates a special `ActionListener` instance and registers it with the associated `ActionSource`, such as `HtmlCommandButton`. The <f:setPropertyActionListener> tag is used to create and register the instance of ActionListener, when the instance of this component is created for the first time. Table 11.47 shows the commonly used attributes of the `<f:setPropertyActionListener>` tag:

Table 11.47: Showing the Common Attributes of the <f:setPropertyActionListener> Tag	
Attribute	**Description**
value	Accepts a value expression that is used to store the value of target attribute
target	Accepts a value expression that defines the destination of value attribute

The following code snippet shows the use of the `<f:setPropertyActionListener>` tag:

```
<h:commandLink action="#{student.delete}" value="Delete">
  <f:setPropertyActionListener target="#{student.roll}" value="#{user.roll}"/>

</h:commandLink>
```

In the preceding code snippet, the roll property of the backing bean, student is set with the value of roll property of user bean before the action invoke phase, that is before the `delete()` method of student backing bean is executed.

You have learned about various HTML and Core tags provided by JSF to build UI components on a Web page. Now, let's learn about the JSF standard components.

JSF Standard UI Components

JSF is a UI framework, implying that it not only provides some tags to create and customize UI components on the Web page, but also enables programmers to access these components within the code. Therefore, all the UI components have their corresponding component classes, which can be created and manipulated with the code to provide logical behavior of the components on the Web page. While discussing different HTML and Core JSF tags, the corresponding component classes have been specified, for example HtmlInputText, HtmlInputTextarea, HtmlCommandButton. The JSF standard components are categorized into different goups according to their characteristics as well as super classes. For example, the classes, such as HtmlInputHidden, HtmlInputSecret, HtmlInputText, and HtmlInputTextarea are provided under Input family as they all extend the base class UIInput and are used to take input string from the user. Table 11.48 lists all base component classes with their categories and different HTML subclasses:

Table 11.48: Describing the Base UI Component Classes and their HTML Subclasses

Class	Family	HTML Subclasses	Description
UIComponent			Serves as an abstract class for all components.
UIComponentBase			Serves as an abstract base class with basic implementations of almost all UIComponent methods.
UIColumn	Column		Represents table column, which is used to configure template columns for the parent UIData component.
UICommand	Command	HtmlCommandButton, HtmlCommandLink	Represents the user command,
UIData	Data	HtmlDataTable	Represents a data-aware component that cycles through rows in the underlying data source and exposes individual rows to child components. Requires child UIColumn components,
UIForm	Form	HtmlForm	Represents an input form, which must enclose all input components,
UIGraphic	Image	HtmlGraphicImage	Displays an image based on its URL
UIInput	Input	HtmlInputHidden, HtmlInputSecret, HtmlInputText, HtmlInputTextarea	Accepts input, processes it, and then displays the output.

Table 11.48: Describing the Base UI Component Classes and their HTML Subclasses

Class	Family	HTML Subclasses	Description
UIMessage	Message	HtmlMessage	Displays messages for a specific component.
UIMessages	Messages	HtmlMessages	Displays all messages, component related and/or application-related.
UIOutput	Output	HtmlOutputFormat, HtmlOutputLabel, HtmlOutputLink, HtmlOutputText	Holds a read-only value and displays it to the user.
UIParameter	Parameter		Represents a parameter for a parent component.
UIPanel	Panel	HtmlPanelGrid, HtmlPanelGroup	Groups a set of child components together.
UISelectBoolean	Checkbox	HtmlSelectBooleanCheckbox	Collects and displays a single boolean value.
UISelectItem	SelectItem		Represents a single item or item group. The instance may be nested with UISelectMany or UISelectOne component. This leads to the addition of SelectItem instances to the parent component of option.
UISelectItems	SelectItems		Represents multiple items or item groups. The instance may be nested with UISelectMany or UISelectOne component. This leads to the addition of SelectItem instances to the parent component of option.
UISelectMany	SelectMany	HtmlSelectManyCheckbox, HtmlSelectManyListbox, HtmlSelectManyMenu	Displays a set of items, and allows the user to select zero or any number of items.
UISelectOne	SelectOne	HtmlSelectOneRadio HtmlSelectOneListbox HtmlSelectOneMenu	Displays a set of items, and allows the user to select only one of them.
UIViewRoot	ViewRoot		Represents entire view; contains all components on the Web page.

Figure 11.5 show the class hierarchy of JSF standard components. It shows different UI component classes that have been extended from the `javax.faces.component.UIComponentBase` class, which implements the default behavior of all methods defined by the `javax.faces.component.UIComponent` class. The `UIComponent` class is an abstract base class for all UI components in JSF. Figure 11.5 shows the hierarchical representation of the JSF standard components:

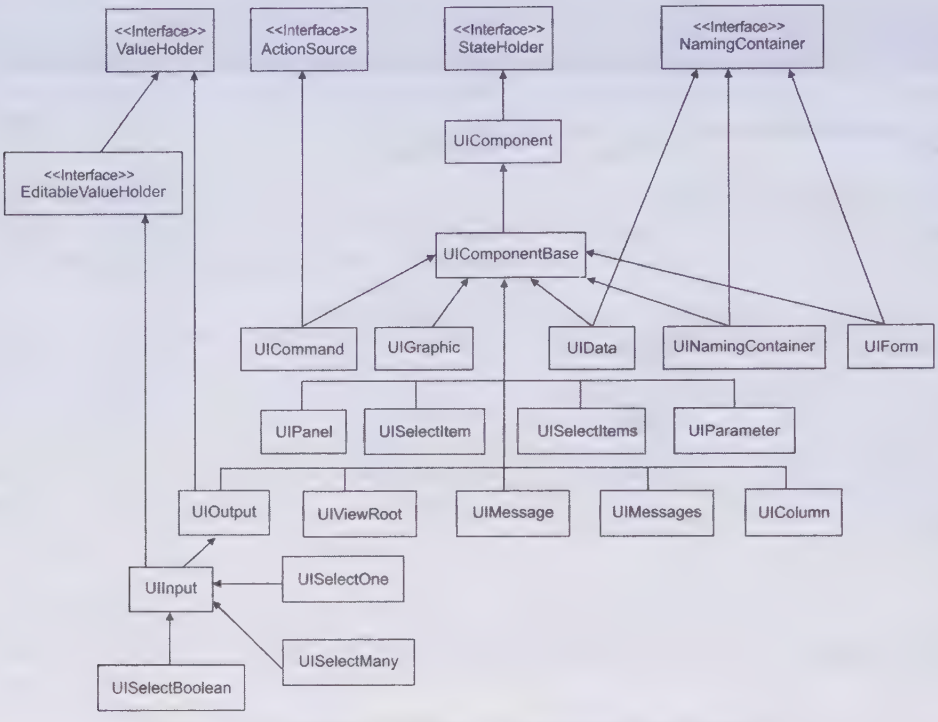


Figure 11.5: Displaying the Base UI Component Class Hierarchy

In Figure 11.5, some of the UI components extend different base classes while some of them implement different interfaces, which provide different behavior to those base classes. For example, the `UIInput` class extends from the `UIOutput` class and implements the `javax.faces.component` package. The `EditableValueHolder` interface provides additional features supported by editable components. These features are further extended by the subclasses, such as `HtmlInputText` and `HtmlInputTextarea`.

Now, let's briefly discuss the various components provided in abstract base class for all UI components (Figure 11.5).

Command Components

We have two command components, `HtmlCommandButton`, and `HtmlCommandLink`. These components or classes have been extended from the `javax.faces.component.UICommand` class and are capable of generating action events, means they behave as action source. The creation of these classes in the JSF page has been discussed earlier while discussing `<h:commandButton>` and `<h:commandLink>`.

Data Component

The `UIData` is a component that supports data binding to a collection of data objects, such as `ArrayList` and `ResultSet`. The `javax.faces.component.html.HtmlDataTable` is the direct subclass of the `UIData` class. The `HtmlDataTable` class is used to create a standard data grid with the iteration of different objects in the set of data objects provided in JSF and creates individual rows of the HTML table for each object in the set. The tag used to create the HtmlDataTable component is `<h:dataTable>`.

Form Component

The `javax.faces.component.html.HtmlForm` is a subclass of the `UIForm` class that represents an input form. All the child components of this component represent input fields and are submitted to the server with the submission of a form. The HtmlForm component renders to simple HTML `<form>` element.

Image Component

The only component in the image family is `javax.faces.component.html.HtmlGraphicImage`, which extends the `UIGraphic` class. It is used to display a graphical image to the user and this graphical image cannot be manipulated by the user. The `HtmlGraphicImage` component renders to HTML `<img>` element and the tag used for its creation is `<h:graphicImage>`.

Input Component

The base class for all input components is javax.faces.component.UIInput and its different HTML subclasses are HtmlInputHidden, HtmlInputSecret, and HtmlInputText, and HtmlInputTextarea. The HtmlInputHidden, HtmlInputSecret, and HtmlInputText components render to the HTML <input> element with the typeset as hidden, password, and text while the HtmlInputTextarea component renders to HTML <textarea> component. The tags for all these subclasses have been discussed earlier in this chapter.

Message and Messages Component

The components `javax.faces.component.html.HtmlMessage` and `javax.faces.component.html.HtmlMessages` are used to display different messages added into FacesContext during various processes, such as validation, conversion, and invocation of action method. The `HtmlMessage` and `HtmlMessages` classes extend the `UIMessage` and `UIMessages` classes, respectively. The tags used for message and message components are `<h:message>` and `<h:messages>`, respectively.

Output Component

The output components include `HtmlOutputFormat`, `HtmlOutputLabel`, `HtmlOutputLink`, and `HtmlOutputText` component classes. These component classes extend the `javax.faces.component.UIOutput` component class. The different tags to create the output components are `<h:outputFormat>`, `<h:outputLabel>`, `<h:outputLink>`, and `<h:outputText>` and the characteristics of these output components are discussed earlier in the *JSF HTML tags* section. These components hold a read-only value and display it to the user.

Parameter Component

The `javax.faces.component.UIParameter` component represents a parameter for the parent component. `UIParameter` extends `UIComponentBase` base class and does not have any HTML subclass, as it is not rendered in the response. The core JSF tag used to create this component is `<f:param>`.

Checkbox Component

In the checkbox family of component, the base component class is `UISelectBoolean` and the only HTML subclass is `javax.faces.component.html.HtmlSelectBooleanCheckbox`. This component collects and displays single boolean value. The tag used for this component is `<h:selectBooleanCheckbox>`.

SelectItem and SelectItems Component

The base classes for the SelectItem and SelectItems groups are UISelectItem and UISelectItems, respectively. These classes do not have any HTML subclass, as they do not render themselves. Their rendering behavior is controlled by the parent component. The core JSF tags for these two components are <f:selectItem> and <f:selectItems>, respectively.

SelectMany and SelectOne Component

The base component for the `SelectMany` and `SelectOne` components are `UISelectMany` and `UISelectOne` respectively. The SelectMany family represents the components that allow users to select zero or more items from the given number of items. The HTML subclasses of the `UISelectMany` component class include the `HtmlSelectManyCheckbox`, `HtmlSelectManyListbox`, and `HtmlSelectManyMenu` classes.

Similarly, the `SelectOne` component allows users to select a single option from the given set of choices. The HTML subclass of the `UISelectOne` component class includes `HtmlSelectOneListbox`,

HtmlSelectOneMenu, and HtmlSelectOneRadio. The tags to create these components have been discussed with their different attributes and examples earlier in the chapter.

ViewRoot Component

The components in a JSF view are organized in a component tree, which is headed by the javax.faces.component.UIViewRoot component at the root. The UIViewRoot is represented by the <f:view> tag. Using the UIViewRoot component, you can set and retrieve the current render kit, Locale for the page, and its viewId property. You can bypass the navigation system and write your own logic for navigation by creating a new instance of UIViewRoot, and calling the setViewRoot() method on the FacesContext object.

You have learned about different UIComponent classes in which you have learned about different component classes that are created and manipulated internally, while using their corresponding tags provided by JSF to place these components in JSF view. Though the methods and properties of these component classes have not been discussed in this chapter, as they are out of scope of this book, but you can use the standard naming conventions to interact and manipulate these components in your code.

