Configuring JavaServer Faces Applications

The process of building and deploying simple JavaServer Faces applications has been described in earlier chapters of this tutorial.

When you create large and complex applications, however, various additional configuration tasks are required.

These tasks include the following:

- Registering back-end objects with the application so that all parts of the application have access to them

- Configuring backing beans and model beans so that they are instantiated with the proper values when a page makes reference to them

- Defining navigation rules for each of the pages in the application so that the application has a smooth page flow, if non-default navigation is needed

- Packaging the application to include all the pages, resources, and other files so that the application can be deployed on any compliant container

The following topics are addressed here:

- Using Annotations to Configure Managed Beans
- Application Configuration Resource File
- Configuring Beans
- Registering Custom Error Messages
- Registering Custom Localized Static Text
- Using Default Validators
- Configuring Navigation Rules
- Basic Requirements of a JavaServer Faces Application

Using Annotations to Configure Managed Beans

JavaServer Faces support for bean annotations has been introduced in Chapter 4, JavaServer Faces Technology.

Bean annotations can be used for configuring JavaServer Faces applications.

The `@ManagedBean` (`javax.faces.bean.ManagedBean`) annotation in a class automatically registers that class as a managed bean class with the server runtime.

Such a registered managed bean does not need `managed-bean` configuration entries in the application configuration resource file.

An example of using the @ManagedBean annotation on a class is as follows:

```
@ManagedBean
@SessionScoped
public class DukesBday{ . . . }
```

The above code snippet shows a bean that is managed by the JavaServer Faces implementation and is available for the length of that session.

You do not need to configure the managed bean instance in the `faces-config.xml` file.

In effect, it is an alternative to the application configuration resource file approach and reduces the task of configuring managed beans.

You can also define the scope of the managed bean within the class file, as shown in the above example.

You can annotate beans with request, session, or application scope, but not view scope.

All classes will be scanned for annotations at startup unless the `faces-config` element in the `faces-config.xml` file has the `metadata-complete` attribute set to `true`.

Annotations are also available for other artifacts such as components, converters, validators, and renderers to be used in place of application configuration resource file entries.

They are discussed, along with registration of custom listeners, custom validators, and custom converters, in Chapter 14, Creating Custom UI Components.

Using Managed Bean Scopes

You can use annotations to define the scope in which the bean will be stored.

You can specify one of the following scopes for a bean class:

- Application (`@ApplicationScoped`): Application scope persists across all users' interactions with a web application.

- Session (`@SessionScoped`): Session scope persists across multiple HTTP requests in a web application

- View (`@ViewScoped`): View scope persists during a user's interaction with a single page (view) of a web application.

- Request (`@RequestScoped`): Request scope persists during a single HTTP request in a web application.

- None (`@NoneScoped`):

Indicates a scope is not defined for the application.

- Custom (`@CustomScoped`): A user-defined, nonstandard scope.

Its value must be configured as a map.

Custom scopes are used infrequently.

You may want to use `@NoneScoped` when a managed bean references another managed bean.

The second bean should not be in a scope (`@NoneScoped`) if it is supposed to be created only when it is referenced.

If you define a bean as `@NoneScoped`, the bean is instantiated anew each time that it is referenced, and so it does not get saved in any scope.

If your managed bean takes part in a single HTTP request, you should define the bean with a request scope.

If you placed the bean in session or application scope instead, the bean would need to take precautions to ensure thread safety because component instances depend on running inside of a single thread.

If you are configuring a bean that allows attributes to be associated with the view, you can use the view scope.

The attributes persist until the user has navigated to the next view.

Eager Application—scoped Beans

Managed beans are lazily instantiated.

That is, that they are instantiated when a request is made from the application.

To force an application-scoped bean to be instantiated and placed in the application scope as soon as the application is started and before any request is made, the `eager` attribute of the managed bean should be set to `true` as shown in the following example:

```
@ManagedBean(eager=true)
@ApplicationScoped
```

Application Configuration Resource File

JavaServer Faces technology provides a portable configuration format (as an XML document) for configuring application resources.

One or more XML documents, called application configuration resource files, may use this format to register and configure objects and resources, and to define navigation rules for applications.

An application configuration resource file is usually named `faces-config.xml`.

You need an application configuration resource file in the following cases:

- To specify configuration elements for your application that are not available through managed bean annotations, such as localized messages and navigation rules
- To override managed bean annotations when the application is deployed

The application configuration resource file must be valid against the XML schema located at http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd.

In addition, each file must include the following information, in the following order:

- The XML version number, usually with an `encoding` attribute:

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

- A `faces-config` tag enclosing all the other declarations:

```
<faces-config version="2.0"
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-facesconfig_2_0.xsd">


  ...
  </faces-config>
```

You can have more than one application configuration resource file for an application.

The JavaServer Faces implementation finds the configuration file or files by looking for the following:

- A resource named `/META-INF/faces-config.xml` in any of the JAR files in the web application's `/WEB-INF/lib/` directory and in parent class loaders.

If a resource with this name exists, it is loaded as a configuration resource.

This method is practical for a packaged library

containing some components and renderers.

In addition, any file with a name that ends in `faces-config.xml` is also considered a configuration resource and is loaded as such.

- A context initialization parameter, `javax.faces.application.CONFIG_FILES`, in your web deployment descriptor file that specifies one or more (comma-delimited) paths to multiple configuration files for your web application.

This method is most often used for enterprise-scale applications that delegate to separate groups the responsibility for maintaining the file for each portion of a big application.

