Developing with JavaServer Faces Technology

Chapter 7, Using JavaServer Faces Technology in Web Pages and Chapter 8, Using Converters, Listeners, and Validators show how to add components to a page and connect them to server-side objects by using component tags and core tags,

as well as how to provide additional functionality to the components through converters, listeners, and validators.

Developing a JavaServer Faces application also involves the task of programming the server-side objects: backing beans, converters, event handlers, and validators.

This chapter provides an overview of backing beans and explains how to write methods and properties of backing beans that are used by a JavaServer Faces application.

This chapter also introduces the Bean Validation feature.

The following topics are addressed here:

- Backing Beans
- Writing Bean Properties
- Writing Backing Bean Methods
- Using Bean Validation

Backing Beans

A typical JavaServer Faces application includes one or more backing beans, each of which is a type of JavaServer Faces managed bean that can be associated with the components used in a particular page.

This section introduces the basic concepts of creating, configuring, and using backing beans in an application.

Creating a Backing Bean

A backing bean is created with a constructor with no arguments (like all JavaBeans components) and a set of properties and a set of methods that perform functions for a component.

Each of the backing bean properties can be bound to one of the following:

- A component value
- A component instance
- A converter instance
- A listener instance
- A validator instance

The most common functions that backing bean methods perform include the following:

- Validating a component's data
- Handling an event fired by a component
- Performing processing to determine the next page to which the application must navigate

As with all JavaBeans components, a property consists of a private data field and a set of accessor methods, as shown by this code:

```
Integer userNumber = null; ...
public void setUserNumber
(Integer user_number)
{ userNumber = user_number; }
public Integer getUserNumber()
{ return userNumber; }
public String getResponse() { ... }
```

When bound to a component's value, a bean property can be any of the basic primitive and numeric types or any Java object type for which the application has access to an appropriate converter.

For example, a property can be of type `Date` if the application has access to a converter that can convert the `Date` type to a `String` and back again.

See Writing Bean Properties for information on which types are accepted by which component tags.

When a bean property is bound to a component instance, the property's type must be the same as the component object.

For example, if a `javax.faces.component.UISelectBoolean` component is bound to the property, the property must accept and return a `UISelectBoolean` object.

Likewise, if the property is bound to a converter, validator, or listener instance, the property must be of the appropriate converter, validator, or listener type. For more information on writing beans and their properties, see Writing Bean Properties.

Using the EL to Reference Backing Beans

To bind component values and objects to backing bean properties or to reference backing bean methods from component tags, page authors use the Expression Language syntax.

As explained in Overview of the EL, the following are some of the features that EL offers:

- Deferred evaluation of expressions
- The ability to use a value expression to both read and write data
- Method expressions

Deferred evaluation of expressions is important because the JavaServer Faces lifecycle is split into several phases in which component event handling, data conversion and validation, and data propagation to external objects are all performed in an orderly fashion.

The implementation must be able to delay the evaluation of expressions until the proper phase of the lifecycle has been reached.

Therefore, the implementation's tag attributes always use deferred-evaluation syntax, which is distinguished by the #{ } delimiter.

To store data in external objects, almost all JavaServer Faces tag attributes use lvalue expressions, which are expressions that allow both getting and setting data on external objects.

Finally, some component tag attributes accept method expressions that reference methods that handle component events or validate or convert component data.

To illustrate a JavaServer Faces tag using the EL, suppose that a tag of an application referenced a method to perform the validation of user input:

```
<h:inputText id="userNo"
value=
"#{UserNumberBean.userNumber}"
validator=
"#{UserNumberBean.validate}"
/>
```

This tag binds the `userNo` component's value to the `UserNumberBean.userNumber` backing bean property by using an lvalue expression.

The tag uses a method expression to refer to the `UserNumberBean.validate` method, which performs validation of the component's local value.

The local value is whatever the user enters into the field corresponding to this tag.

This method is invoked when the expression is evaluated.

Nearly all JavaServer Faces tag attributes accept value expressions.

In addition to referencing bean properties, value expressions can reference lists, maps, arrays, implicit objects, and resource bundles.

Another use of value expressions is binding a component instance to a backing bean property.

A page author does this by referencing the property from the `binding` attribute:

```
<inputText
binding=
"#{UserNumberBean.userNoComponent}"
/>
```

In addition to using expressions with the standard component tags, you can configure your custom component properties to accept expressions by creating `javax.el.ValueExpression` or `javax.el.MethodExpression` instances for them.