Working with Backing Beans

Backing bean is a simple JavaBean that stores data at an intermediate stage and facilitates interaction between the view and model layer. A backing bean is used with a set of properties to map each property to the value of some component in a view or to the component itself. In addition to the get and set methods for these properties, the backing bean can have various methods to support event handling and validation of the data before the application logic is executed or model layer is interacted for database operations. These backing beans are created and managed automatically by JSF, which is configured by the developer in the faces-config.xml file; therefore, the backing beans are also known as Managed beans. Note that all backing beans should be managed but all managed beans are not supposed to be backing beans. Let's learn to develop a backing bean to use with a given view. The following code snippet shows code for a JSF page with different components:

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
  <head>
        <title>Welcome</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
  </head>
  <body>
        <f:view>
              <h:form id="studentForm">
                    <h:outputText id="message" value="Welcome, Guest!"
                    binding="#{student.outputText}"/>
                    <h:panelGrid id="panel" columns="2" border="1">
                    <h:outputText value="Roll No." />
                    <h:inputText id="rollno" value="#{student.rollno}" />
                    <h:outputText value="Name" />
                    <h:inputText label="Name" id="name" value="#{student.name}" />
                    <h:commandButton id="button1"
                    actionListener="#{student.showMessage}"
                    value="Show Message"/>
                    <h:commandButton id="button2"
                    action="#{student.showDetail}" value="Show Detail"/>
                    </h:panelGrid>
              </h:form>
        </f:view>
  </body>
</html>
```

In preceding code snippet, we have created an HtmlForm component with different child components, such as HtmlInputText and HtmlCommandButton. The values of two input field components, rollno and name are bound to two different properties of student backing bean.

In preceding code snippet, we have bound an `HtmlOutputText` component (with id message) with another property, `outputText`, of student backing bean. You can bind a backing bean property with the value of a component by using the value expression for the value attribute; however to bind the value with a component, you can use value expression for binding attribute of the component.

The following code snippet shows the structure of student backing bean with its three properties along with get and set methods:

```
package com.kogent;

import javax.faces.component.html.HtmlOutputText;
import javax.faces.event.ActionEvent;

public class Student {
  private String rollno;
  private String name;

  private HtmlOutputText outputText;

  public String getName() {
        return name;
  }
  public void setName(String name) {
        this.name = name;
  }
  public String getRollno() {
        return rollno;
  }
  public void setRollno(String rollno) {
        this.rollno = rollno;
  }
  public HtmlOutputText getOutputText() {
        return outputText;
  }


  public void setOutputText(HtmlOutputText outputText) {
        this.outputText = outputText;
  }

  public void showMessage(ActionEvent event){
        //Some application logic..
        outputText.setValue("Welcome "+this.name);
  }

  public String showDetail(){
        //Some application logic..
        return "success";
  }
}
```

In preceding code snippet, the backing bean class is `com.kogent.Student` and we are referring it with the name `student` in the JSF page, which is a reference name for the `Student` bean when managed in the `faces config.xml` file. All the three properties of student backing bean are automatically populated during the Apply Request Values phase. The binding of a component with backing bean property makes the component accessible in the backing bean to be manipulated logically.

Using the Backing Bean Method as an Event Handler

A backing bean can have various methods that can be used as event listeners. We can handle two types of events through backing bean methods, which are action events and value change events.

A backing bean can be used as action listener. The action listeners are of two types, one that affects navigation and other does not.

The backing bean method, which can be used as an action listener and does not affect navigation must have a signature `void methodName(ActionEvent event)`. The `void showMessage(ActionEvent event)` method that has been created in the Student class does not affect navigation and the current view is redisplayed, as given in the following code snippet:

```
public void showMessage(ActionEvent event){
  //Some application logic..
  outputText.setValue("Welcome "+this.name);
}
```

The methods, such as methodName(ActionEvent event) and showMessage(ActionEvent event) are commonly referred as action listener methods and are registered with the action resources, such as `HtmlCommandButton`, using its `actionListener` attribute, as given in the following code snippet:

```
<h:commandButton id="button"
actionListener="#{student.showMessage}" value="Submit"/>
```

Further, there are other types of backing bean methods, which can be registered with action resources and are used by some default action listener affecting navigation from one view to another. Such backing bean methods are known as action methods and can be associated with the action resource component using its action attribute, as shown in the following code snippet:

```
<h:commandButton id="button2"
action="#{student.showDetail}" value="Show Detail"/>
```

The signature of an action method must be `String methodName()`. The string returned by the action method, which is an outcome of some application logic execution is further used by navigation system to decide the next view to be displayed. The following code snippet shows the implementation of the `showDetail()` method created in the `Student` class:

```
public String showDetail(){
  //Some application Logic
  return "success";
}
```

In addition, you can also register a backing bean method as value change event listener; therefore, the signature of a registered method must be `void methodName(ValueChangeEvent event)`. This method will be fired with a value change event only when there is a change in the value of the associated component. You can register a value change listener with different input components such as `HtmlInputText` and `HtmlInputTextarea` by giving a method expression for their `valueChangeListener` attribute, as shown in the following code snippet:

```
<h:inputText id="rollno" value="#{student.rollno}"
valueChangeListener="#{student.rollChanged}"/>
```

In the preceding code snippet, we have registered a backing bean method `rollChanged()` as `ValueChangeEvent` listener. This method is invoked before the invocation of any action method or action listener method. The following code snippet shows the code that add the backing bean method, `rollChanged()` to the `Student` class as a `ValueChangeEvent` listener:

```
public void rollChanged(ValueChangeEvent event){
  HtmlInputText component=(HtmlInputText)event.getComponent();
  if(((String)event.getNewValue()).length()!=3)
  component.setStyle("color:red");
  else
  component.setStyle("color:black");
}
```

Let's learn to use backing bean as validator.

Using Backing Bean Method as Validator

We have different standard validators provided by the JSF framework. These validators can be used to validate the values of different input components submitted by the user. However, JSF provides another way to validate the input components, that is, by using a backing bean method as a validator method. A backing bean method having signature, `void methodName(FacesContext, UIComponent, Object)` can be registered as a validator with an input component. the following code snippet shows an example of validator method created in backing bean:

```
public void validateRollNo(FacesContext facesContext, UIComponent uiComponent,
  Object value) throws ValidatorException{
  //Some Logic to validate value
  HtmlInputText htmlInputText = (HtmlInputText) uiComponent;
  FacesMessage facesMessage = new FacesMessage(htmlInputText.getLabel()+
  ": Some Validation Error");
  throw new ValidatorException(facesMessage);
}
```

In the preceding code snippet, the `validateRollNo()` method simply executes a logic to validate the given value; and if the value is invalid, this method creates an instance of the `FacesMessage` class and throws `ValidatorException`, which stops further execution of the JSF life cycle phases and the current page is redisplayed with some error message using the `<h:messages/>` tag element.

Managing Backing Beans

The backing beans need to be created and initialized automatically using the Managed Bean Creation facility. This facility allows the declaration of different beans with their initialization in a central location, i.e., at the JSF configuration file. You can also define the scope of a bean, such as request, session, or application. All managed beans can be accessed using JSF EL. You can configure a managed bean with the `<managed-bean>` element, as shown in the following code snippet:

```
<managed-bean>
  <managed-bean-name>student</managed-bean-name>
  <managed-bean-class>com.kogent.Student</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

While declaring a managed bean, you need to specify its name, class, and scope. The `<managed-bean-name>` element defines the reference name, which can be used in JSF EL to access a bean. You can also initialize simple properties while declaring managed beans by using the `<managed-property>` element, as shown in the following code snippet:

```
<managed-bean>
  <managed-bean-name>student</managed-bean-name>
  <managed-bean-class>com.kogent.Student</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>name</property-name>
    <value>Prakash</value>
  </managed-property>
</managed-bean>
```

In addition to simple bean properties, you can also initialize `List` or `Map` type of properties using `<list-entries>` and `<map-entries>` respectively. For example, a bean having a List type property can be initialized as `cities` and `Map` type of property as Locales, as shown in the following code snippet:

```
<managed-bean>
  <managed-bean-name>citybean</managed-bean-name>
  <managed-bean-class>com.kogent.CityBean</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <managed-property>

      <property-name>cities</property-name>
```

```
        <list-entries>
                <value>Delhi</value>
                <value>Pune</value>
                <value>Mumbai</value>
        </list-entries>
    </managed-property>
    <managed-property>
        <property-name>locales</property-name>
        <map-entries>

                <map-entry>
                        <key>en</key>
                        <value>English</value>
                </map-entry>
                <map-entry>
                        <key>fr</key>
                        <value>French</value>
                </map-entry>
        </map-entries>
    </managed-property>
</managed-bean>
```

You can also declare `ArrayList` and Maps as managed beans by setting `<managed-bean-class>` as `java.util.ArrayList` or `java.util.HashMap`, as shown in the following code snippet:

```
<managed-bean>
    <managed-bean-name>cities</managed-bean-name>
    <managed-bean-class>java.util.ArrayList</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <list-entries>
        <value>Delhi</value>
        <value>Mumbai</value>
        <value>Kolkata</value>
    </list-entries>
</managed-bean>
```

You can also refer to one managed bean while setting the property of another managed bean using a value binding expression.

JSF Input Validation

In a Web application, the user input validation is required before executing any application logic. The invalid data submitted by the user may cause unexpected errors and may lead to inconsistency; therefore, a framework is preferred. A framework supports user input validation in a way that it does not affect other application logic development. JSF framework can validate user submitted data in the Process Validation phase, which comes before the Update Model Values phase and Invoke Application phase; this reduces the risk of updating model layer with any invalid data input. JSF being a UI framework equips all input component classes and corresponding tags to handle the validation of values. JSF provides following two basic ways to apply validation on the components that are used for taking inputs:

- Using validator method in backing bean
- Using validators

Using Validator Method

You have learned to create a validator method in a backing bean and associating it with input component using a method expression for its validator attribute. The signature of the validator method must be `void methodName(FacesContext, UIComponent, Object) throws ValidatorException`. You can use a `validator` attribute, `validateEmail()` of backing bean, `student`, with an input component, as shown in the following code snippet:

```
<h:inputText id="email" value="#{student.email}" size="30"
validator="#{student.validateEmail}" />
```

You can define the `validateEmail()` method in the backing bean class, as shown in the following code snippet:

```
public void validateEmail(FacesContext facesContext, UIComponent uiComponent,
Object value) throws ValidatorException
{
  //Logic to validate value for being a valid email id.
}
```

You can create a number of validator methods in a backing bean. The only disadvantage of defining a validator method in a backing bean is that you can bind only one validator method with a component in a Web page. This also makes backing bean more complex.

Using Validators

Another way to validate the value of an input component is to associate the value with custom validators class, which implements the `javax.faces.validator.Validator` interface. JSF provides its own set of validators, known as standard JSF validators. You can also create your own validator classes. The validators can be associated with a component in following two ways:

- Using the custom tag of validator inside the component tag.

The following code snippet shows an example of using the custom tag of validator to associate a specific validator with an input component:

```
<h:inputText id="uid" label="User ID" value="#{user.userName}">
  <f:validateLength minimum="4" maximum="10"/>
</h:inputText>
```

The preceding code snippet uses the <f:validateLength> custom tag of validator.

- Using the `<f:validator>` tag inside the component tag. The following code snippet shows the use of the `<f:validator>` tag:

```
<h:inputText id="uid" label="User ID" value="#{user.userName}" >
  <f:validator validatorId="someID"/>
</h:inputText>
```

The preceding code snippet uses the <f:validator> element with the identifier of validator.

The validators, which do not have customized tags, can be implemented by using the `<f:validator>` tag.

You can associate single input components with multiple validators, which is not possible when a validator method is used.

Let's discuss the two types of validators, standard JSF validators and custom validators.

Standard JSF Validators

JSF provides various standard validators to support some common validation rules, such as validating string for length, validating input value whether it falls under the given range or not. There is a set of three standard JSF validators, which are listed in Table 11.49:

Table 11.49: Showing the Standard JSF Validators

Validator Class	Validator's Custom Tag	Properties
LengthValidator	<f:validateLength>	maximum, minimum
DoubleRangeValidator	<f:validateDoubleRange>	maximum, minimum
LongRangeValidator	<f:validateLongRange>	maximum, minimum

These validators can be used with their custom tags.

We have already described these standard validator tags in detail while discussing JSF Core Tags earlier in this chapter.