- A resource named `faces-config.xml` in the `/WEB-INF/` directory of your application.

Simple web applications make their configuration files available in this way.

To access the resources registered with the application, you can use an instance of the `Application` class, which is automatically created for each application.

The `Application` instance acts as a centralized factory for resources that are defined in the XML file.

When an application starts up, the JavaServer Faces implementation creates a single instance of the `Application` class and configures it with the information that you provided in the application configuration resource file.

Ordering of Application Configuration Resource Files

Because JavaServer Faces technology allows the use of multiple application configuration resource files stored in different locations, the order in which they are loaded by the implementation becomes important in certain situations (for example, when using application level objects).

This order can be defined through an `ordering` element and its sub-elements in the application configuration resource file itself.

The ordering of application configuration resource files can be absolute or relative.

Absolute ordering is defined by an `absolute-ordering` element in the file.

With absolute ordering, the user specifies the order in which application configuration resource files will be loaded.

The following example shows an entry for absolute ordering:

File `my-faces-config.xml`:

```
<faces-config>
<name>myJSF</name>
<absolute-ordering>
<name>A</name>
<name>B</name>
<name>C</name>
</absolute-ordering>
</faces-config>
```

In this example, A, B, and C are different application configuration resource files and are to be loaded in the listed order.

If there is an `absolute-ordering` element in the file, only the files listed by the sub-element `name` are processed.

To process any other application configuration resource files, an `others` sub-element is required.

In the absence of the `others` sub-element, all other unlisted files will be ignored at load time.

Relative ordering is defined by an `ordering` element and its sub-elements `before` and `after`.

With relative ordering, the order in which application configuration resource files will be loaded is calculated by considering ordering entries from the different files.

The following example shows some of these considerations.

In the following example, `config-A`, `config-B`, and `config-C` are different application configuration resource files.

File config-A contains the following elements:

```
<faces-config>
 <name>config-A</name>
 <ordering>
 <before>
 <name>config-B</name>
 </before>
 </ordering>
</faces-config>
```

File `config-B` (not shown here) does not contain any `ordering` elements.

File `config-C` contains the following elements:

```
<faces-config>
 <name>config-C</name>
 <ordering>
 <after>
 <name>config-B</name>
 </after>
 </ordering>
</faces-config>
```

Based on the `before` sub-element entry, file `config-A` will be loaded before the `config-B` file.

Based on the `after` sub-element entry, file `config-C` will be loaded after the `config-B` file.

In addition, a sub-element `others` can also be nested within the `before` and `after` sub-elements.

If the `others` element is present, the file may receive highest or lowest preference among both listed and unlisted configuration files.

If an `ordering` element is not present in an application configuration file, then that file will receive the lowest order when being loaded, compared to the files that contain an `ordering` element.

Configuring Beans

When a page references a managed bean for the first time, the JavaServer Faces implementation initializes it based on either a `@ManagedBean` annotation in the bean class, or according to its configuration in the application configuration resource file.

For information on using annotations to initialize beans, see Using Annotations to Configure Managed Beans.

You can use either annotations or the application configuration resource file to instantiate backing beans and other managed beans that are used in a JavaServer Faces application and to store them in scope.

The managed bean creation facility is configured in the application configuration resource file using `managed-bean` XML elements to define each bean.

This file is processed at application startup time.

For information on using this facility, see Using the `managed-bean` Element.

With the managed bean creation facility, you can:

- Create beans in one centralized file that is available to the entire application, rather than conditionally instantiate beans throughout the application
- Customize the bean's properties without any additional code

- Customize the bean's property values directly from within the configuration file so that it is initialized with these values when it is created

- Using `value` elements, set the property of one managed bean to be the result of evaluating another value expression

This section shows you how to initialize beans using the managed bean creation facility.

See Writing Bean Properties and Writing Backing Bean Methods for information on programming backing beans.

Using the managed-bean Element

A managed bean is initiated using a managed-bean element in the application configuration resource file, which represents an instance of a bean class that must exist in the application.

At runtime, the JavaServer Faces implementation processes the `managed-bean` element.

If a page references the bean, and if no bean instance exists, the JavaServer Faces implementation instantiates the bean as specified by the element configuration.

Here is an example managed bean configuration:

```
<managed-bean>
<managed-bean-name>
UserNumberBean
</managed-bean-name>
<managed-bean-class>
guessNumber.UserNumberBean
</managed-bean-class>
<managed-bean-scope>
session
</managed-bean-scope>
```

```
<managed-property>
<property-name>
maximum
</property-name>
<property-class>
long
</property-class>
<value> 10 </value>
</managed-property>
  . . .
</managed-bean>
```

Using NetBeans IDE, you can add a managed bean declaration by doing the following:

1. After opening your project in NetBeans IDE, expand the project node in the Projects pane.

2. Expand the Web Pages and WEB-INF nodes of the project node.

3. If there is no `faces-config.xml` in the project, create one as follows:

a. From the File menu, choose New File.

b. Select File→New File.

c. In the New File wizard, select the JavaServer Faces category, then select JSF Faces Configuration and click Next.

d. On the Name and Location page, change the name and location of the file if necessary.

The default file name is `faces-config.xml`.

e. Click Finish.

4. Double-click `faces-config.xml` if the file is not already open.

5. After `faces-config.xml` opens in the editor pane, select XML from the sub tab panel options.

6. Right-click in the editor pane.

7. From the Insert menu, choose Managed Bean.

8. In the Add Managed Bean dialog box:

   a. Type the display name of the bean in the Bean Name field.

   b. Click Browse to locate the bean's class.