For information on the EL, see Chapter 6, Expression Language.

For information on referencing backing bean methods from component tags, see Referencing a Backing Bean Method.

Writing Bean Properties

As explained in Backing Beans, a backing bean property can be bound to one of the following items:

- A component value
- A component instance
- A converter implementation
- A listener implementation
- A validator implementation

These properties follow the conventions of JavaBeans components (also called beans).

For more information on JavaBeans components, see the *JavaBeans Tutorial* at http://download.oracle.com/javase/tutorial/javabeans/index.html.

The component's tag binds the component's value to a backing bean property by using its `value` attribute and binds the component's instance to a backing bean property by using its `binding` attribute.

Likewise, all the converter, listener, and validator tags use their `binding` attributes to bind their associated implementations to backing bean properties.

To bind a component's value to a backing bean property, the type of the property must match the type of the component's value to which it is bound.

For example, if a backing bean property is bound to a `UISelectBoolean` component's value, the property should accept and return a `boolean` value or a `Boolean` wrapper `Object` instance.

To bind a component instance to a backing bean property, the property must match the type of component.

For example, if a backing bean property is bound to a `UISelectBoolean` instance, the property should accept and return a `UISelectBoolean` value.

Similarly, to bind a converter, listener, or validator implementation to a backing bean property, the property must accept and return the same type of converter, listener, or validator object.

For example, if you are using the `convertDateTime` tag to bind a `DateTimeConverter` to a property, that property must accept and return a `DateTimeConverter` instance.

The rest of this section explains how to write properties that can be bound to component values, to component instances for the component objects described in Adding Components to a Page Using HTML Tags, and to converter, listener, and validator implementations.

Writing Properties Bound to Component Values

To write a backing bean property that is bound to a component's value, you must match the property type to the component's value.

Table 9-1 lists the `javax.faces.component` classes and the acceptable types of their values.

Table 9-1 Acceptable Types of Component Values

Component Class	Acceptable Types of Component Values
`UIInput`, `UIOutput`, `UISelectItem`, `UISelectOne`	Any of the basic primitive and numeric types or any Java programming language object type for which an appropriate `Converter` implementation is available
`UIData`	`array` of beans, `List` of beans, single bean, `java.sql.ResultSet`, `javax.servlet.jsp.jstl.sql.Result`, `javax.sql.RowSet`
`UISelectBoolean`	`boolean` or `Boolean`
`UISelectItems`	`java.lang.String`, `Collection`, `Array`, `Map`
`UISelectMany`	`array` or `List`, though elements of the `array` or `List` can be any of the standard types

When they bind components to properties by using the `value` attributes of the component tags, page authors need to ensure that the corresponding properties match the types of the components' values.

UIInput and UIOutput Properties

In the following example, an `h:inputText` tag binds the `name` component to the `name` property of a backing bean called `CashierBean`.

```
<h:inputText id="name" size="50"
value="#{cashier.name}">
</h:inputText>
```

The following code snippet from the backing bean CashierBean shows the bean property type bound by the preceding component tag:

```
protected String name = null;
public void setName(String name)
{ this.name = name; }
public String getName()
{ return this.name; }
```

As described in Using the Standard Converters, to convert the value of an input or output component, you can either apply a converter or create the bean property bound to the component with the matching type.

Here is the example tag, from Using `DateTimeConverter`, that displays the date when items will be shipped.

```
<h:outputText
value="#{cashier.shipDate}">
<f:convertDateTime dateStyle="full" />
</h:outputText>
```

The bean property represented by this tag must have a type of `java.util.Date`.

The following code snippet shows the `shipDate` property, from the backing bean `CashierBean`, that is bound by the tag's value in the preceding example:

```
protected Date shipDate;
public Date getShipDate()
{ return this.shipDate; }
public void setShipDate
(Date shipDate)
{ this.shipDate = shipDate; }
```

`UIData` *Properties*

Data components must be bound to one of the backing bean property types listed in Table 9-1.

Data components are discussed in Using Data-Bound Table Components.

Here is part of the start tag of `dataTable` from that section:

```
<h:dataTable id="items" ...
value="#{cart.items}"
var="item" >
```

The value expression points to the `items` property of a shopping cart bean named `cart`.