Custom Validators

If the standard validators do not satisfy the validation criteria that you want to apply for a component, you can create custom validators. For example, JSF do not provide any validator to validate the email id of a user. In such cases, JSF allows you to create custom validator classes. You can create a validator class by implementing the `javax.faces.validator.Validator` interface and defining its only method , `validate()`.The following code snippet shows the code to create custom validator class:

```
package com.kogent.validator;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.html.HtmlInputText;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
public class MyValidator implements Validator {
  public void validate(FacesContext facesContext, UIComponent uiComponent,
  Object value) throws ValidatorException {

        boolean valid=true;
        //Some logic to validate the value.
        .. .. ..
        .. .. ..
        if (!valid){
                HtmlInputText htmlInputText = (HtmlInputText) uiComponent;
                FacesMessage facesMessage = new FacesMessage(htmlInputText.getLabel()+
                 ": Email Format is not valid");
                throw new ValidatorException(facesMessage);
        }
  }
}
```

The following code snippet shows the code to use the custom validator, MyValidator by registering it in the `faces-config.xml` file using the `<validator>` element. by providing an identifier for it

```
<validator>
  <validator-id>myValidator</validator-id>
  <validator-class>com.kogent.validator.MyValidator </validator-class>
</validator>
```

The value provided in the preceding code snippet with `<validator-id>` is used to associate the validator, MyValidator with the input component, which needs to be validated. The use of the <h:inputText> tag is shown in the following code snippet:

```
<h:inputText id="empid" value="#{employee.id}">
  <f:validator validatorId="myValidator"/>
</h:inputText>
```

You have learned about the JSF input validation support provided in terms of validator methods, standard and custom validators.

JSF Type Conversion

The user input is normally a text string, which submitted to the server for further processing. However, when request parameters are mapped to different backing bean properties internally, the string values need to be converted to different Java types, such as Integer, boolean, Short, and Double. On the other hand, when the response is created for the user, these Java types are again changed to strings and in this case JSF converters are used. The JSF converters create the string presentation of an object and object presentation of a string. If the converter is unable to convert any value, error messages are generated and displayed back to the user by using the <h:messages> elements.

JSF provides its own set of standard converters, which can be registered with input as well as output UI components. You can register a converter with a component in the following three ways:

- Setting converter identifier with the converter attribute of the component's tag, as shown in the following code snippet:

```
<h:inputText id="price" value="#{book.price}" converter="javax.faces.Double"/>
```

- Using the <f:converter> element with converter's identifier inside the component tag, as shown in the following code snippet:

```
<h:inputText id="price" value="#{book.price}">
  <f:converter converterId="javax.faces.Double"/>
</h:inputText>
Or
<h:inputText id="price" value="#{book.price}">
  <f:converter converterId="myConverter"/>
</h:inputText>
```

- Using converter's custom tag inside the component tag, as shown in the following code snippet:

```
<h:inputText value="#{book.price}" >
  <f:convertNumber type="currency" currencySymbol="$"/>
</h:inputText>
```

You have learned about standard JSF converters. You can also create your own converter and use it after registering the converter extension in the faces-config.xml file.

Standard JSF Converters

All the JSF standard converters, which convert a string to a simple Java data type, implement the `javax.faces.convert.Converter` interface. Following are the standard JSF converter classes that form the javax.faces.convert package:

- BigDecimalConverter
- BigIntegerConverter
- booleanConverter
- ByteConverter
- CharacterConverter
- DateTimeConverter
- DoubleConverter
- FloatConverter
- IntegerConverter
- LongConverter
- NumberConverter
- ShortConverter

There are two standard JSF converters, `DateTimeConverter` and `NumberConverter`, and these converters contain custom tags. These two standard converters allow us to configure the format of the component data by setting different tag attributes. These standard tag converter tags are:

- <f:convertDateTime/>
- <f:convertNumber/>

These two tags have been discussed with all their attributes and examples in the *JSF Core Tags* section earlier in this chapter. Let's learn about custom converters next.

Creating Custom Converters

Similar to validators, you can also create custom converters by implementing the `javax.faces.convert.Converter` interface, and defining two methods, `getAsObject()` and `getAsString()`. The following code snippet the structure of a simple custom converter:

```
package com.kogent.converter;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
public class MyConverter implements Converter{
  public Object getAsObject(FacesContext fc, UIComponent comp, String value) {
        //Some Logic
        return null;
  }

  public String getAsString(FacesContext fc, UIComponent comp, Object value) {
        //Some Logic
        return null;
  }

}
```

You need to register the converter that are created in preceding code snippet in the `faces-config.xml` file using the `<converter>` element to use the converter with your component, as shown in the following code snippet:

```
<converter>
  <converter-id>myConverter</converter-id>
  <converter-class>com.kogent.converter.MyConverter</converter-class>
</converter>
```

The value specified in the following code snippet with `<converter-id>` is `myConverter`, which is an identifier for converter created in the preceding code snippet and is used to register this converter with a component, as shown in the following code snippet:

```
<h:inputText id="empid" value="#{employee.empId}" >
  <f:converter converterId="myConverter"/>
</h:inputText>
```

The JSF framework provides support for type conversion using standard and custom converters.

Handling Page Navigation in JSF

One of the important features provided by JSF framework is its support for navigation from one view to another view. JSF allows you to define all possible navigation rules in a centralized location, JSF configuration file. It reduces the need to embed the navigation logic in the components of a JSF page. If the navigation logic is placed at one location, it becomes easy for the developers to maintain the Web application.

The navigation handler plays a vital role in the JSF navigation system. The navigation handler is the code that decides the view to be loaded next and displayed to the user. The default navigation handler operates in response to the action events. Different action resources, such as `HtmlCommandButton` and `HtmlCommandLink`, cause the action events. An action resource may be associated either with a hardcoded outcome string or with an action method, which can return a logical outcome after the execution of an application logic, as shown in the following code snippet:

```
//An HtmlCommandButton associated with hardcode outcome string "home"
<h:commandButton id="home" action="home" value="Home"/>
//An HtmlCommandButton associated with an action method, which returns a logical
outcome.
<h:commandButton id="add" action="#{student.addNew}" value="Add New"/>
```

The navigation handler defines all possible navigation paths. It is the responsibility of navigation handler to take the outcome string and process it on a set of navigation rules. A navigation rule can be defined as the specification that tells which page can be navigated to from a given page or a set of pages according to different outcomes. The outcome can be any string, such as success, failure, error, and login. The following code snippet shows a simple navigation rule defined in the faces-config.xml file:

```
<navigation-rule>
  <from-view-id>/student.jsp</from-view-id>
  <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/display.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

In preceding code snippet the navigation rule defines that if the action event is generated from `student.jsp` page and the outcome string is success, then the next page that needs to be loaded is `display.jsp`. The `<from-view-id>` and `<to-view-id>` tags take the view id as the filename. The following code snippet shows the code to define different navigation cases from one view:

```
<navigation-rule>
  <from-view-id>/login.jsp</from-view-id>
  <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/welcome.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
        <from-outcome>failure</from-outcome>
        <to-view-id>/login_failure.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

In preceding code snippet, the two different navigation cases are declared. In the first navigation case, the request is passed from the login.jsp page to `welcome.jsp` page. However, according to the other navigation case, the request is passed to the `login_failure.jsp` page. The navigation to either welcome.jsp or login_failure.jsp page depends on the outcome string, which can be `success` or `failure`.

You can define different navigation cases not only for a single page but for all pages. You can define navigation rules globally for a Web application, which works for all the JSP and JSF pages. The outcome string of the action method specifies the view page that will be generated. For example, in the following code snippet, if the outcome string is home then the home.jsp page is generated instead of re-generation of the request page. The following code snippet shows the global navigation rule:

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
        <from-outcome>home</from-outcome>
        <to-view-id>/home.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

You can even define a particular action method from where you want to receive the outcome. In case when you have two action methods that return same outcome string and have been associated with action resources on the same view you can define a particular action method from where you want to receive the outcome. You can use the `<from-action>` element to specify the action method while declaring navigation cases, as shown in the following code snippet:

```
<navigation-case>

  <from-action>#{user.login}</from-action>
  <from-outcome>success</from-outcome>
  <to-view-id>/welcome.jsp</to-view-id>

</navigation-case>
```

You have learned about the declarative support in JSF for navigation rules. These rules are defined at a single location; therefore, the development process becomes simpler as well as flexible while implementation and maintenance of the JSF application.

Describing Internationalization Support in JSF

A Web application is accessible from all over the world, so it needs to support localization of messages to display messages according to the language and format preferred by the users.

The UI framework, such as JSF, which supports the creation of rich UI components, supports Internationalization (also known as I18N) and provides different ways to localize the messages of an application without making any change in the code. The most important concept is not to hard code all messages for the user. Instead, the message strings should be fetched from some resource bundle according to the current user Locale. In this section, you learn about all the JSF support provided for Internationalization of an application.

Configuring Supported Locales

The JSF based Web application supports different Locales and you need to configure the Web application for all the supported Locales. You can configure supported Locales for a Web application in the JSF configuration file. The supported Locales can be specified using the `<supported-locale>` and `<locale-config>` elements under the `<application>` element, as shown in the following code snippet:

```
<application>
  <locale-config>
        <default-locale>en</default-locale>
        <supported-locale>en</supported-locale>
        <supported-locale>fr</supported-locale>
  </locale-config>
  <message-bundle>com.kogent.ApplicationMessages</message-bundle>
</application>
```

The Locales supported in the preceding code are `en` (English) and `fr` (French) and the default Locale is en. You can also specify the country for different versions of a language, as shown in the following code snippet:

```
<application>
  <locale-config>
        <default-locale>en</default-locale>
        <supported-locale>en_US</supported-locale>
        <supported-locale>fr_FR</supported-locale>
  </locale-config>
  <message-bundle>com.kogent.ApplicationMessages</message-bundle>
</application>
```

A Web application supports Locales other than the Locale of a Web application's JVM if you configure the supported Locales. You can also declare the base name of the default resource bundle using the <message-bundle> element.

Creating Resource Bundles

Resource bundles are required to make an application to support different Locales. A resource bundle stores Locale-specific text strings, which can be used in a Web application. A resource bundle is a simple properties file, which contains key/value pairs. The properties files have a number of strings placed corresponding to unique keys. You need to create a separate properties file that have the same base name, and the Locale prefixed with the base name for each supported Locale. For example, to provide all text strings to be used for `fr` Locale, you need to create an `ApplicationMessages_fr.properties` file. The structure of `ApplicationMessages_en.properties`, which is the default resource bundle is shown in the following code snippet:

```
welcome=Welcome
roll=Enter Roll No.
name=Enter Name
showdetail=Show Detail
```

A properties file, which contains all text strings translated to support French language is shown in the following code snippet:

```
welcome=Bienvenue
roll=Entrez Roll No.
```

```
name=Entrez Nom
showdetail=Voir Détail
```

In the preceding code snippet, Locales are specified for some keys and the corresponding values are translated into French language. Similarly, you can create resource bundles for other supported Locales also.