9. In the Browse Class dialog box:

a. Start typing the name of the class that you are looking for in the Class Name field.

While you are typing, the dialog will show the matching classes.

b. Select the class from the Matching Classes box.

c. Click OK.

10. In the Add Managed Bean dialog box:

a. Select the bean's scope from the Scope menu.

b. Click Add.

The preceding steps will add the `managed-bean` element and three elements inside of that element: a `managed-bean-name` element, a `managed-bean-class` element, and a `managed-bean-scope` element.

You will need to edit the XML of the configuration file directly to further configure this managed bean.

The `managed-bean-name` element defines the key under which the bean will be stored in a scope.

For a component's value to map to this bean, the component tag's `value` attribute must match the `managed-bean-name` up to the first period.

The `managed-bean-class` element defines the fully qualified name of the JavaBeans component class used to instantiate the bean.

The `managed-bean` element can contain zero or more `managed-property` elements, each corresponding to a property defined in the bean class.

These elements are used to initialize the values of the bean properties.

If you don't want a particular property initialized with a value when the bean is instantiated, do not include a `managed-property` definition for it in your application configuration resource file.

If a `managed-bean` element does not contain other `managed-bean` elements, it can contain one `map-entries` element or `list-entries` element.

The `map-entries` element configures a set of beans that are instances of `Map`.

The `list-entries` element configures a set of beans that are instances of `List`.

To map to a property defined by a `managed-property` element, you must ensure that the part of a component tag's `value` expression after the period matches the `managed-property` element's `property-name` element.

In the earlier example, the `maximum` property is initialized with the value `10`.

Initializing Properties Using the `managed-property` Element explains in more detail how to use the `managed-property` element.

See Initializing Managed Bean Properties for an example of initializing a managed bean property.

Initializing Properties Using the `managed-property` Element

A `managed-property` element must contain a `property-name` element, which must match the name of the corresponding property in the bean.

A `managed-property` element must also contain one of a set of elements (listed in Table 11-1) that defines the value of the property.

This value must be of the same type as that defined for the property in the corresponding bean.

Which element you use to define the value depends on the type of the property defined in the bean.

Table 11-1 lists all the elements that are used to initialize a value.

Table 11-1 Sub-elements of `managed-property` Elements That Define Property Values

Element	Value That It Defines
`list-entries`	Defines the values in a list
`map-entries`	Defines the values of a map
`null-value`	Explicitly sets the property to `null`
`value`	Defines a single value, such as a `String`, `int`, or JavaServer Faces EL expression

Using the `managed-bean` Element includes an example of initializing an `int` property (a primitive type) using the `value` sub-element.

You also use the `value` sub-element to initialize `String` and other reference types.

The rest of this section describes how to use the `value` sub-element and other sub-elements to initialize properties of Java `Enum` types, `java.util.Map`, `array`, and `java.util.Collection`, as well as initialization parameters.

Referencing a Java `Enum` Type

A managed bean property can also be a Java `Enum` type (see http://download.oracle.com/javase/6/docs/api/java/lang/Enum.html).

In this case, the `value` element of the `managed-property` element must be a `String` that matches one of the `String` constants of the `Enum`.

In other words, the `String` must be one of the valid values that can be returned if you were to call `valueOf(Class, String)` on `enum`, where `Class` is the `Enum` class and `String` is the contents of the `value` subelement.

For example, suppose the managed bean property is the following:

```
public enum Suit
{ Hearts, Spades, Diamonds, Clubs}
...
public Suit getSuit()
{ ... return Suit.Hearts; }
```

Assuming that you want to configure this property in the application configuration resource file, the corresponding `managed-property` element would look like this:

```
<managed-property>
<property-name>Suit</property-name>
<value>Hearts</value>
</managed-property>
```

When the system encounters this property, it iterates over each of the members of the `enum` and calls `toString()` on each member until it finds one that is exactly equal to the value from the `value` element.

Referencing an Initialization Parameter

Another powerful feature of the managed bean creation facility is the ability to reference implicit objects from a managed bean property.

Suppose that you have a page that accepts data from a customer, including the customer's address.

Suppose also that most of your customers live in a particular area code.

You can make the area code component render this area code by saving it in an implicit object and referencing it when the page is rendered.

You can save the area code as an initial default value in the context `initParam` implicit object by adding a context parameter to your web application and setting its value in the deployment descriptor.

For example, to set a context parameter called `defaultAreaCode` to `650`, add a `context-param` element to the deployment descriptor, and give the parameter the name `defaultAreaCode` and the value `650`.

Next, you write a managed-bean declaration that configures a property that references the parameter:

```
<managed-bean>
<managed-bean-name>
customer
</managed-bean-name>
<managed-bean-class>
CustomerBean
</managed-bean-class>
```

```
<managed-bean-scope>
request
</managed-bean-scope>
<managed-property>
<property-name>
areaCode
</property-name>
<value>#{initParam.defaultAreaCode}
</value>
</managed-property> ...
</managed-bean>
```

To access the area code at the time that the page is rendered, refer to the property from the `area` component tag's `value` attribute:

```
<h:inputText id=area
value="#{customer.areaCode}"
```

Retrieving values from other implicit objects is done in a similar way.

Initializing Map Properties

The `map-entries` element is used to initialize the values of a bean property with a type of `java.util.Map` if the `map-entries` element is used within a `managed-property` element.