The `cart` bean maintains a map of `ShoppingCartItem` beans.

The `getItems` method from the `cart` bean populates a `List` with `ShoppingCartItem` instances that are saved in the items map when the customer adds items to the cart, as shown in the following code segment:

```
public synchronized List getItems()
{
List results = new ArrayList();

results.addAll
```

```
(this.items.values());
return results;
}
```

All the components contained in the data component are bound to the properties of the `cart` bean that is bound to the entire data component.

For example, here is the h:outputText tag that displays the item name in the table:

```
<h:commandLink
action="#{showcart.details}"
>
<h:outputText
value="#{item.item.name}"
/>
</h:commandLink>
```

`UISelectBoolean` *Properties*

Backing bean properties that hold a `UISelectBoolean` component's data must be of `boolean` or `Boolean` type.

The example `selectBooleanCheckbox` tag from the section Displaying Components for Selecting One Value binds a component to a property.

The following example shows a tag that binds a component value to a boolean property:

```
<h:selectBooleanCheckbox
title="#{bundle.receiveEmails}"
value=
"#{custFormBean.receiveEmails}"
>
</h:selectBooleanCheckbox>
<h:outputText
value="#{bundle.receiveEmails}">
```

Here is an example property that can be bound to the component represented by the example tag:

```
protected boolean receiveEmails =
false; ...
public void setReceiveEmails
(boolean receiveEmails)
{this.receiveEmails = receiveEmails;}
public boolean getReceiveEmails()
{ return receiveEmails; }
```

`UISelectMany` *Properties*

Because a `UISelectMany` component allows a user to select one or more items from a list of items, this component must map to a bean property of type `List` or `array`.

This bean property represents the set of currently selected items from the list of available items.

The following example of the selectManyCheckbox tag comes fromDisplaying Components for Selecting Multiple Values:

```
<h:selectManyCheckbox
id="newsletters"
layout="pageDirection"
value="#{cashier.newsletters}"
>
```

```
<f:selectItems
value="#{newsletters}"
/>
</h:selectManyCheckbox>
```

Here is the bean property that maps to the `value` of the `selectManyCheckbox` tag from the preceding example:

```
protected String newsletters[] =
new String[0];
public void setNewsletters
(String newsletters[])
{ this.newsletters = newsletters; }
public String[] getNewsletters()
{ return this.newsletters; }
```

The `UISelectItem` and `UISelectItems` components are used to represent all the values in a `UISelectMany` component.

See `UISelectItem` Properties and `UISelectItems` Properties for information on writing the bean properties for the `UISelectItem` and `UISelectItems` components.

`UISelectOne` *Properties*

`UISelectOne` properties accept the same types as `UIInput` and `UIOutput` properties, because a `UISelectOne` component represents the single selected item from a set of items.

This item can be any of the primitive types and anything else for which you can apply a converter.

Here is an example of the selectOneMenu tag from Displaying a Menu Using the h:selectOneMenu Tag:

```
<h:selectOneMenu
id="shippingOption"
required="true"
value="#{cashier.shippingOption}"
>
```

```
<f:selectItem
itemValue="2"
itemLabel="#{bundle.QuickShip}"/>
<f:selectItem
itemValue="5"
itemLabel="#{bundle.NormalShip}"/>
<f:selectItem
itemValue="7"
itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>
```

Here is the bean property corresponding to this tag:

```
protected String shippingOption =
"2";
public void setShippingOption
(String shippingOption) {
this.shippingOption =
shippingOption;
}
```

```
public String getShippingOption()
{ return this.shippingOption; }
```

Note that `shippingOption` represents the currently selected item from the list of items in the `UISelectOne` component.

The `UISelectItem` and `UISelectItems` components are used to represent all the values in a `UISelectOne` component.

This is explained in the section Displaying a Menu Using the `h:selectOneMenu` Tag.

For information on how to write the backing bean properties for the `UISelectItem` and `UISelectItems` components, see `UISelectItem` Properties and `UISelectItems` Properties.

`UISelectItem` Properties

A `UISelectItem` component represents a single value in a set of values in a `UISelectMany` or a `UISelectOne` component.

A `UISelectItem` component must be bound to a backing bean property of type `javax.faces.model.SelectItem`.

A `SelectItem` object is composed of an `Object` representing the value, along with two `Strings` representing the label and description of the `UISelectItem` object.