Accessing Localized Messages from Resource Bundle

You can load the resource bundle having localized static data for the Locale of current view and use it as a map in the current request. Accessing localized message strings include two basic steps, which are as follows:

- Loading resource bundle in the request scope
- Referencing localized messages

A resource bundle supporting current view Locale can be loaded into the request scope using the `<f:loadBundle>` tag in the JSF page. The key/value pairs stored in the loaded resource bundle are exposed as a map object, which can be accessed using JSF EL. You need to specify the base name of the resource bundle for loading the Locales and the reference name for the map object by setting the basename and var attributes of the `<f:loadBundle>` tag, as shown in the following code snippet:

```
<f:loadBundle basename="com.kogent.ApplicationMessages" var="msg"/>
```

You can use a value-binding expression from an attribute of the component tag that displays the localized data to reference or access a localized message from resource bundle, as shown in the following code snippet:

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
  <head>
        <title>Welcome</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
  </head>       <body>
        <f:view>
        <f:loadBundle basename="com.kogent.ApplicationMessages" var="msg"/>
        <h:messages/>
        <h:outputText value="#{msg.welcome}"/>
        <h:form id="studentForm">
        <h:panelGrid id="panel" columns="2" border="1">
        <h:outputText value="#{msg.roll}" />
        <h:inputText id="rollno" value="#{student.rollno}"/>
        <h:outputText value="#{msg.name}" />
        <h:inputText id="name" value="#{student.name}" />
        <h:commandButton id="button2" action="#{student.showDetail}"
        value="#{msg.showdetail}"/>
        </h:panelGrid> </h:form> </f:view>
  </body>
</html>
```

In the preceding code snippet the Web page is internationalized; therefore supports customized message strings for the current Locale, while loading the resource bundle. You can change the Locale of a view while using the `<f:view>` tag by setting its Locale property. The Locale of a view can also be changed programmatically in the code, as shown in the following code snippet:

```
public String changeToFrench()
{
  FacesContext context = FacesContext.getCurrentInstance();
  context.getViewRoot().setLocale(Locale.FRENCH);
  return null;
}
```

Using the preceding code snippet, you can handle internationalization for all the components created in a Web application by changing the code programmatically.

You can also internationalize all validation and conversion error messages generated by different validators and converters by providing additional key/value pairs in resource bundle. Table 11.50 lists some of the keys for error messages with default text:

Table 11.50: Displaying Keys and their Default Text for Standard JSF Error Messages

Key	Defaut Error Message
javax.faces.validator.NOT_IN_RANGE	Indicates a Validation Error if the given attribute is not in the specified range of {0} and {1}.
javax.faces.validator.NOT_IN_RANGE_ detail	Defines that value should be in between {0} and {1}.
javax.faces.validator.DoubleRangeValidator.LIMIT	Indicates Validation Error: value of specified attribute cannot be converted into proper type.
javax.faces.validator.DoubleRange Validator.MAXIMUM	Indicates Validation Error: value is greater than allowed maximum of {0}.
javax.faces.validator.DoubleRange Validator.MINIMUM	Indicates Validation Error: value is greater than allowed minimum of {0}.
javax.faces.validator.DoubleRange Validator.TYPE	Indicates Validation Error: value is not of the correct type.
javax.faces.validator.LengthValidator. LIMIT	Indicates Validation Error: value of specified attribute cannot be converted into proper type.
javax.faces.validator.LengthValidator. MAXIMUM	Indicates Validation Error: the value has exceeded than allowed maximum of {0}.
javax.faces.validator.LengthValidator. MINIMUM	Indicates Validation Error: the allowed minimum is {0} but the value is more than that.
javax.faces.component.UIInput. CONVERSION	Indicates Conversion Error: value cannot be converted during model data update.
javax.faces.component.UIInput. REQUIRED	Indicates Validation Error: value is required.
javax.faces.component.UISelectOne. INVALID	Indicates Validation Error: value is not valid.
javax.faces.component.UISelectMany. INVALID	Indicates Validation Error: invalid value.
javax.faces.validator.RequiredValidator. FAILED	Indicates Validation Error: value is required.
javax.faces.validator.LongRangeValidator.LIMIT	Indicates Validation Error: value of a specified attribute cannot be converted into proper type.
javax.faces.validator.LongRangeValidator.MAXIM UM	Indicates Validation Error: value is greater than allowed maximum {0} of Long type validator.
javax.faces.validator.LongRangeValidator.MINIM UM	Indicates Validation Error: value is greater than allowed minimum {0} of Long type validator.
javax.faces.validator.LongRangeValidator.TYPE	Indicates Validation Error: Incorrect type.

You can override some of the keys in your resource bundle for different Locales. Using a key you can easily localize the error message produced by standard validators and converters in an application.

Configuring JSF Applications

Configuring a Web application involves addition of different mappings in configuration files. In a JSF application, the basic configuration starts with updating deployment descriptors to support JSF framework. In addition, the JSF application has its own configuration file, which helps in implementing various features of the JSF framework in a JSF application. You need to set the deployment descriptor (web.xml) and JSF configuration file (faces-config.xml) while configuring a JSF application.

Setting web.xml

The JSF framework needs to implement its own standard request processing life cycle for all requests. The request processing is handled by the `javax.faces.webapp.FacesServlet` class, which is a simple servlet and works as an engine for JSF based applications. You need to map JSF requests to `FacesServlet` class so that the requests can be processed through the standard phases of JSF framework . Similar any other Servlet, you need to provide servlet mapping for `FacesServlet` with a specific URL pattern. The following code snippet defines the FacesServlet servlet class and its mapping, which is provided in the web.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="3.0"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>0</load-on-startup>
  </servlet>

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

  <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

In preceding code snippet, all the requests ending with `.faces` are mapped to the `javax.faces.webapp.FacesServlet class`. You can provide any name for a servlet but the use of `javax.faces.webapp.FacesServlet` class is mandatory. It is also known as suffix mapping and the default suffix for resources to load is `.jsp`; therefore, any request ending with `home.faces`, the Servlet searches for `home.jsp` page to handle the request. .

You can also set some JSF Web application parameters, which can be used by the `FacesServlet` servlet class. The JSF Web application parameters are set as context parameters using the `<context-param>` element, as shown in the following code snippet:

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

<context-param>
  <param-name>javax.faces.CONFIG_FILES</param-name>
  <param-value>/WEB-INF/faces-config.xml</param-value>
</context-param>
```

The context parameters, which can be set in the web.xml file to customize the JSF Web application behavior are listed in Table 11.51:

Table 11.51: Showing the JSF Web Application Configuration Parameters

Context Parameter	Description	Default
javax.faces.CONFIG_FILES	Accepts a comma-delimited list of context-relative JSF configuration file, which the framework loads before it loads the WEB-INF/faces-config.xml file.	None
javax.faces.DEFAULT_SUFFIX	Loads resources when extension mapping is used.	.jsp

Table 11.51: Showing the JSF Web Application Configuration Parameters		
Context Parameter	**Description**	**Default**
javax.faces.LIFE-CYCLE_ID	Serves as an identifier for the life cycle of the JSF instance to be used while processing JSF requests within an application.	The default Life-cycle instance
javax.faces.STATE_SAVING_METHOD	Indicates the location where the component states need to be saved (server side of client side).	server

After configuring the Web application deployment descriptor, let's learn about the configuration details provided in JSF configuration file.

Setting the faces-config.xml File

Similar to other Web application frameworks, the JSF configuration file provides essential mapping for the Web application. The configuration details, such as navigation rules, managed beans, and Internationalization settings are provided in the faces-config.xml file. In addition, you can configure customized validators, converters, and components in this file. You have learned to define navigation rules and managed beans in the faces-config.xml file earlier in this chapter.

Let's discuss about the common elements, such as <application>, <managed-bean>, <referenced-bean>, and <navigation-rule> used in the faces-config.xml file.

The `<application>` element is used to declare supported Locales to specify the location of resource bundle, and define the default render kit and other pluggable components. The `<managed-bean>` element implements the managed bean creation facility and helps to create and manage different beans in given scopes. The `<navigation-rule>` element is used to define the navigation rules that should be followed in a Web application. The following code snippet shows an example of using these elements in the `faces-config.xml` file:

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config 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-facesconfig_2_0.xsd"
  version="2.0">
  <application>
        <locale-config>
              <supported-locale>en</supported-locale>
              <supported-locale>fr</supported-locale>
        </locale-config>
        <message-bundle>com.kogent.ApplicationMessages</message-bundle>
  </application>
  <navigation-rule>
        <from-view-id>/login.jsp</from-view-id>
        <navigation-case>
              <from-outcome>success</from-outcome>
              <to-view-id>/welcome.jsp</to-view-id>
        </navigation-case>
        <navigation-case>
              <from-outcome>failure</from-outcome>
              <to-view-id>/login.jsp</to-view-id>
        </navigation-case>
  </navigation-rule>
  <managed-bean>
        <managed-bean-name>beanName</managed-bean-name>
        <managed-bean-class>com.kogent.BeanClassName</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
```

```
    </managed-bean>
</faces-config>
```

Whenever a framework is extended by creating custom UI components, converters, validators, or renderers, the extension should be registered in the JSF configuration file. The elements used in the faces-config.xml file are `<component>`, `<render-kit>`, `<validator>`, and `<converter>`. The basic structure for registering extensions in the faces-config.xml file is shown in the following code snippet:

```
<validator>
    <validator-id>myValidator</validator-id>
    <validator-class>com.kogent.validator.MyValidator</validator-class>
</validator>
<converter>
    <converter-id>myConverter</converter-id>
    <converter-class>com.kogent.converter.MyConverter</converter-class>
</converter>
<component>
    <component-type>SomeType</component-type>
    <component-class>com.kogent.component.MyUIComponent</component-class>
</component>
```

The JSF framework can support multiple configuration files that are defined by setting `javax.faces.CONFIG_FILES` context parameter in the deployment descriptor, web.xml. However, the `faces-config.xml` file does not need to be configured in the `web.xml` file, as this file is by default found in the WEB-INF/faces-config.xml file and is loaded automatically. The `faces-config.xml` file is the location where most of the application configuration details are placed. You need additional configuration files only when you are working on a large application, which is divided into different modules.

You have learned to create components and customize them, manage beans, define navigation rules and handle action events, and apply validations in the Web application with all implementation and configuration details. Now, let's learn to develop a JSF based Web application.

Developing a JSF Application

Let's develop the KogentPro Web application with a number of JSF pages using various JSF UI components, backing beans implementing application logic and a model class handling database interaction for an application. The application created in this section implements all concepts, uses various HTML and Core tags, and other JSF features, such as managed bean, centralized navigation rules, input validation and type conversion support. In addition, the application supports Internationalization through which the localized messages are displayed to a user.

Setting Development Environment

The basic functionality provided by the Web application created in this subsection is maintaining the database of employees, such as adding new employees, editing or deleting details of an existing employee, in an organization. A JSF application is created with the support of following components:

- Set of JSP pages using JSF components.
- A backing bean, `Employee`
- A Model class, `EmployeeDB`
- A custom validator, `EmailValidator`

You cannot declare a faces page as the welcome file; therefore, you have to provide a page to simply create the first JSF request to your home page, which is a JSF page and needs FacesServlet to handle it. Let's create the index.jsp page and provided the `<jsp:forward>` element to access the `home.jsp` page (yet to be created). Listing 11.1 shows the code to create the `index.jsp` page (you can find this file on the CD in the code\JavaEE\Chapter11\KogentPro\folder):

Listing 11.1: Showing the Code for the index.jsp Page

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<html>
```

```
    <body>
        <jsp:forward page="home.jsp"/>
    </body>
</html>
```

Rest of the JSP pages, such as addNew.jsp and viewAll.jsp created in the JSF application use JSF components; therefore, those pages are referred as JSF pages instead of JSP pages.let's discuss these JSP pages one by one.

Creating JSF Pages

A JSF application uses different JSF pages, which further use different JSF UI components to create interactive interfaces with the user with proper formats, localized messages, and required fields. Let's create the following JSP pages in the JSF application:

- home.jsp
- addnew.jsp
- viewall.jsp
- detail.jsp
- edit.jsp

Creating home.jsp page

The home.jsp page is the first page that is displayed to the users. This page contains header and footer content and an image. The most important components used in the home.jsp page are three HtmlCommandLink components, which provide navigational links to other pages. The code of home.jsp page has been provided in Listing 11.2 (you can find this file on the CD in the code\JavaEE\Chapter11\KogentPro\ folder):

Listing 11.2: Showing the Code for the home.jsp Page

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
  <head>
        <title>Welcome - Kogent Solutions Inc</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css"/>
  </head>
  <body>
  <f:view>
        <f:loadBundle basename="com.kogent.ApplicationMessages" var="msg"/>
        <table width="700" align="center" height="500" cellpadding="0" cellspacing="0">
        <tr><td height="70" bgcolor="#5577C6" align="center">
        <h2 style="color: white;"><h:outputText value="#{msg.header}"/></h2>
        </td></tr>
        <tr><td>
        <table>
        <tr>
        <td width="300" valign="middle" align="center">
        <h:form>
        <h:commandLink action="addnew" value="#{msg.addnew}"/><br><br>
        <h:commandLink action="edit" value="#{msg.edit}"/><br><br>
        <h:commandLink action="#{employee.getEmployees}"
        value="#{msg.viewall}"/><br><br>
        </h:form>
        </td>
        <td width="300" align="center">
        <h:graphicImage value="images/people.jpg" width="300" /></td>
        </tr>
        </table>
        </td></tr>
        <tr><td height="40" bgcolor="#C6D1EC" align="center">
        <h:outputText value="#{msg.footer}"/>
        </td></tr>
```