A `map-entries` element contains an optional `key-class` element, an optional `value-class` element, and zero or more `map-entry` elements.

Each of the `map-entry` elements must contain a `key` element and either a `null-value` or `value` element.

Here is an example that uses the map-entries element:

```
<managed-bean>
...
<managed-property>
<property-name>
prices
</property-name>
<map-entries>
<map-entry>
```

```
<key>
My Early Years: Growing Up on *7
</key>
<value>30.75</value>
</map-entry>
<map-entry>
<key>
Web Servers for Fun and Profit
</key>
<value>40.75</value>
</map-entry>
```

```
</map-entries>
</managed-property>
</managed-bean>
```

The map that is created from this `map-entries` tag contains two entries.

By default, all the keys and values are converted to `java.lang.String`.

If you want to specify a different type for the keys in the map, embed the `key-class` element just inside the `map-entries` element:

```
<map-entries>
<key-class>
java.math.BigDecimal
</key-class>
...
</map-entries>
```

This declaration will convert all the keys into `java.math.BigDecimal`.

Of course, you must make sure that the keys can be converted to the type that you specify.

The key from the example in this section cannot be converted to a `java.math.BigDecimal`, because it is a `String`.

If you also want to specify a different type for all the values in the map, include the `value-class` element after the `key-class` element:

```
<map-entries>
<key-class>int</key-class>
<value-class>
java.math.BigDecimal
</value-class>...
</map-entries>
```

Note that this tag sets only the type of all the `value` subelements.

The first `map-entry` in the preceding example includes a `value` subelement.

The `value` subelement defines a single value, which will be converted to the type specified in the bean.

The second `map-entry` defines a `value` element, which references a property on another bean.

Referencing another bean from within a bean property is useful for building a system from fine-grained objects.

For example, a request-scoped form-handling object might have a pointer to an application-scoped database mapping object.

Together the two can perform a form-handling task.

Note that including a reference to another bean will initialize the bean if it does not already exist.

Instead of using a `map-entries` element, it is also possible to assign the entire map using a `value` element that specifies a map-typed expression.

Initializing Array and List Properties

The `list-entries` element is used to initialize the values of an `array` or `java.util.List` property.

Each individual value of the array or `List` is initialized using a `value` or `null-value` element.

Here is an example:

```
<managed-bean>
...
<managed-property>
<property-name>
books
</property-name>
```

```
<list-entries>
<value-class>java.lang.String
</value-class>
<value>
Web Servers for Fun and Profit
</value>
<value>#{myBooks.bookId[3]}</value>
<null-value/>
</list-entries>
</managed-property>
</managed-bean>
```

This example initializes an `array` or a `List`.

The type of the corresponding property in the bean determines which data structure is created.

The `list-entries` element defines the list of values in the `array` or `List`.

The `value` element specifies a single value in the `array` or `List` and can reference a property in another bean.

The `null-value` element will cause the `setBooks` method to be called with an argument of `null`.

A `null` property cannot be specified for a property whose data type is a Java primitive, such as `int` or `boolean`.

Initializing Managed Bean Properties

Sometimes you might want to create a bean that also references other managed beans so that you can construct a graph or a tree of beans.

For example, suppose that you want to create a bean representing a customer's information, including the mailing address and street address, each of which is also a bean.

The following `managed-bean` declarations create a `CustomerBean` instance that has two `AddressBean` properties: one representing the mailing address, and the other representing the street address.

This declaration results in a tree of beans with `CustomerBean` as its root and the two `AddressBean` objects as children.

```
<managed-bean>
<managed-bean-name>
customer
</managed-bean-name>
<managed-bean-class>
com.mycompany.mybeans.CustomerBean
</managed-bean-class>
<managed-bean-scope>
request
</managed-bean-scope>
```

```
<managed-property>
<property-name>
mailingAddress
</property-name>
<value>#{addressBean}</value>
</managed-property>
<managed-property>
<property-name>
streetAddress
</property-name>
```

```
<value>#{addressBean}</value>
</managed-property>
<managed-property>
<property-name>
customerType
</property-name>
<value>New</value>
</managed-property>
</managed-bean>
```

```
<managed-bean>
<managed-bean-name>
addressBean
</managed-bean-name>
<managed-bean-class>
com.mycompany.mybeans.AddressBean
</managed-bean-class>
<managed-bean-scope>
none
</managed-bean-scope>
```

```
<managed-property>
<property-name>
street
</property-name>
<null-value/>
<managed-property>
  . . .
</managed-bean>
```

The first `CustomerBean` declaration (with the `managed-bean-name` of `customer`) creates a `CustomerBean` in request scope.

This bean has two properties, `mailingAddress` and `streetAddress`.

These properties use the `value` element to reference a bean named `addressBean`.

The second managed bean declaration defines an `AddressBean`, but does not create it, because its `managed-bean-scope` element defines a scope of `none`.

Recall that a scope of `none` means that the bean is created only when something else references it.

Because both the `mailingAddress` and the `streetAddress` properties reference `addressBean` using the `value` element, two instances of `AddressBean` are created when `CustomerBean` is created.

When you create an object that points to other objects, do not try to point to an object with a shorter life span, because it might be impossible to recover that scope's resources when it goes away.

A session-scoped object, for example, cannot point to a request-scoped object.

And objects with none scope have no effective life span managed by the framework, so they can point only to other none scoped objects.

Table 11-2 outlines all of the allowed connections.