The example `selectOneMenu` tag from Displaying a Menu Using the `h:selectOneMenu` Tag contains `selectItem` tags that set the values of the list of items in the page.

Here is an example of a bean property that can set the values for this list in the bean:

```
SelectItem itemOne = null;
SelectItem getItemOne()
{ return itemOne; }
void setItemOne(SelectItem item)
{ itemOne = item; }
```

`UISelectItems` *Properties*

`UISelectItems` components are children of `UISelectMany` and `UISelectOne` components.

Each `UISelectItems` component is composed of a set of either `javax.faces.model.SelectItem` instances or any collection of objects, such as an array, a list, or even POJOs.

This section explains how to write the properties for `selectItems` tags containing `SelectItem` instances.

You can populate the `UISelectItems` with `SelectItem` instances programmatically in the backing bean.

1. In your backing bean, create a list that is bound to the `SelectItem` component.
2. Define a set of `SelectItem` objects, set their values, and populate the list with the `SelectItem` objects.

The following example code snippet from a backing bean shows how to create a SelectItems property:

```
import
javax.faces.model.SelectItem; ...
protected ArrayList options = null;
protected SelectItem newsletter0 =
new SelectItem
("200", "Duke's Quarterly", "");
...
```

```
//in constructor, populate the list
options.add(newsletter0);
options.add(newsletter1);
options.add(newsletter2); ...
public SelectItem getNewsletter0()
{ return newsletter0; }
void setNewsletter0
(SelectItem firstNL)
{ newsletter0 = firstNL; }
```

```
// Other SelectItem properties
public Collection[] getOptions()
{ return options; }
public void setOptions
(Collection[] options){
this.options =
new ArrayList(options);
}
```

The code first initializes `options` as a list.

Each newsletter property is defined with values.

Then each newsletter `SelectItem` is added to the list.

Finally, the code includes the obligatory `setOptions` and `getOptions` accessor methods.

Writing Properties Bound to Component Instances

A property bound to a component instance returns and accepts a component instance rather than a component value.

The following components bind a component instance to a backing bean property:

```
<h:selectBooleanCheckbox
id="fanClub" rendered="false"
binding="#{cashier.specialOffer}"
/>
<h:outputLabel for="fanClub"
rendered="false"
binding=
"#{cashier.specialOfferText}"
>
```

```
<h:outputText id="fanClubLabel"
value="#{bundle.DukeFanClub}"
/>
</h:outputLabel>
```

The `selectBooleanCheckbox` tag renders a check box and binds the `fanClub` `UISelectBoolean` component to the `specialOffer` property of `CashierBean`.

The `outputLabel` tag binds the `fanClubLabel` component, which represents the check box's label, to the `specialOfferText` property of `CashierBean`.

If the user orders more than $100 worth of items and clicks the Submit button, the `submit` method of `CashierBean` sets both components' `rendered` properties to `true`, causing the check box and label to display when the page is rerendered.

Because the components corresponding to the example tags are bound to the backing bean properties, these properties must match the components' types.

This means that the `specialOfferText` property must be of type `UIOutput`, and the `specialOffer` property must be of type `UISelectBoolean`:

```
UIOutput specialOfferText = null;
public UIOutput
getSpecialOfferText()
{ return this.specialOfferText; }
public void setSpecialOfferText
(UIOutput specialOfferText) {
this.specialOfferText =
specialOfferText;
}
```

```
UISelectBoolean specialOffer =
null;
public UISelectBoolean
getSpecialOffer()
{ return this.specialOffer; }
public void setSpecialOffer
(UISelectBoolean specialOffer)
{this.specialOffer = specialOffer;}
```

For more general information on component binding, see Backing Beans.

For information on how to reference a backing bean method that performs navigation when a button is clicked, see Referencing a Method That Performs Navigation. For more information on writing backing bean methods that handle navigation, see Writing a Method to Handle Navigation.

Writing Properties Bound to Converters, Listeners, or Validators

All the standard converter, listener, and validator tags included with JavaServer Faces technology support binding attributes that allow you to bind converter, listener, or validator implementations to backing bean properties.