```
            </table>
        </f:view>
        </body>
    </html>
```

In Listing 11.2, the home.jsp page uses localized messages, which are accessed by all `HtmlCommandLink` components created on the home.jsp page. The `<f:loadBundle>` tag is used to load resource bundle and value expressions are set for the value attribute by using the `msg` variable.

First two `HtmlCommandLink` components have associated string outcome and help navigation handler to select the next view. These components are set in the `addnew.jsp` and `edit.jsp pages` through navigation rules in faces-config.xml file, which is discussed later in this chapter. The third `HtmlCommandLink` is associated with an action method `getEmployees()` of the backing bean, `employee`. This backing bean has been discussed later in this chapter. The output of home.jsp page is shown in Figure 11.6:

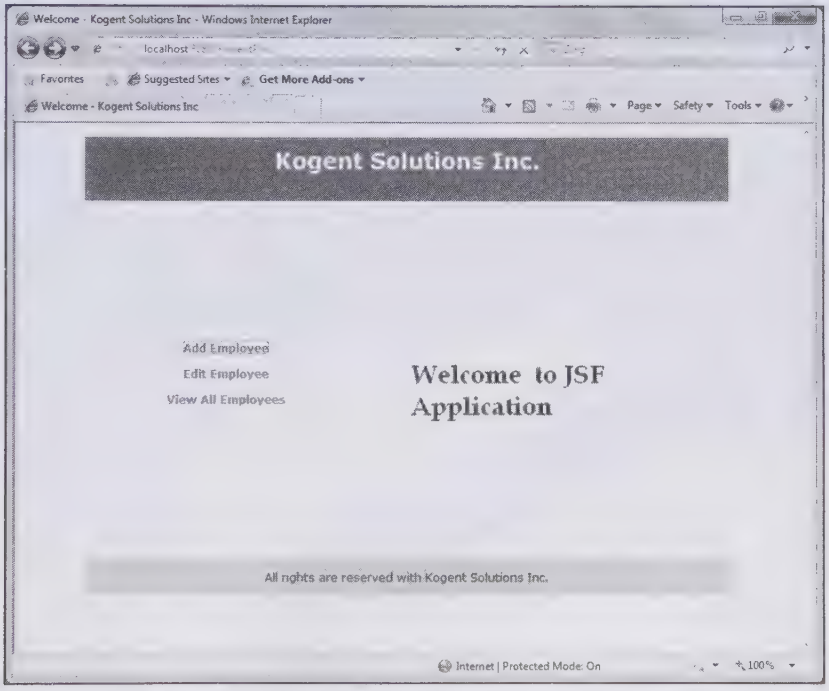


Figure 11.6: Displaying the Output of home.jsp Page

Figure 11.6 shows the the home page of the `KogentPro` Web application and provides links to the `addnew.jsp` and `edit.jsp` pages. Next, let's create the `addnew.jsp` and `edit.jsp` pages.

Creating addnew.jsp page

The addnew.jsp page provides a form with various input fields for giving informations of new employees who need to be added. To get the information about new employees a form and other input components, such as text box, rich text box are created using JSF components. The UI components, such as `HtmlForm`, `HtmlCommandButton`, `HtmlInputText`, `HtmlInputTextarea`, `HtmlSelectOneListbox`, `HtmlSelectOneMenu`, and `HtmlSelectOneRadio are also used in this page`. Let's create a form by using the `HtmlPanelGrid` components, which are used to create a two-column table. Listing 11.3 shows the code for the addnew.jsp file (you can find this file on the CD in the `code\JavaEE\Chapter11\KogentPro\` web folder):

Listing 11.3: Displaying the Code for the `addnew.jsp File`

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
```

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
  <head>
        <title>Adding New Employee...</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
  </head>
  <body>
  <f:view>
        <f:loadBundle basename="com.kogent.ApplicationMessages" var="msg" />
        <table width="700" align="center" height="500" cellpadding="0" cellspacing="0">
        <tr><td height="70" bgcolor="#5577C6" align="center">
        <h2 style="color: white;"><h:outputText value="#{msg.header}" /></h2>
        </td></tr>
        <tr><td align="center" valign="top"><br>
        <h:form>
        <h:panelGrid columns="3" cellspacing="0" cellpadding="5"
        styleClass="menu" border="1">
                <h:commandLink action="home" value="#{msg.home}" />
                <h:commandLink action="edit" value="#{msg.edit}"/>
                <h:commandLink action="#{employee.getEmployees}" value="#{msg.viewall}"/>
                </h:panelGrid>
        </h:form>
        <h5>Enter New Employee Details</h5>
        <h:messages styleClass="error"/>
        <h:form id="addForm">
        <h:panelGrid columns="2" bgcolor="#F1F1F8" width="400" cellpadding="2">
        <h:outputLabel for="empid" value="#{msg.empid}"/>
        <h:inputText id="empid" value="#{employee.id}" required="true">
                <f:validateLongRange minimum="100"/>
        </h:inputText>

        <h:outputLabel for="name" value="#{msg.empname}"/>
        <h:inputText id="name" value="#{employee.name}" required="true"/>
        <h:outputLabel for="address" value="#{msg.address}"/>
        <h:inputTextarea id="address" cols="18" rows="2" value="#{employee.address}"/>

        <h:outputLabel for="city" value="#{msg.city}"/>
        <h:inputText id="city" value="#{employee.city}"/>
        <h:outputLabel for="email" value="#{msg.email}"/>
        <h:inputText id="email" value="#{employee.email}" size="30">
                <f:validator validatorId="emailValidator"/>
        </h:inputText>

        <h:outputLabel for="sex" value="#{msg.sex}"/>
        <h:selectOneRadio id="sex" value="#{employee.sex}">
                <f:selectItem itemValue="Male" itemLabel="Male" />
                <f:selectItem itemValue="Female" itemLabel="Female"/>
        </h:selectOneRadio>

        <h:outputLabel for="department" value="#{msg.department}"/>
        <h:selectOneMenu id="department" value="#{employee.department}" required="true">
        <f:selectItem itemValue="Content Solutions" itemLabel="Centent Solutions"/>
                <f:selectItem itemValue="Testing" itemLabel="Testing"/>
                <f:selectItem itemValue="HR" itemLabel="HR"/>
        </h:selectOneMenu>

        <h:outputLabel for="skills" value="#{msg.skills}"/>
        <h:selectOneListbox id="skills" value="#{employee.skills}">
                <f:selectItem itemValue="Java" itemLabel="Java"/>
                <f:selectItem itemValue="Struts" itemLabel="Struts"/>
                <f:selectItem itemValue="Spring" itemLabel="Spring"/>
        </h:selectOneListbox>
        <h:outputLabel for="joindate" value="#{msg.joindate}"/>
        <h:panelGroup>
        <h:inputText id="joindate" value="#{employee.joinDate}" required="true">
```

```
                    <f:convertDateTime type="date" pattern="dd-MM-yyyy"/>
                </h:inputText>
                <h:outputText value=" [dd-MM-yyyy]"/>
            </h:panelGroup>
            <h:commandButton action="#{employee.addNew}" value="#{msg.addnew}"/>
            <h:commandButton type="reset" value="#{msg.reset}" />
            </h:panelGrid>
            </h:form><br>
            </td></tr>
            <tr><td height="40" bgcolor="#C6D1EC" align="center">
                <h:outputText value="#{msg.footer}" />
            </td></tr>
            </table>
        </f:view>
        </body>
    </html>
```

In Listing 11.3, a `LongRangeValidator` is used with input field, empid and a custom validator, `EmailValidator` to validate the input field, `email`. The output of the `addnew.jsp` file is shown in Figure 11.7:

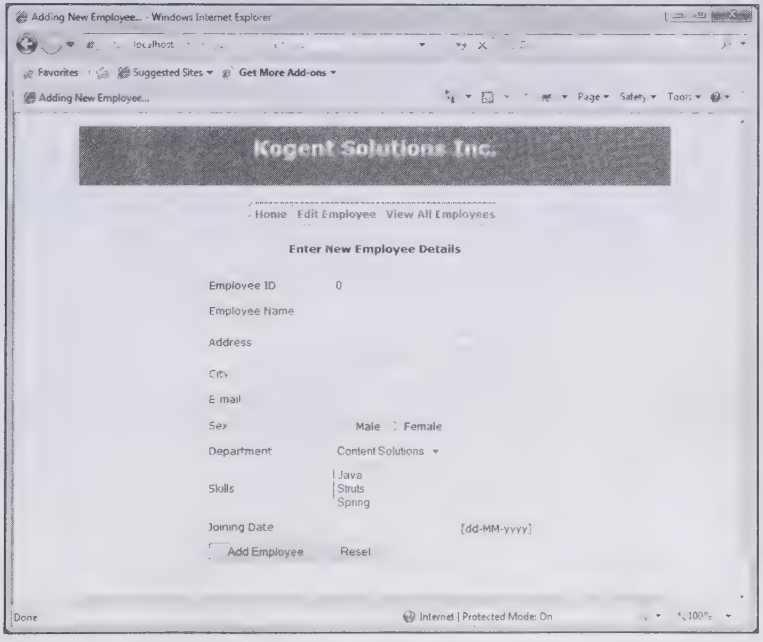


Figure 11.7: Displaying the Output of addnew JSF Page

All the input fields created in the KogentPro application have been associated with different properties of employee backing bean. Listing 11.3. An HTMLCommandButton component is created that binds the request with the action method, addNew() and this method is created in the same backing bean. The `addNew()` method processes the logic of adding a new employee record in the database. The `DateTimeConverter` with `HtmlInputText` component is used for joining date to ensure that the date is entered in the correct format. The `HtmlMessages` component is used to display all components and application level messages.

NOTE

During the execution of the KogentPro application, all JSP pages (.jsp files) get converted into the .faces pages, as configured in the web.xml file of the application.

Creating the viewall.jsp Page

The viewall.jsp page shows the list of all existing employees. This page uses an `HtmlDataTable` component, which iterates over an `ArrayList` in the application scope. The `<h:dataTable>` and `<h:column>` elements are also used in this page. In addition, two `HtmlCommandLink` components (details and delete hyperlinks) are used in each row for getting the details of an employee and deleting an employee from database using the `HtmlDataTable` component. Listing 11.4 shows the the code of the viewall.jsp page (you can find this file on the CD in the code\JavaEE\Chapter11\KogentPro\ web folder):

Listing 11.4: Showing the Code for the viewall.jsp Page

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
  <head>
        <title>Listing all Employees</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
  </head>
  <body>
  <f:view>
        <f:loadBundle basename="com.kogent.ApplicationMessages" var="msg" />
        <table width="700" align="center" height="500" cellpadding="0" cellspacing="0">
        <tr><td height="70" bgcolor="#5577C6" align="center">
        <h2 style="color: white;"><h:outputText value="#{msg.header}"/></h2>
        </td>
        </tr>
        <tr>
        <td align="center" valign="top"><br>
        <h:form>
        <h:panelGrid columns="3" cellspacing="0" cellpadding="5"
        styleClass="menu" border="1">
                <h:commandLink action="home" value="#{msg.home}" />
                <h:commandLink action="addnew" value="#{msg.addnew}" />
                <h:commandLink action="edit" value="#{msg.edit}" />
                </h:panelGrid>
        </h:form>
        <h5 align="left">Listing all employees</h5>
        <h:form>
        <h:dataTable value="#{applicationScope.employees}" var="emp" cellspacing="1"
        cellpadding="4" width="600" headerClass="header" rowClasses="odd-row, even-row">
                <h:column>
                        <f:facet name="header">
                                <h:outputText value="ID" />
                        </f:facet>
                        <h:outputText value="#{emp.id}" />
                </h:column>
                <h:column>
                        <f:facet name="header">
                                <h:outputText value="Employee Name" />
                        </f:facet>
                        <h:outputText value="#{emp.name}" />
                </h:column>
                <h:column>
                        <f:facet name="header">
                                <h:outputText value="Department" />
                        </f:facet>
                        <h:outputText value="#{emp.department}" />
                </h:column>
                <h:column>
                        <f:facet name="header">
                                <h:outputText value="Date of Joining" />
                        </f:facet>
                        <h:outputText value="#{emp.joinDate}">
                                <f:convertDateTime pattern="dd MMM, yyyy" />
                        </h:outputText>
```

```
                </h:column>
                <h:column>
                        <f:facet name="header">
                                <h:outputText value="- -" />
                        </f:facet>
                        <h:commandLink value="#{msg.details}"
                        action="#{employee.getDetail}">
                        <f:setPropertyActionListener
                        target="#{employee.id}" value="#{emp.id}" />
                        </h:commandLink>
                </h:column>
                <h:column>
                        <f:facet name="header">
                                <h:outputText value="- -" />
                        </f:facet>
                        <h:commandLink action="#{employee.deleteEmployee}"
                        value="#{msg.delete}" onclick="return confirm('Do you want to
                        delete.')">
                        <f:setPropertyActionListener target="#{employee.id}"
                        value="#{emp.id}" />
                        </h:commandLink>
                </h:column>
                </h:dataTable>
        </h:form>
        </td></tr>
        <tr><td height="40" bgcolor="#C6D1EC" align="center">
                <h:outputText value="#{msg.footer}" />
        </td></tr>
        </table>
    </f:view>
    </body>
</html>
```