Table 11-2 Allowable Connections Between Scoped Objects

An Object of This Scope	May Point to an Object of This Scope
`none`	`none`
`application`	`none, application`
`session`	`none, application, session`
`request`	`none, application, session, request, view`
`view`	`none, application, session, view`

Be sure not to allow cyclical references between objects.

For example, neither of the `AddressBean` objects in the preceding example should point back to the `CustomerBean` object, because `CustomerBean` already points to the `AddressBean` objects.

Initializing Maps and Lists

In addition to configuring `Map` and `List` properties, you can also configure a `Map` and a `List` directly so that you can reference them from a tag rather than referencing a property that wraps a `Map` or a `List`.

Registering Custom Error Messages

If you create custom error messages (which are displayed by the `message` and `messages` tags) for your custom converters or validators, you must make them available at application startup time.

You do this in one of two ways:

- By queuing the message onto the `FacesContext` instance programmatically, as described in Using `FacesMessage` to Create a Message
- By registering the messages with your application using the application configuration resource file

Here is an example of the section of the `faces-config.xml` file that registers the messages for an application:

```
<application>
<resource-bundle>
<base-name>
dukestutoring.web.messages.Messages
</base-name>
<var>bundle</var>
</resource-bundle>
```

```
<locale-config>
<default-locale>en</default-locale>
<supported-locale>
es
</supported-locale>
<supported-locale>
de
</supported-locale>


<supported-locale>
```

```
fr
</supported-locale>
</locale-config>
</application>
```

This set of elements will cause your `Application` instance to be populated with the messages that are contained in the specified resource bundle.

The `resource-bundle` element represents a set of localized messages.

It must contain the fully qualified path to the resource bundle containing the localized messages (in this case, `dukestutoring.web.messages.Messages`).

The `locale-config` element lists the default locale and the other supported locales.

The `locale-config` element enables the system to find the correct locale based on the browser's language settings.

The `supported-locale` and `default-locale` tags accept the lowercase, two-character codes as defined by ISO 639 (see http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt).

Make sure that your resource bundle actually contains the messages for the locales that you specify with these tags.

To access the localized message, the application developer merely references the key of the message from the resource bundle.

Using `FacesMessage` to Create a Message

Instead of registering messages in the application configuration resource file, you can access the `ResourceBundle` directly from backing bean code.

The code snippet below locates an email error message:

```
String message = ""; ...
message = ExampleBean.
loadErrorMessage(context,
ExampleBean.EX_RESOURCE_BUNDLE_NAME
, "EMailError");
context.addMessage(
toValidate.getClientId(context),
new FacesMessage(message));
```

These lines call the bean's `loadErrorMessage` method to get the message from the `ResourceBundle`.

Here is the `loadErrorMessage` method:

```
public static String
loadErrorMessage
 (FacesContext context,
String basename, String key)  {
```

```
if ( bundle == null ) {
try {
bundle =
ResourceBundle.getBundle(basename,
context.getViewRoot().getLocale());
} catch (Exception e)
{ return null; }
}
return bundle.getString(key);
}
```

Referencing Error Messages

A JavaServer Faces page uses the `message` or `messages` tags to access error messages, as explained in Displaying Error Messages with the `h:message` and `h:messages` Tags.

The error messages that these tags access include:

- The standard error messages that accompany the standard converters and validators that ship with the API.

See Section 2.5.2.4 of the JavaServer Faces specification for a complete list of standard error messages.

- Custom error messages contained in resource bundles registered with the application by the application architect using the `resource-bundle` element in the configuration file.

When a converter or validator is registered on an input component, the appropriate error message is automatically queued on the component.

A page author can override the error messages queued on a component by using the following attributes of the component's tag:

- `converterMessage`: References the error message to display when the data on the enclosing component can not be converted by the converter registered on this component.

- `requiredMessage`: References the error message to display when no value has been entered into the enclosing component.

- `validatorMessage`: References the error message to display when the data on the enclosing component cannot be validated by the validator registered on this component.

All three attributes are enabled to take literal values and value expressions.

If an attribute uses a value expression, this expression references the error message in a resource bundle.

This resource bundle must be made available to the application in one of the following ways:

- By the application architect using the `resource-bundle` element in the configuration file

- By the page author using the `f:loadBundle` tag

Conversely, the `resource-bundle` element must be used to make available to the application those resource bundles containing custom error messages that are queued on the component as a result of a custom converter or validator being registered on the component.

The following tags show how to specify the `requiredMessage` attribute using a value expression to reference an error message:

```
<h:inputText id="ccno" size="19"
required="true"
requiredMessage=
"#{customMessages.ReqMessage}"
>
...
</h:inputText>
<h:message styleClass="error-
message" for="ccno"/>
```

The value expression that `requiredMessage` is using in this example references the error message with the `ReqMessage` key in the resource bundle, `customMessages`.

This message replaces the corresponding message queued on the component and will display wherever the `message` or `messages` tag is placed on the page.

Registering Custom Localized Static Text

You can register custom localized static text with the application by using the `resource-bundle` element in the application configuration resource file.

Any custom error messages that are referenced by the `converterMessage`, `requiredMessage`, or `validatorMessage` attributes of an input component tag can be made available to the application by using the `resource-bundle` element of the application configuration file.