The following example shows a standard `convertDateTime` tag using a value expression with its `binding` attribute to bind the `DateTimeConverter` instance to the `convertDate` property of `LoginBean`:

```
<h:inputText
value="#{LoginBean.birthDate}"
>
```

```
<f:convertDateTime
binding="#{LoginBean.convertDate}"
/>
</h:inputText>
```

The `convertDate` property must therefore accept and return a `DateTimeConverter` object, as shown here:

```
private DateTimeConverter
convertDate;
public DateTimeConverter
getConvertDate()
{... return convertDate; }
public void setConvertDate
(DateTimeConverter convertDate) {
convertDate.setPattern
("EEEEEEEE, MMM dd, yyyy");
this.convertDate = convertDate;
}
```

Because the converter is bound to a backing bean property, the backing bean property can modify the attributes of the converter or add new functionality to it.

In the case of the preceding example, the property sets the date pattern that the converter uses to parse the user's input into a `Date` object.

The backing bean properties that are bound to validator or listener implementations are written in the same way and have the same general purpose.

Writing Backing Bean Methods

Methods of a backing bean can perform several application-specific functions for components on the page.

These functions include

- Performing processing associated with navigation
- Handling action events
- Performing validation on the component's value
- Handling value-change events

By using a backing bean to perform these functions, you eliminate the need to implement the `Validator` interface to handle the validation or one of the listener interfaces to handle events.

Also, by using a backing bean instead of a `Validator` implementation to perform validation, you eliminate the need to create a custom tag for the `Validator` implementation.

In general, it's good practice to include these methods in the same backing bean that defines the properties for the components referencing these methods.

The reason for doing so is that the methods might need to access the component's data to determine how to handle the event or to perform the validation associated with the component.

The following sections explain how to write various types of backing bean methods.

Writing a Method to Handle Navigation

An action method, a backing bean method that handles navigation processing, must be a public method that takes no parameters and returns an `Object`, which is the logical outcome that the navigation system uses to determine the page to display next.

This method is referenced using the component tag's `action` attribute.

The following action method is from a backing bean named `CashierBean`, which is invoked when a user clicks the Submit button on the page.

If the user has ordered more than $100 worth of items, this method sets the `rendered` properties of the `fanClub` and `specialOffer` components to `true`, causing them to be displayed on the page the next time that page is rendered.

After setting the components' `rendered` properties to `true`, this method returns the logical outcome `null`.

This causes the JavaServer Faces implementation to rerender the page without creating a new view of the page, retaining the customer's input.

If this method were to return `purchase`, which is the logical outcome to use to advance to a payment page, the page would rerender without retaining the customer's input.

If the user does not purchase more than $100 worth of items, or if the `thankYou` component has already been rendered, the method returns `receipt`.

The JavaServer Faces implementation loads the page after this method returns:

```
public String submit() { ...
if(cart().getTotal() > 100.00 &&
!specialOffer.isRendered()){
specialOfferText.setRendered(true);
specialOffer.setRendered(true);
return null;
} else if
(specialOffer.isRendered() &&
!thankYou.isRendered()){
thankYou.setRendered(true);
return null;
```

```
} else {
clear();
return ("receipt");
}
}
```

Typically, an action method will return a `String` outcome, as shown in the previous example.

Alternatively, you can define an `Enum` class that encapsulates all possible outcome strings and then make an action method return an `enum` constant, which represents a particular `String` outcome defined by the `Enum` class.

The following example uses an `Enum` class to encapsulate all logical outcomes:

```
public enum Navigation {
main, accountHist, accountList,
atm, atmAck, transferFunds,
transferAck, error
}
```

When it returns an outcome, an action method uses the dot notation to reference the outcome from the Enum class:

```
public Object submit(){...
return Navigation.accountHist;
}
```

The section Referencing a Method That Performs Navigation explains how a component tag references this method.

The section Writing Properties Bound to Component Instances explains how to write the bean properties to which the components are bound.

Writing a Method to Handle an Action Event

A backing bean method that handles an action event must be a public method that accepts an action event and returns `void`.

This method is referenced using the component tag's `actionListener` attribute.

Only components that implement `javax.faces.component.ActionSource` can refer to this method.

In the following example, a method from a backing bean named `LocaleBean` processes the event of a user clicking one of the hyperlinks on the page:

```
public void chooseLocaleFromLink
(ActionEvent event) {
String current =
event.getComponent().getId();
FacesContext context =
FacesContext.getCurrentInstance();
context.getViewRoot().
setLocale((Locale)
locales.get(current));
}
```

This method gets the component that generated the event from the event object; then it gets the component's ID, which indicates a region of the world.