In Listing 11.4, two `HtmlCommandLink` components are created for each row and associated with two different action methods, `getDetail()` and `deleteEmployee()`. We have used the `<f:setPropertyActionListener>` element with the `HtmlCommandLink` components to set the employee id before the action method is executed. The output of the `viewall.jsp` page displays the existing three employees in the database, as shown in Figure 11.8:

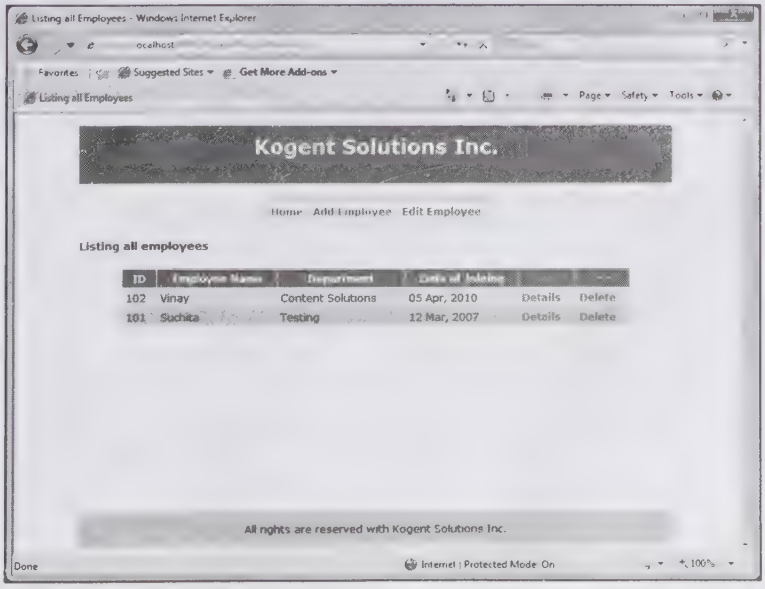


Figure 11.8: Showing the Output of the viewall JSF Page

In Figure 11.8, header and alternative rows are provided in different styles by setting the `headerClass` and `rowClasses` attributes of the `HtmlDataTable` component.

Creating the detail.jsp page

The `detail.jsp` page shows the detail of a particular employee. This page can be accessed from the `viewall.jsp` page, which has the `details` hyperlink (`HtmlCommandLink` component) to view the details corresponding to each employee. The `detail.jsp` page uses the `HtmlOutputText` element associated with individual properties of the `employee` backing bean. Listing 11.5 shows the code for the detail.jsp page (you can find this file on the CD in the code\JavaEE\Chapter11\KogentPro\ folder):

Listing 11.5: Showing the Code for the detail.jsp Page

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
  <head>
        <title>Employee Detail...</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
  </head>
  <body>
  <f:view>
        <f:loadBundle basename="com.kogent.ApplicationMessages" var="msg"/>
        <table width="700" align="center" height="500" cellpadding="0" cellspacing="0">
        <tr><td height="70" bgcolor="#5577C6" align="center">
        <h2 style="color: white;"><h:outputText value="#{msg.header}"/></h2>
        </td></tr>
        <tr><td align="center" valign="top"><br>
        <h:form>
        <h:panelGrid columns="3" cellspacing="0" cellpadding="5" styleClass="menu"
        border="1">
                <h:commandLink action="home" value="#{msg.home}"/>
                <h:commandLink action="addnew" value="#{msg.addnew}"/>
                <h:commandLink action="#{employee.getEmployees}" value="#{msg.viewall}"/>
                        </h:panelGrid>
        </h:form>

        <br><h5>Employee Details</h5>
        <h:form id="empForm">
        <h:panelGrid columns="2" cellpadding="4" cellspacing="4" border="1" rules="rows">
                <h:outputText value="#{msg.empid}"/>
                <h:outputText id="empid" value="#{employee.id}"/>
                <h:outputText value="#{msg.empname}"/>
                <h:outputText id="name" value="#{employee.name}" />
                <h:outputText value="#{msg.address}"/>
                <h:outputText id="address" value="#{employee.address}"/>
                <h:outputText value="#{msg.city}"/>
                <h:outputText id="city" value="#{employee.city}"/>
                <h:outputText value="#{msg.email}"/>
                <h:outputText id="email" value="#{employee.email}" />
                <h:outputText value="#{msg.sex}"/>
                <h:outputText id="sex" value="#{employee.sex}"/>
                <h:outputText value="#{msg.department}"/>
                <h:outputText id="department" value="#{employee.department}"/>
                <h:outputText value="#{msg.skills}"/>
                <h:outputText id="skills" value="#{employee.skills}"/>
                <h:outputText value="#{msg.joindate}"/>
                <h:outputText id="joindate" value="#{employee.joinDate}">
                <f:convertDateTime type="date" pattern="dd-MM-yyyy"/>
                </h:outputText>
                </h:panelGrid>
        </h:form>
        </td></tr>
        <tr><td height="40" bgcolor="#C6D1EC" align="center">
                <h:outputText value="#{msg.footer}"/>
```

```
            </td></tr>
            </table>
        </f:view>
        </body>
    </html>
```

The output of the detail.jsp page is shown in Figure 11.9:

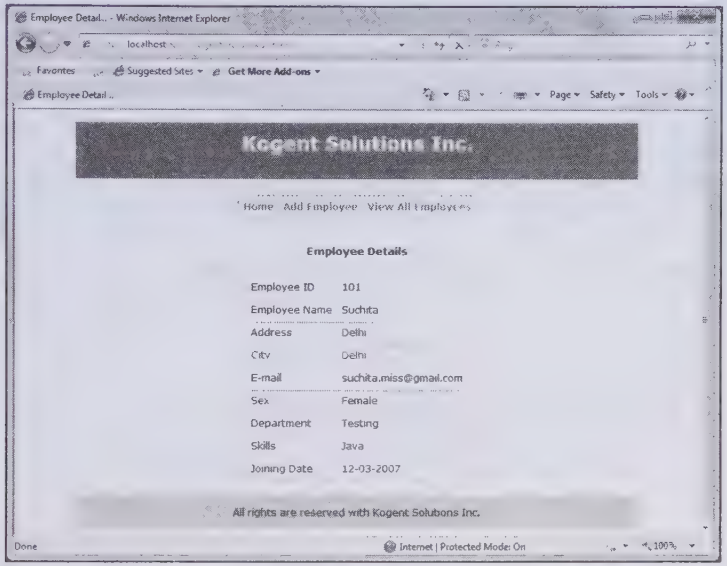


Figure 11.9: Displaying the Output of the detail JSF Page

In Figure 11.9 the detail.jsp page displays the properties of employee backing bean, which is further updated for the selected employee by the `getDetail()` method of the `employee` backing bean.

Creating the edit.jsp Page

The `edit.jsp` page creates two different HtmlForm components, `getEmp` and `editForm`. The `editForm` HtmlForm component does not render initially as its render attribute is set to `false`. The `getEmp` HtmlForm component is submitted and handled by the `getEmployee()` action method. The `getEmployee()` method processes the business logic and updates the details of the employee with respect to the entered `employee id`. Listing 11.6 shows the code of the `edit.jsp` page (you can find this file on the CD in the code\JavaEE\Chapter11\KogentPro\ folder):

Listing 11.6: Showing the Code for the edit.jsp Page

```
<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
  <head>
        <title>Editing Employee...</title>
        <link rel="stylesheet" href="mystyle.css" type="text/css" />
  </head>
  <body>
  <f:view>
        <f:loadBundle basename="com.kogent.ApplicationMessages" var="msg" />
<table width="700" align="center" height="500" cellpadding="0" cellspacing="0">
        <tr><td height="70" bgcolor="#5577C6" align="center">
<h2 style="color: white;"><h:outputText value="#{msg.header}" /></h2>
        </td></tr>
        <tr><td align="center" valign="top"><br>
        <h:form>
<h:panelGrid columns="3" cellspacing="0" cellpadding="5" styleClass="menu" border="1">
                <h:commandLink action="home" value="#{msg.home}" />
```

```
            <h:commandLink action="addnew" value="#{msg.addnew}"/>
    <h:commandLink action="#{employee.getEmployees}" value="#{msg.viewall}"/>
            </h:panelGrid>
    </h:form>
    <h:messages styleClass="error"/>

    <h:form id="getEmp">
            <h:outputLabel value="#{msg.enterid}" for="intputId"/> 
            <h:inputText id="empid" value="#{employee.id}"/>
    <h:commandButton value="Enter" actionListener="#{employee.getEmployee}"/>
    </h:form>

    <h:form id="editForm" rendered="false" binding="#{employee.editForm}" >
    <h:inputHidden value="#{employee.id}"/>
    <h:panelGrid columns="2" bgcolor="#F1F1F8" width="400" cellpadding="2">
    <h:outputLabel for="empid" value="#{msg.empid}"/>
    <h:inputText id="empid" value="#{employee.id}" readonly="true"/>

    <h:outputLabel for="name" value="#{msg.empname}"/>
    <h:inputText id="name" value="#{employee.name}" required="true"/>

    <h:outputLabel for="address" value="#{msg.address}"/>
    <h:inputTextarea id="address" cols="18" rows="2" value="#{employee.address}"/>

    <h:outputLabel for="city" value="#{msg.city}"/>
    <h:inputText id="city" value="#{employee.city}"/>

    <h:outputLabel for="email" value="#{msg.email}"/>
    <h:inputText id="email" value="#{employee.email}" size="30">
            <f:validator validatorId="emailvalidator"/>
    </h:inputText>

    <h:outputLabel for="sex" value="#{msg.sex}"/>
    <h:selectOneRadio id="sex" value="#{employee.sex}">
            <f:selectItem itemValue="Male" itemLabel="Male" />
            <f:selectItem itemValue="Female" itemLabel="Female"/>
    </h:selectOneRadio>

    <h:outputLabel for="department" value="#{msg.department}"/>
    <h:selectOneMenu id="department" value="#{employee.department}" required="true">
    <f:selectItem itemValue="Content Solutions" itemLabel="Centent Solutions"/>
            <f:selectItem itemValue="Testing" itemLabel="Testing"/>
            <f:selectItem itemValue="HR" itemLabel="HR"/>
    </h:selectOneMenu>

    <h:outputLabel for="skills" value="#{msg.skills}"/>
    <h:selectOneListbox id="skills" value="#{employee.skills}">
            <f:selectItem itemValue="Java" itemLabel="Java"/>
            <f:selectItem itemValue="Struts" itemLabel="Struts"/>
            <f:selectItem itemValue="Spring" itemLabel="Spring"/>
    </h:selectOneListbox>

    <h:outputLabel for="joindate" value="#{msg.joindate}"/>
    <h:panelGroup>
    <h:inputText id="joindate" value="#{employee.joinDate}" required="true">
            <f:convertDateTime type="date" pattern="dd-MM-yyyy"/>
            </h:inputText>
            <h:outputText value=" [dd-MM-yyyy]"/>
    </h:panelGroup>
    <h:commandButton action="#{employee.update}" value="#{msg.update}"/>
    </h:panelGrid>
    </h:form><br>
    </td></tr>
    <tr><td height="40" bgcolor="#C6D1EC" align="center">
            <h:outputText value="#{msg.footer}" />
```

```
            </td></tr>
            </table>
    </f:view>
    </body>
</html>
```

In Listing 11.6, the `HtmlForm` component, `editForm` is similar to the `addnew.jsp` page except the fact that the `HtmlInputText` component for employee id is set as readonly and an `HtmlInputHidden` component has been placed. The `HtmlInputHidden` component prohibits the user from changing the employee id, as there is no need to change employee id while modifying the details of an existing employee. The `update()` method of the backing bean is responsible for updating the employee record with the modified detail and this method is bound as action method.

You have learned to create various JSF pages that use a backing bean, Employee. The UI components in these JSF pages are used to either read or set the values of properties of the Employee backing bean. Different action methods of the Employee bean are used to perform these actions. Now, let's learn to create a backing bean for request processing.

Creating the Employee Backing Bean

A single backing bean, employee is created in the `KogentPro` application. This backing bean has a set of properties and their corresponding get and set methods. Different action methods, such as `addNew()`, `update()`, `deleteEmployee()`, `getDetail()`, and `getEmployees()` are defined in the bean. In addition, we have an action listener method, `getEmployee()`. All these action methods have been bound with different command components on various pages to process some logic. Listing 11.7 shows the code for the backing bean class, com.kogent.Employee (you can find this file on the CD in the `code\JavaEE\Chapter11\KogentPro\src\com\kogent\` folder):

Listing 11.7: Displaying the Code of Employee.java File