Here is the part of a file that registers some custom error messages:

```
<application>
...
<resource-bundle>
<base-name>
guessNumber.ApplicationMessages
</base-name>
<var>customMessages</var>
</resource-bundle>
...
</application>
```

The value of the `base-name` sub-element specifies the fully-qualified class name of the `ResourceBundle` class, which in this case is located in the `resources` package of the application.

The `var` sub-element of the `resource-bundle` element is an alias to the `ResourceBundle` class.

This alias is used by tags in the page to identify the resource bundle.

The `locale-config` element shown in the previous section also applies to the messages and static text identified by the `resource-bundle` element.

As with resource bundles identified by the `message-bundle` element, make sure that the resource bundle identified by the `resource-bundle` element actually contains the messages for the locales that you specify with these `locale-config` elements.

You can also pull localized text into an `alt` tag for a graphic image, as in the following example:

```
<h:graphicImage id="mapImage"
url="/template/world.jpg"
alt="#{bundle.ChooseLocale}"
usemap="#worldMap"
/>
```

The `alt` attribute can accept value expressions.

In this case, the `alt` attribute refers to localized text that will be included in the alternative text of the image rendered by this tag.

You can also use the `f:loadBundle` tag to load a resource bundle.

This tag has attributes named `var` and `basename` that specify the same values as the `var` and `basename` sub-elements of the resource bundle specification in the configuration file.

Using Default Validators

In addition to the validators you declare on the components, you can also specify zero or more default validators in the application configuration resource file.

The default validator applies to all `UIInput` instances in a view or component tree and is appended after the local defined validators.

Here is an example of a default validator registered in the application configuration resource file:

```
<faces-config>
<application>
<default-validators>
<validator-id>
javax.faces.Bean
</validator-id>
</default-validators>
<application/>
</faces-config>
```

Configuring Navigation Rules

Navigation between different pages of a JavaServer Faces application, such as choosing the next page to be displayed after a button or hyperlink component is clicked, is defined by a set of rules.

Navigation rules can be implicit, or they can be explicitly defined in the application configuration resource file.

For more information on implicit navigation rules, see Implicit Navigation Rules.

Each navigation rule specifies how to navigate from one page to another page or a set of other pages.

The JavaServer Faces implementation chooses the proper navigation rule according to which page is currently displayed.

After the proper navigation rule is selected, the choice of which page to access next from the current page depends on two factors:

- The action method that was invoked when the component was clicked

- The logical outcome that is referenced by the component's tag or was returned from the action method

The outcome can be anything that the developer chooses, but Table 11-3 lists some outcomes commonly used in web applications.

Table 11-3 Common Outcome Strings

Outcome	What It Means
`success`	Everything worked. Go on to the next page.
`failure`	Something is wrong. Go on to an error page.
`login`	The user needs to log in first. Go on to the login page.
`no results`	The search did not find anything. Go to the search page again.

Usually, the action method performs some processing on the form data of the current page.

For example, the method might check whether the user name and password entered in the form match the user name and password on file.

If they match, the method returns the outcome `success`.

Otherwise, it returns the outcome `failure`.

As this example demonstrates, both the method used to process the action and the outcome returned are necessary to determine the proper page to access.

Here is a navigation rule that could be used with the example just described:

```
<navigation-rule>
<from-view-id>
/login.xhtml
</from-view-id>
<navigation-case>
<from-action>
#{LoginForm.login}
</from-action>
<from-outcome>
success
</from-outcome>
```

```
<to-view-id>
/storefront.xhtml
</to-view-id>
</navigation-case>
<navigation-case>
<from-action>
#{LoginForm.logon}
</from-action>
<from-outcome>
failure
</from-outcome>
```

```
<to-view-id>
/logon.xhtml
</to-view-id>
</navigation-case>
</navigation-rule>
```

This navigation rule defines the possible ways to navigate from `login.xhtml`.

Each `navigation-case` element defines one possible navigation path from `login.xhtml`.

The first `navigation-case` says that if `LoginForm.login` returns an outcome of `success`, then `storefront.xhtml` will be accessed.

The second `navigation-case` says that `login.xhtml` will be re-rendered if `LoginForm.login` returns `failure`.

The configuration of an application's page flow consists of a set of navigation rules.

Each rule is defined by the `navigation-rule` element in the `faces-config.xml` file.

Each `navigation-rule` element corresponds to one component tree identifier defined by the optional `from-view-id` element.

This means that each rule defines all the possible ways to navigate from one particular page in the application.

If there is no `from-view-id` element, the navigation rules defined in the `navigation-rule` element apply to all the pages in the application.

The `from-view-id` element also allows wildcard matching patterns.

For example, this `from-view-id` element says that the navigation rule applies to all the pages in the `books` directory:

```
<from-view-id>
/books/*
</from-view-id>
```

A `navigation-rule` element can contain zero or more `navigation-case` elements.

The `navigation-case` element defines a set of matching criteria.

When these criteria are satisfied, the application will navigate to the page defined by the `to-view-id` element contained in the same `navigation-case` element.

The navigation criteria are defined by optional `from-outcome` and `from-action` elements.

The `from-outcome` element defines a logical outcome, such as `success`.

The `from-action` element uses a method expression to refer to an action method that returns a `String`, which is the logical outcome.

The method performs some logic to determine the outcome and returns the outcome.

The `navigation-case` elements are checked against the outcome and the method expression in this order:

- Cases specifying both a `from-outcome` value and a `from-action` value.

Both of these elements can be used if the action method returns different outcomes depending on the result of the processing it performs.

- Cases specifying only a `from-outcome` value.