The method matches the ID against a `HashMap` object that contains the locales available for the application.

Finally, the method sets the locale by using the selected value from the `HashMap` object.

Referencing a Method That Handles an Action Event explains how a component tag references this method.

Writing a Method to Perform Validation

Instead of implementing the `Validator` interface to perform validation for a component, you can include a method in a backing bean to take care of validating input for the component.

A backing bean method that performs validation must accept a `FacesContext`, the component whose data must be validated, and the data to be validated, just as the `validate` method of the `Validator` interface does.

A component refers to the backing bean method by using its `validator` attribute.

Only values of `UIInput` components or values of components that extend `UIInput` can be validated.

Here is an example of a backing bean method that validates user input:

```
public void
validateEmail(FacesContext context,
UIComponent toValidate,
Object value) {
String message = "";
String email = (String) value;
if (!email.contains('@')) {
((UIInput)toValidate).
setValid(false);
```

```
message = CoffeeBreakBean.
loadErrorMessage(context,
CoffeeBreakBean.
CB_RESOURCE_BUNDLE_NAME,
"EMailError");
context.addMessage
(toValidate.getClientId(context),
new FacesMessage(message));
}
}
```

Take a closer look at the preceding code segment:

1. The `validateEmail` method first gets the local value of the component.

2. The method then checks whether the `@` character is contained in the value.

3. If not, the method sets the component's `valid` property to `false`.

4. The method then loads the error message and queues it onto the `FacesContext` instance, associating the message with the component ID.

See Referencing a Method That Performs Validation for information on how a component tag references this method.

Writing a Method to Handle a Value-Change Event

A backing bean that handles a value-change event must use a public method that accepts a value-change event and returns `void`.

This method is referenced using the component's `valueChangeListener` attribute.

This section explains how to write a backing bean method to replace the `ValueChangeListener` implementation.

The following example tag comes from Registering a Value-Change Listener on a Component, where the `h:inputText` tag with the `id` of `name` has a `ValueChangeListener` instance registered on it.

This `ValueChangeListener` instance handles the event of entering a value in the field corresponding to the component.

When the user enters a value, a value-change event is generated, and the `processValueChange(ValueChangeEvent)` method of the `ValueChangeListener` class is invoked:

```
<h:inputText id="name" size="50"
value="#{cashier.name}"
required="true">
<f:valueChangeListener
type="listeners.NameChanged"
/>
</h:inputText>
```

Instead of implementing `ValueChangeListener`, you can write a backing bean method to handle this event.

To do this, you move the `processValueChange(ValueChangeEvent)` method from the `ValueChangeListener` class, called `NameChanged`, to your backing bean.

Here is the backing bean method that processes the event of entering a value in the `name` field on the page:

```
public void processValueChange
(ValueChangeEvent event)
throws AbortProcessingException{
if (null != event.getNewValue()){
FacesContext.getCurrentInstance().
getExternalContext().
getSessionMap().
put("name", event.getNewValue());
}
}
```

To make this method handle the `ValueChangeEvent` generated by an input component, reference this method from the component tag's `valueChangeListener` attribute.

See Referencing a Method That Handles a Value-Change Event for more information.

Using Bean Validation

Validating input received from the user to maintain data integrity is an important part of application logic.

Validation of data can take place at different layers in even the simplest of applications, as shown in the `guessnumber` example application from an earlier chapter.

The `guessnumber` example application validates the user input (in the `h:inputText` tag) for numerical data at the presentation layer and for a valid range of numbers at the business layer.

JavaBeans Validation (Bean Validation) is a new validation model available as part of Java EE 6 platform.

The Bean Validation model is supported by constraints in the form of annotations placed on a field, method, or class of a JavaBeans component, such as a backing bean.

Constraints can be built in or user defined.

User-defined constraints are called custom constraints.

Several built-in constraints are available in the `javax.validation.constraints` package.

Table 9-2 lists all the built-in constraints.

Table 9-2 Built-In Bean Validation Constraints

Constraint	Description	Example
`@AssertFalse`	The value of the field or property must be false.	`@AssertFalse boolean isUnsupported;`
`@AssertTrue`	The value of the field or property must be	`@AssertTrue boolean isActive;`

	true.	
@DecimalMax	The value of the field or property must be a decimal value lower than or equal to the number in the value element.	`@DecimalMax("30.00")` `BigDecimal discount;`
@DecimalMin	The value of the field or property must be a decimal value greater than or equal to the number in the value element.	`@DecimalMin("5.00")` `BigDecimal discount;`

`@Digits`	The value of the field or property must be a number within a specified range.
	