```
package com.kogent;

import java.util.ArrayList;
import java.util.Date;
import java.util.Map;

import javax.faces.application.FacesMessage;
import javax.faces.component.html.HtmlForm;
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.faces.event.ActionEvent;

public class Employee {
  int id;
  String name;
  String address;
  String city;
  String email;
  String sex;
  String department;
  String skills;
  Date joinDate;

  ArrayList<Employee> employees;

  HtmlForm editForm;

  public String getAddress() {
        return address;
  }
  public void setAddress(String address) {
```

```
        this.address = address;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getDepartment() {
        return department;
    }
    public void setDepartment(String department) {
        this.department = department;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }


    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public Date getJoinDate() {
        return joinDate;
    }
    public void setJoinDate(Date joinDate) {
        this.joinDate = joinDate;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getSex() {
        return sex;
    }
    public void setSex(String sex) {
        this.sex = sex;
    }
    public String getSkills() {
        return skills;
    }
    public void setSkills(String skills) {
        this.skills = skills;
    }

    public HtmlForm getEditForm() {
        return editForm;
    }
    public void setEditForm(HtmlForm editForm) {
        this.editForm = editForm;
    }
```

```
    public String addNew(){
            FacesContext fc=FacesContext.getCurrentInstance();
            EmployeeDB empdb=new EmployeeDB();

            if(!empdb.checkEmployee(id)){
                    empdb.addEmployee(this);
            }
            else{
            fc.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR, "Employee with
            this ID exists.", ""));
                    return "failure";
            }
            Map<String, Object>
application=FacesContext.getCurrentInstance().getExternalContext().getApplicationMap();
            application.put("employees", empdb.getEmployees());

            return "success";
    }
    public  String getEmployees(){
            EmployeeDB empdb=new EmployeeDB();
            FacesContext fc=FacesContext.getCurrentInstance();
            ExternalContext exc=fc.getExternalContext();
            Map<String, Object> application=exc.getApplicationMap();
            application.put("employees", empdb.getEmployees());
            return "viewall";
    }

    public void getEmployee(ActionEvent event){

            FacesContext fc=FacesContext.getCurrentInstance();
            ExternalContext exc=fc.getExternalContext();
            Map<String, Object> request=exc.getRequestMap();

            EmployeeDB empdb=new EmployeeDB();
            Employee employee=empdb.getEmployee(id);
            if(employee==null){
            fc.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR, "Employee with
            this ID does not exist.", ""));
                    editForm.setRendered(false);
            }
            else{
                    request.put("employee", employee);
                    editForm.setRendered(true);
            }
    }

    public String update(){
            FacesContext fc=FacesContext.getCurrentInstance();
            EmployeeDB empdb=new EmployeeDB();
            int i=empdb.updateEmployee(this);
            if(i==1){
            Map<String, Object> application=fc.getExternalContext().getApplicationMap();
                    application.put("employees", empdb.getEmployees());
                    return "success";
            }
            else{
            fc.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR, "Employee could
            not be Updated.", ""));
                    return "failure";
            }
    }

    public String getDetail(){
            EmployeeDB empdb=new EmployeeDB();
            Employee employee=empdb.getEmployee(this.id);
```

```
Map<String, Object>
   request=FacesContext.getCurrentInstance().getExternalContext().getRequestMap();
         request.put("employee", employee);
         return "detail";
   }
   public String deleteEmployee(){

         EmployeeDB empdb=new EmployeeDB();
         int i=empdb.deleteEmployee(this.id);
         System.out.print(i+"row deleted");
Map<String, Object>
   application=FacesContext.getCurrentInstance().getExternalContext().getApplicationMap();
         application.put("employees", empdb.getEmployees());
         return "viewall";
   }

}
```

In Listing 11.7, the `com.kogent.EmployeeDB` class is used in the `Employee` class. The `EmployeeDB` class is a model class that interacts with database and the backing bean invokes methods on its object to perform various operations, such as adding a record, deleting and modifying a record, and fetching employee records from the `EMPLOYEE` table Let's discuss various action methods of the employee backing bean.

String getEmployees()

The getEmployees() method simply returns an `ArrayList` of Employee objects using the `getEmployees()` method on the `EmployeeDB` object and places this `ArrayList` object in the application scope by the name, employees. This action method returns viewall as the outcome string. You can see the `faces-config.xml` file to configure the mapping for this outcome.

String addNew()

The addNew() method checks whether the employee with the given id already exists or not and adds the new record by calling the `addEmployee()` method on the `EmployeeDB` object. Further, it updates an application scope attribute named employee with a new set of employees. In case, the employee with the given id exists, the FacesMessage object is created and added into the `FacesContext` instance.

String update()

The update() method invokes the `updateEmployee()` method on the `EmployeeDB` object and returns success or failure accordingly. If the record is updated successfully, the application scoped `ArrayList` object, employee is also updated again.

String deleteEmployee()

The `deleteEmployee()` method is invoked to delete the employee with the given employee id. The `deleteEmployee()` method of the Employee class uses the `deleteEmployee()` method of the `EmployeeDB` class. This method also returns `viewall` as outcome string.

String getDetail()

The getDetail() method fetches an Employee object representing a given employee and uses the `getEmployee()` method of the `EmployeeDB` class. After fetching the `Employee` object, this object is placed in the request scope so that it is accessible from the next view to be loaded. Returns `detail` as outcome string.

void getEmployee(ActionEvent)

The getEmployee(ActionEvent) is the only action listener method among other action methods and does not affect navigation. It performs logic and receives an `Employee` object by calling the `getEmployee(int)` method on the `EmployeeDB` object. If the fetched `Employee` object is not null, it simply places the object in the request scope and sets the render property of the `empForm` object to true. After setting the render property to true, the

empForm is visible to the user; thereby, allows the user to edit the employee details through the `edit.jsp` page.

Managing Employee Bean

The `com.kogent.Employee` class is a backing bean and needs to be managed in the `faces-config.xml` file. You need to provide a three-line configuration detail to define the identifier for this backing bean, set the class name and the scope of the bean. The configuration details of managed beans is shown in the following code snippet:

```
<managed-bean>
  <managed-bean-name>employee</managed-bean-name>
  <managed-bean-class>com.kogent.Employee</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

In the preceding code snippet, the employee managed bean is configured by defining its class and scope. Let's now create the `EmployeeDB` class.

Creating the EmployeeDB Class

As we have seen, backing bean has performed varied database operations without executing any JDBC code. The database interaction code has been bundled into a model class that has been used in all action methods of the `Employee` class. Table 11.52 shows the structure of employee table used in the `KogentPro` application:

Table 11.52: Showing the Structure of Employee Table

Name	Type
ID	NUMBER(38)
NAME	VARCHAR2(35)
ADDRESS	VARCHAR2(50)
CITY	VARCHAR2(15)
EMAIL	VARCHAR2(30)
SEX	VARCHAR2(6)
DEPARTMENT	VARCHAR2(25)
SKILLS	VARCHAR2(40)
JOINDATE	DATE

The model class, `com.kogent.EmployeeDB` has various methods created for different database operations. Listing 11.8 shows the code of the EmployeeDB class file (you can find this file on the CD in the `code\JavaEE\Chapter11\KogentPro\src\com\kogent\` folder):

Listing 11.8: Showing the Code of EmployeeDB.java File

```
package com.kogent;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;

public class EmployeeDB {

  Connection con;
  ResultSet rs;
  Statement stmt;
```

```
public EmployeeDB() {
        try {
                Class.forName("oracle.jdbc.driver.OracleDriver");
con = DriverManager.getConnection
        ("jdbc:oracle:thin:@192.168.1.123:1521:XE","scott","tiger");
        } catch (Exception e) {
                e.printStackTrace();
        }
}

public boolean  checkEmployee(int empid){
        boolean  found=false;
        try {
                stmt=con.createStatement();
                rs=stmt.executeQuery("select * from employee where id="+empid);
                if(rs.next())
                        found=true;
        } catch (SQLException e) {
                e.printStackTrace();
        }
        return found;
}
public ArrayList<Employee> getEmployees(){
        ArrayList<Employee> employees=new ArrayList<Employee>();
        Employee emp=null;

        try {
                stmt=con.createStatement();
                rs=stmt.executeQuery("select * from employee ");
                while(rs.next()){
                        emp=new Employee();
                        emp.setId(rs.getInt(1));
                        emp.setName(rs.getString(2));
                        emp.setAddress(rs.getString(3));
                        emp.setCity(rs.getString(4));
                        emp.setEmail(rs.getString(5));
                        emp.setSex(rs.getString(6));
                        emp.setDepartment(rs.getString(7));
                        emp.setSkills(rs.getString(8));
                        emp.setJoinDate(rs.getDate(9));
                        employees.add(emp);
                }
                rs.close();
        } catch (Exception e) {
                e.printStackTrace();
        }
        return employees;
}

public Employee getEmployee(int empid){
        Employee emp=null;

        try {
                stmt=con.createStatement();
                rs=stmt.executeQuery("select * from employee where id="+empid);
                if(rs.next()){
                        emp=new Employee();
                        emp.setId(rs.getInt(1));
                        emp.setName(rs.getString(2));
                        emp.setAddress(rs.getString(3));
                        emp.setCity(rs.getString(4));
                        emp.setEmail(rs.getString(5));
                        emp.setSex(rs.getString(6));
                        emp.setDepartment(rs.getString(7));
                        emp.setSkills(rs.getString(8));
                        emp.setJoinDate(rs.getDate(9));
```

```
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return emp;
    }

    public void addEmployee(Employee emp){
        try {
        PreparedStatement pstmt=con.prepareStatement("insert into employee
        values(?,?,?,?,?,?,?,?,?)");
                pstmt.setInt(1,emp.getId());
                pstmt.setString(2, emp.getName());
                pstmt.setString(3, emp.getAddress());
                pstmt.setString(4, emp.getCity());
                pstmt.setString(5, emp.getEmail());
                pstmt.setString(6, emp.getSex());
                pstmt.setString(7, emp.getDepartment());
                pstmt.setString(8, emp.getSkills());
                java.sql.Date date=new java.sql.Date(emp.getJoinDate().getTime());
                pstmt.setDate(9, date);
                pstmt.executeUpdate();
                pstmt.close();

        } catch (SQLException e) {
                e.printStackTrace();
        }

    }

    public int updateEmployee(Employee employee){
        int i=0;
        try {
                stmt=con.createStatement();
                String query="update employee set name='"+employee.getName()+"', ";
                query+="address='"+employee.getAddress()+"', ";
                query+="city='"+employee.getCity()+"', ";
                query+="email='"+employee.getEmail()+"', ";
                query+="sex='"+employee.getSex()+"', ";
                query+="department='"+employee.getDepartment()+"', ";
                query+="skills='"+employee.getSkills()+"', ";
                query+="joindate=to_date('"+new
                java.sql.Date(employee.getJoinDate().getTime())+"', 'yyyy-MM-dd') ";
                query+="where id="+employee.getId();
                System.out.println(query);
                i=stmt.executeUpdate(query);
        } catch (Exception e) {
                e.printStackTrace();
        }
        return i;
    }

    public int deleteEmployee(int id){
        int i=0;
        try {
                stmt=con.createStatement();
                System.out.println("delete from employee where id="+id);
                i=stmt.executeUpdate("delete from employee where id="+id);
        } catch (Exception e) {
                e.printStackTrace();
        }
        return i;
    }
}
```

In Listing 11.8, the `EmployeeDB` class is all about handling JDBC statements. We have created various methods to perform different database operations. The methods of the EmployeeDB class are as follows:

- boolean checkEmployee(int empid)
- ArrayList<Employee> getEmployees()
- Employee getEmployee(int empid)
- void addEmployee(Employee emp)
- int updateEmployee(Employee employee)
- int deleteEmployee(int id)

The names of these methods are self-explanatory for the functions they perform. Their return types and the argument types further describe their usage in the backing bean.