The `from-outcome` element must match either the outcome defined by the `action` attribute of the `UICommand` component or the outcome returned by the method referred to by the `UICommand` component.

- Cases specifying only a `from-action` value.

This value must match the `action` expression specified by the component tag.

When any of these cases is matched, the component tree defined by the `to-view-id` element will be selected for rendering.

Using NetBeans IDE, you can configure a navigation rule by doing the following:

1. After opening your project in NetBeans IDE, expand the project node in the Projects pane.

2. Expand the Web Pages and WEB-INF nodes of the project node.

3. Double-click `faces-config.xml`.

4. After `faces-config.xml` opens in the editor pane, right-click in the editor pane.

5. From the Insert menu, choose Navigation Rule.

6. In the Add Navigation Rule dialog:
   a. Enter or browse for the page that represents the starting view for this navigation rule.
   b. Click Add.

7. Right-click again in the editor pane.

8. From the Insert menu, choose Navigation Case.

9. In the Add Navigation Case dialog box:

a. From the From View menu, choose the page that represents the starting view for the navigation rule (from Step 6 a).

b. (optional) In the From Action field, type the action method invoked when the component that triggered navigation is activated.

c. (optional) In the From Outcome field, enter the logical outcome string that the activated component references from its `action` attribute.

d. From the To View menu, choose or browse for the page that will be opened if this navigation case is selected by the navigation system.

e. Click Add.

Referencing a Method That Performs Navigation explains how to use a component tag's `action` attribute to point to an action method.

Writing a Method to Handle Navigation explains how to write an action method.

Implicit Navigation Rules

JavaServer Faces technology supports implicit navigation rules for Facelets applications.

Implicit navigation applies when `navigation-rules` are not configured in the application configuration resource files.

When you add a component such as a `commandButton` in a page, and assign another page as the value for its `action` property, the default navigation handler will try to match a suitable page within the application implicitly.

```
<h:commandButton value="submit"
action="response"
>
```

In the above example, the default navigation handler will try to locate a page named `response.xhtml` within the application and navigate to it.

Basic Requirements of a JavaServer Faces Application

In addition to configuring your application, you must satisfy other requirements of JavaServer Faces applications, including properly packaging all the necessary files and providing a deployment descriptor.

This section describes how to perform these administrative tasks.

JavaServer Faces 2.x applications must be compliant with at least version 2.5 of the Servlet specification, and at least version 2.1 of the JavaServer Pages specification.

All applications compliant with these specifications can be packaged in a WAR file, which must conform to specific requirements to execute across different containers.

At a minimum, a WAR file for a JavaServer Faces application must contain the following:

- A web application deployment descriptor, called `web.xml`, to configure resources required by a web application

- An application configuration resource file, which configures application resources

- A specific set of JAR files containing essential classes

- A set of application classes, JavaServer Faces pages, and other required resources, such as image files

For example, a Java Server Faces web application WAR file using Facelets typically has the following directory structure:

```
$PROJECT_DIR
[Web Pages]
+- /[xhtml documents]
+- /resources
+- /WEB-INF
  +- /classes
  +- /lib
  +- /web.xml
  +- /faces-config.xml
  +- /glassfish-web.xml
```

The `web.xml` file (or web deployment descriptor), the set of JAR files, and the set of application files must be contained in the `WEB-INF` directory of the WAR file.

Configuring an Application With a Web Deployment Descriptor

Web applications are commonly configured using elements that are contained in the web application deployment descriptor.

The deployment descriptor for a JavaServer Faces application must specify certain configurations, which include the following:

- The servlet used to process JavaServer Faces requests
- The servlet mapping for the processing servlet
- The path to the configuration resource file, if it exists and is not located in a default location

The deployment descriptor can also specify other, optional configurations, including:

- Specifying where component state is saved
- Encrypting state saved on the client
- Compressing state saved on the client
- Restricting access to pages containing JavaServer Faces tags
- Turning on XML validation
- Information regarding Project Stage
- Verifying custom objects

This section gives more details on these configurations.

Where appropriate, it also describes how you can make these configurations using NetBeans IDE.

Identifying the Servlet for Lifecycle Processing

A requirement of a JavaServer Faces application is that all requests to the application that reference previously saved JavaServer Faces components must go through `javax.faces.webapp.FacesServlet`.

A `FacesServlet` instance manages the request processing lifecycle for web applications and initializes the resources required by JavaServer Faces technology.

Before a JavaServer Faces application can launch its first web page, the web container must invoke the `FacesServlet` instance in order for the application lifecycle process to start.

The following example shows the default configuration of the FacesServlet:

```
<servlet>
<servlet-name>
FacesServlet
</servlet-name>
<servlet-class>
javax.faces.webapp.FacesServlet
</servlet-class>
 </servlet>
```

To make sure that the `FacesServlet` instance is invoked, you must provide a mapping to it.

The mapping to `FacesServlet` can be a prefix mapping, such as `/faces/*`, or an extension mapping, such as `*.faces`.

The mapping is used to identify a page as having JavaServer Faces content.

Because of this, the URL to the first page of the application must include the URL pattern mapping.

```
<servlet-mapping>
<servlet-name>
FacesServlet
</servlet-name>
<url-pattern>
/faces/*
</url-pattern>
</servlet-mapping>
```

In the case of prefix mapping, there are two ways to accomplish this:

- The page author can include a simple HTML page, such as an `index.xhtml` file in the application that provides the URL to the first page.