	The `integer` element specifies the maximum integral digits for the number, and the `fraction` element specifies the maximum fractional digits for the number.
		`@Digits`
		`(integer=6,fraction=2)`
		`BigDecimal price;`

`@Future`	The value of the field or property must be a date in the future.	`@Future` `Date eventDate;`
`@Max`	The value of the field or property must be an integer value lower than or equal to the number in the value element.	`@Max(10)` `int quantity;`
`@Min`	The value of the field or property must be an integer value greater than or equal to the number in the value element.	`@Min(5)` `int quantity;`

`@NotNull`	The value of the field or property must not be null.	`@NotNull` `String username;`
`@Null`	The value of the field or property must be null.	`@Null` `String unusedString;`
`@Past`	The value of the field or property must be a date in the past.	`@Past` `Date birthday;`

`@Pattern`	The value of the field or property must match the regular expression defined in the `regexp` element.	`@Pattern(regexp="\\`
		`(\\d{3}\\)`
		`\\d{3}-\\d{4}")`
		`String phoneNumber;`

`@Size`	The size of the field or property is evaluated and must match the specified boundaries. If the field or property is a `String`, the size of the string is evaluated. If the field or property is a `Collection`, the size of the	`@Size(min=2, max=240)` `String briefMessage;`

`Collection` is evaluated.

If the field or property is a `Map`, the size of the `Map` is evaluated.

If the field or property is an array, the size of the array is evaluated.
Use one of the optional `max` or `min` elements to specify the boundaries.

In the following example, a constraint is placed on a field using the built-in @NotNull constraint:

```
public class Name {
@NotNull
private String firstname;
@NotNull
private String lastname;
}
```

You can also place more than one constraint on a single JavaBeans component object.

For example, you can place an additional constraint for size of field on the `firstname` and the `lastname` fields:

```
public class Name {
@NotNull
@Size(min=1, max=16)
private String firstname;
```

```
@NotNull
@Size(min=1, max=16)
private String lastname;
}
```

The following example shows a method with a user-defined constraint that checks for a predefined email address pattern such as a corporate email account:

```
@ValidEmail
```

```
public String getEmailAddress()
{  return emailAddress;  }
```

For a built-in constraint, a default implementation is available.

A user-defined or custom constraint needs a validation implementation.

In the above example, the `@ValidEmail` custom constraint needs an implementation class.

Any validation failures are gracefully handled and can be displayed by the `h:messages` tag.

Any backing bean that contains Bean Validation annotations automatically gets validation constraints placed on the fields on a JavaServer Faces application's web pages.

See Validating Persistent Fields and Properties for more information on using validation constraints.

Validating Null and Empty Strings

The Java programming language distinguishes between null and empty strings.

An empty string is a string instance of zero length, whereas a null string has no value at all.

An empty string is represented as "".

It is a character array of zero characters.

A null string is represented by `null`.

It can be described as the absence of a string instance.

Backing bean elements represented as a JavaServer Faces text component such as `inputText` are initialized with the value of the empty string by the JavaServer Faces implementation.

Validating these strings can be an issue when user input for such fields is not required.

Consider the following example, where the string `testString` is a bean variable that will be set using input typed by the user.

In this case, the user input for the field is not required.

```
if (testString.equals(null))
{ doSomething(); }
else
{ doAnotherThing(); }
```

By default, the `doAnotherThing` method is called even when the user enters no data, because the `testString` element has been initialized with the value of an empty string.

In order for the Bean Validation model to work as intended, you must set the context parameter `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` to `true` in the web deployment descriptor file, `web.xml`:

```
<context-param>
<param-name>
javax.faces.INTERPRET_EMPTY_STRING_
SUBMITTED_VALUES_AS_NULL
</param-name>
```

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

This parameter enables the JavaServer Faces implementation to treat empty strings as null.

Suppose, on the other hand, that you have a `@NotNull` constraint on an element, meaning that input is required.

In this case, an empty string will pass this validation constraint.

However, if you set the context parameter `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` to true, the value of the backing bean attribute is passed to the Bean Validation runtime as a null value, causing the `@NotNull` constraint to fail.