Creating the **EmailValidator** Class

We have used a custom validator to validate the email id of an employee. The custom validator is used with the `HtmlInputText` component in the `addnew.jsp` and `edit.jsp` pages, as shown in the following code snippet:

```
<h:inputText id="email" value="#{employee.email}" size="30">
  <f:validator validatorId="emailValidator"/>
</h:inputText>
```

In the preceding code snippet, we have created a custom validator by implementing the `javax.faces.validator.Validator` interface and defining its `validate()` method. The logic implemented to validate a string for a valid email id is shown in Listing 11.9 (you can find this file on the CD in the `code\JavaEE\Chapter11\KogentPro\src\com\kogent\validator\` folder):

Listing 11.9: Showing the Code of the EmailValidator.java File

```
package com.kogent.validator;

import java.util.regex.Pattern;
import java.util.regex.Matcher;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.html.HtmlInputText;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class EmailValidator implements Validator {

public void validate(FacesContext facesContext, UIComponent uiComponent, Object value)
  throws ValidatorException {

        String patternStr = ".+@.+\\.[a-z]+"; // For Email
        Pattern pattern = Pattern.compile(patternStr);

        // Passed Value must be type cast into CharSequence
        Matcher matcher = pattern.matcher((CharSequence)value);
        boolean matchFound = matcher.matches();
        if (!matchFound){
                HtmlInputText htmlInputText = (HtmlInputText) uiComponent;
                FacesMessage facesMessage = new FacesMessage(htmlInputText.getLabel()+ ":
                Email Format is not valid");
                throw new ValidatorException(facesMessage);
        }
  }
}
```

After compiling the validator class and placing it in the application's classpath, the custom validator should be registered in the `faces-config.xml` file. The registration of custom validator is required to define its

identifier, which is used by the input components. The `com.kogent.validator.EmailValidator` class has been registered in the `faces-config.xml` file, as shown in the following code snippet:

```
<validator>
    <validator-id>emailValidator</validator-id>
    <validator-class>com.kogent.validator.EmailValidator</validator-class>
</validator>
```

Now, let's learn about configuring a JSF application.

Configuring a JSF Application

Configuring a JSF application involves setting the `web.xml` and `faces-config.xml` files. The setting provided in deployment descriptor includes configuring `FacesServlet`, whereas configuring the `faces-config.xml` file includes navigation rules to be followed by the navigation handler, managing backing beans, and registering custom validator used in a Web application.

Enabling JSF Servlet in the web.xml File

A JSF-based application needs all JSF requests to be handled by the `FacesServlet` class. Listing 11.10 shows the code to define `FacesServlet` and provide a servlet mapping for `FacesServlet` in the deployment descriptor (you can find this file on the CD in the code\JavaEE\Chapter11\KogentPro\WEB-INF\folder):

Listing 11.10: Showing the Code of the web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns=http://java.sun.com/xml/ns/javaee
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="23.0"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

  <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>0</load-on-startup>
  </servlet>

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

  <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

In Listing 11.10, we have used a `*.faces` as the url pattern, which tells that all requests ending with `.faces` will be mapped to `FacesServlet` and handled by the JSF engine. If you need to access the `edit.jsp` page, which is a JSF page, you can access it by requesting the `edit.faces` page.The `index.jsp` page is a welcome file that is forwarded to the `home.jsp` page that requests a JSF Servlet for `home.faces` page..

Navigation Rules Defined in the faces-config.xml File

All the navigation rules for the application are defined in the `faces-config.xml` file. Different action methods are used to return string as an outcome and navigation handler to search for navigation cases defined for matching outcome string to load the next view for the user. All the navigation rules defined in the `faces-config.xml` file are shown in Listing 11.11 (you can find this file on the CD in the code\JavaEE\Chapter11\KogentPro\WEB-INF\ folder):

Listing 11.11: Showing the Navigation Rules Provided in faces-config.xml File

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config 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-facesconfig_2_0.xsd"
```

```
version="2.0">

<navigation-rule>
      <from-view-id>*</from-view-id>
      <navigation-case>
            <from-outcome>home</from-outcome>
            <to-view-id>/home.jsp</to-view-id>
      </navigation-case>
</navigation-rule>

<navigation-rule>
      <from-view-id>*</from-view-id>
      <navigation-case>
            <from-outcome>addnew</from-outcome>
            <to-view-id>/addnew.jsp</to-view-id>
      </navigation-case>
</navigation-rule>
<navigation-rule>
      <from-view-id>*</from-view-id>
      <navigation-case>
            <from-outcome>viewall</from-outcome>
            <to-view-id>/viewall.jsp</to-view-id>
      </navigation-case>
</navigation-rule>
<navigation-rule>
      <from-view-id>*</from-view-id>
      <navigation-case>
            <from-outcome>edit</from-outcome>
            <to-view-id>/edit.jsp</to-view-id>
      </navigation-case>
</navigation-rule>

<navigation-rule>
      <from-view-id>/addnew.jsp</from-view-id>
      <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/viewall.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>failure</from-outcome>
            <to-view-id>/addnew.jsp</to-view-id>
      </navigation-case>
</navigation-rule>

<navigation-rule>
      <from-view-id>/edit.jsp</from-view-id>
      <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/viewall.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>failure</from-outcome>
            <to-view-id>/edit.jsp</to-view-id>
      </navigation-case>
</navigation-rule>

<navigation-rule>
      <from-view-id>/viewall.jsp</from-view-id>
      <navigation-case>
            <from-outcome>detail</from-outcome>
            <to-view-id>/detail.jsp</to-view-id>
```

```
        </navigation-case>
    </navigation-rule>

</faces-config>
```

In Listing 11.11, we have defined different global navigation rules for outcome strings, `home`, `addnew`, `viewall`, and `edit`. The navigation handler loads, `home.jsp`, `addnew.jsp`, `viewall.jsp`, and `edit.jsp` respectively with outcome strings. All navigation rules and cases are shown in Figure 11.10:

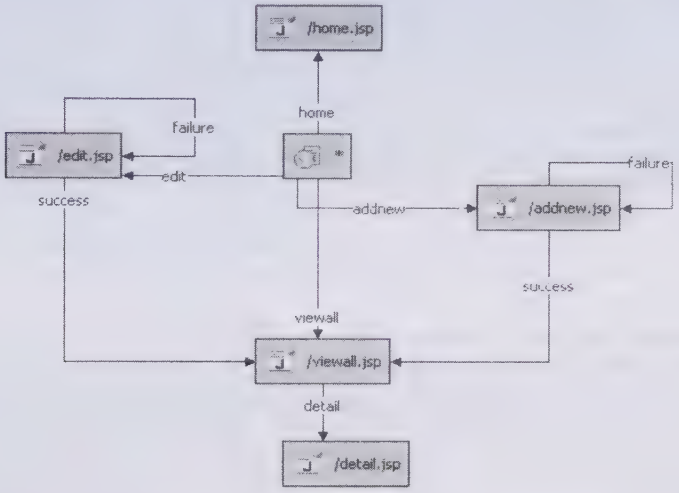


Figure 11.10: Showing the Various Navigational Paths from View to View

Let's learn about internationalization.

Supporting Internationalization

All the Locales supported by an application need to be configured in the `faces-config.xml` file. In addition, the location of resource bundle containing custom and localized messages is also placed under the `<application>` element, as shown in the following code snippet:

```
<application>

  <locale-config>
       <supported-locale>en</supported-locale>
       <supported-locale>fr</supported-locale>
  </locale-config>

  <message-bundle>com.kogent.ApplicationMessages</message-bundle>
</application>
```

In the preceding code snippet, we have created a resource bundle by the base name, `ApplicaitonMessages`. There are two different properties files containing custom messages to support two different Locales, English (`en`) and French (`fr`). The key and corresponding messages created to support default Locale is `ApplicationMessages_en.properties` file, as shown in Listing 11.12 (you can find this file on the CD in the `code\JavaEE\Chapter11\KogentPro\WEB-INF\classes\com\kogent\` folder):

Listing 11.12: Showing the ApplicationMessages_en.properties File

```
header=Kogent Solutions Inc.
footer=All rights are reserved with Kogent Solutions Inc.
home=Home
addnew=Add Employee
edit=Edit Employee
viewall=View All Employees
```

```
delete=Delete
update=Update
reset=Reset

details=Details
enterid=Enter Employee ID
empid=Employee ID
empname=Employee Name
address=Address

city=City
email=E-mail
sex=Sex
department=Department
skills=Skills
joindate=Joining Date
```

In Listing 11.12, the other properties file supports French Locale and has the same set of keys but with all messages translated to French. The key/value pairs of the `ApplicationMessages_fr.properties` file are shown in Listing 11.13 (you can find this file on the CD in the `code\JavaEE\Chapter11\KogentPro\WEB-INF\classes\com\kogent\` folder):

Listing 11.13: Showing the ApplicationMessages_fr.properties File

```
header=Kogent Solutions Inc.
footer=Tous les droits sont réservés par Kogent Solutions Inc
home=Accueil

addnew=Ajouter des employés
edit=Edit Employee
viewall=Voir tous les employés

delete=Supprimer
update=Update
reset=Reset
details=Détails

enterid=Entrez ID employé
empid=ID employé
empname=Employee Nom
address=Adresse

city=Ville
email=E-mail
sex=Sexe

department=Département
skills=Compétences

joindate=Participer Date
```

In the preceding code snippet, both the properties files, ApplicationMessages_en.properties and ApplicationMessages_fr.properties are placed in the com\kogent folder in the classes folder of the KogentPro application (Figure 11.6).

Exploring the Directory Structure of the Application

Let's create the KogentPro folder to place all the components of the KogenPro application. Figure 11.11 shows the directory structure of the KogentPro JSF application:

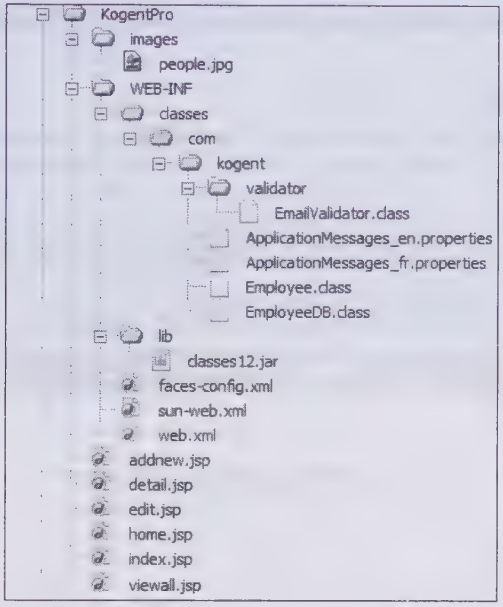


Figure 11.11: Displaying the Directory Structure of KogentPro Application

Let's now run the application.

Running the KogentPro Application

After successfully deploying the KogentPro application, you can launch the KogentPro application and access the home page, as shown in Figure 11.12:

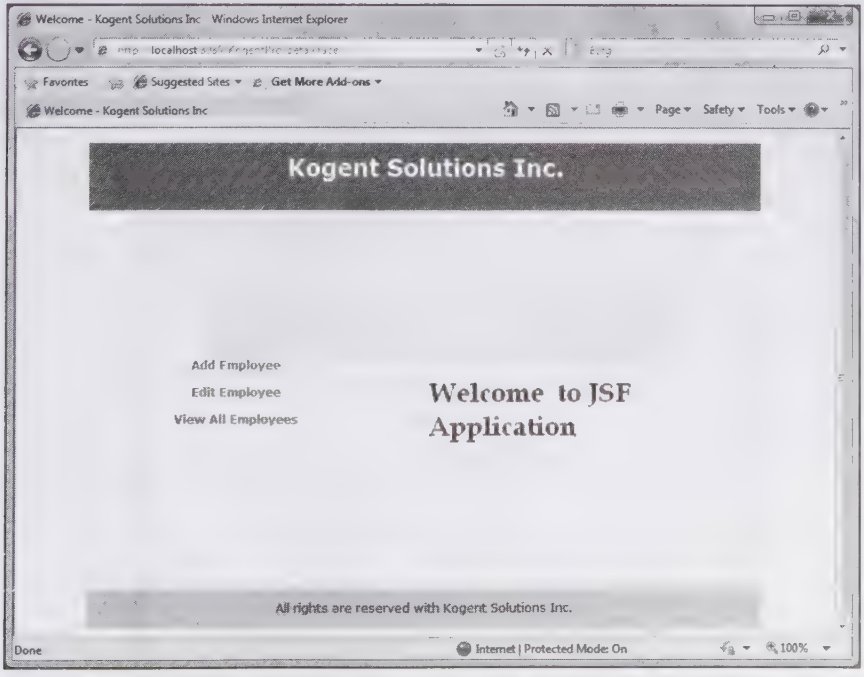


Figure 11.12: Displaying the Home Page of the KogentPro Application