This URL must include the path to `FacesServlet`, as shown by this tag, which uses the mapping defined in the `guessNumber` application:

```
<a href="faces/greeting.xhtml">
```

- Users of the application can include the path to `FacesServlet` in the URL to the first page when they enter it in their browser, as shown in the example below:

```
http://localhost:8080/guessNumber/
faces/greeting.xhtml
```

The second method allows users to start the application from the first page of the application, rather than start it from another HTML page.

However, the second method requires users to identify the first page of the application in the URL.

When you use the first method, users need only enter the path to the application, as shown in the following example:

http://localhost:8080/guessNumber

In the case of extension mapping, if a request comes to the server for a page with a `.faces` extension, the container will send the request to the `FacesServlet` instance, which will expect a corresponding page of the same name to exist containing the content.

If you are using NetBeans IDE to create your application, a web deployment descriptor is automatically created for you with default configurations.

If you created your application without an IDE, you can create a web deployment descriptor.

Specifying a Path to an Application Configuration Resource File

As explained in Application Configuration Resource File, an application can have multiple application configuration resource files.

If these files are not located in the directories that the implementation searches by default or the files are not named `faces-config.xml`, you need to specify paths to these files.

To specify these paths using NetBeans IDE, do the following:

1. Expand the node of your project in the Projects pane.

2. Expand the Web Pages and WEB-INF nodes that are under the project node.

3. Double-click `web.xml`.

4. After the `web.xml` file appears in the editor pane, click General at the top of the editor pane.

5. Expand the Context Parameters node.

6. Click Add.

7. In the Add Context Parameter dialog:
   a. Enter `javax.faces.CONFIG_FILES` in the Param Name field.
   b. Enter the path to your configuration file in the Param Value field.
   c. Click OK.

8. Repeat steps 1 through 7 for each configuration file.

To specify paths to the files by editing the deployment descriptor directly, follow these steps:

1. Add a `context-param` element to the deployment descriptor.

2. Add a `param-value` element inside the `context-param` element and call it `javax.faces.CONFIG_FILES`.

3. Add a `param-value` element inside the `context-param` element and give it the path to your configuration file.

For example, the path to the `guessNumber` application's application configuration resource file is `/WEB-INF/faces-config.xml`.

4. Repeat steps 2 and 3 for each application configuration resource file that your application contains.

Specifying Where State Is Saved

When implementing the state-holder methods, you specify in your deployment descriptor where you want the state to be saved, either client or server.

You do this by setting a context parameter in your deployment descriptor.

To specify where state is saved using NetBeans IDE, do the following:

1. Expand the node of your project in the Projects pane.

2. Expand the Web Pages and WEB-INF nodes that are under the project node.

3. Double-click `web.xml`.

4. After the `web.xml` file appears in the editor pane, click General at the top of the editor pane.

5. Expand the Context Parameters node.

6. Click Add.

7. In the Add Context Parameter dialog:
   a. Enter
      `javax.faces.STATE_SAVING_METHOD`
      in the Param Name field.
   b. Enter `client` or `server` in the Param Value field.
   c. Click OK.

To specify where state is saved by editing the deployment descriptor directly, follow these steps:

1. Add a `context-param` element to the deployment descriptor.

2. Add a `param-name` element inside the `context-param` element and give it the name `javax.faces.STATE_SAVING_METHOD`.

3. Add a `param-value` element to the `context-param` element and give it the value `client` or `server`, depending on whether you want state saved in the client or the server.

If state is saved on the client, the state of the entire view is rendered to a hidden field on the page.

The JavaServer Faces implementation saves the state on the client by default.

Duke's Bookstore saves its state in the client.

Configuring Project Stage

Project Stage is a context parameter identifying the status of a JavaServer Faces application in the software lifecycle.

The stage of an application can affect the behavior of the application.

For example, error messages can be displayed during the Development stage but suppressed during the Production stage.

The possible Project Stage values are as follows:

- `Production`
- `Development`
- `UnitTest`
- `SystemTest`
- `Extension`

Project Stage is configured through a context parameter in the web deployment descriptor file.

Here is an example:

```
<context-param>
<param-name>
javax.faces.PROJECT_STAGE
</param-name>
```

```
<param-value>
Development
</param-value>
</context-param>
```

If no Project Stage is defined, the default stage is considered as `Development`.

You can also add custom stages according to your requirements.

The Project Stage value can also be configured through JNDI.

When using JNDI based Project Stage, you need to configure the JNDI resource in the web deployment descriptor file.

```
<resource-ref>
<res-ref-name>
jsf/ProjectStage
</res-ref-name>
```

```
<res-type>
java.lang.String
</res-type>
</resource-ref>
```

Including the Classes, Pages, and Other Resources

When packaging web applications using the included build scripts, you'll notice that the scripts package resources in the following ways:

- All web pages are placed at the top level of the WAR file.

- The `faces-config.xml` file and the `web.xml` file are packaged in the `WEB-INF` directory.

- All packages are stored in the `WEB-INF/classes/` directory.

- All application JAR files are packaged in the `WEB-INF/lib/` directory.

- All resource files are either under the root of the web application `/resources` directory, or in the web application's classpath, `META-INF/resources/<resourceIdentifier>` directory.

For more information on resources, see Resources.

When packaging your own applications, you can use NetBeans IDE or you can use the build scripts such as those built for Ant.

You can modify the build scripts to fit your situation.

However, you can continue to package your WAR files by using the directory structure described in this section, because this technique complies with the commonly accepted practice for packaging web applications.