



# JavaServer Faces

Beatrice Amrhein



# Contents

---

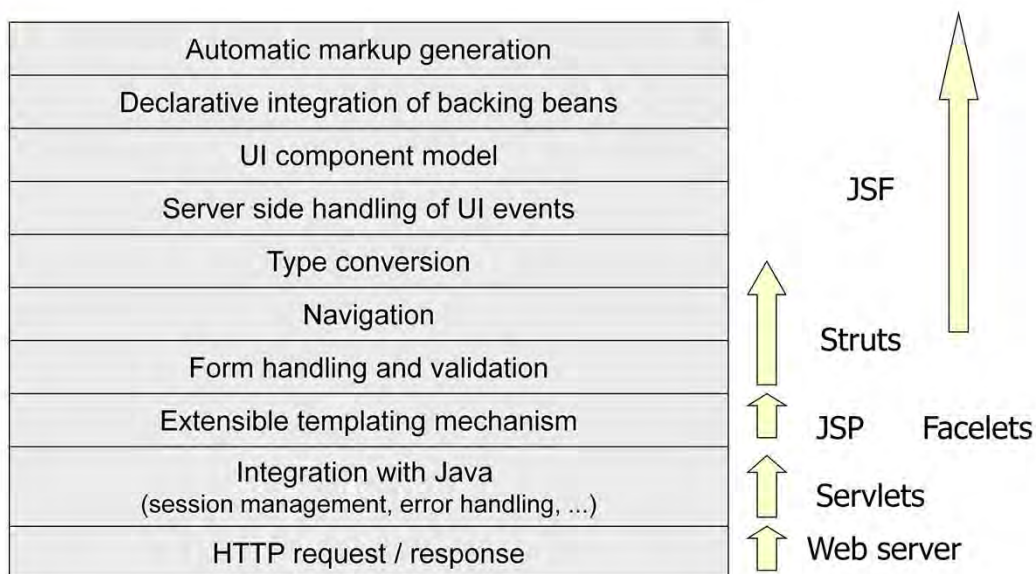
1 Introduction	3
2 Application Configuration	19
3 The Standard Components	35
4 Facelets Templating	53
5 Internationalization	67
6 Messaging	75
7 The Request Processing Lifecycle	91
8 Converters and Validators	99
9 Custom Tags	115
10 Composite Components	121
11 AJAX Support	143



# 1 Introduction



# Web Frameworks



The biggest advantage of the JSF technology is its flexible, extensible component model, which includes:

- An extensible component API for the usual standard components. Developers can also create their own components based on the JSF APIs, and many third parties have already done so and have made their component libraries publicly available (e.g. MyFaces: [www.myfaces.org](http://www.myfaces.org), RichFaces: [www.jboss.org/richfaces](http://www.jboss.org/richfaces), PrimeFaces: [primefaces.org](http://primefaces.org) or ICEFaces: [www.icesoft.org](http://www.icesoft.org)).
- A separate rendering model that defines how to render the components in various ways. For example, a component used for selecting an item from a list can be rendered as a menu or a set of radio buttons.
- An event model that defines how to handle events generated by activating a component, such as what to do when a user clicks a button or a hyper link.
- Automatic conversion and validation.

# JSF Scope

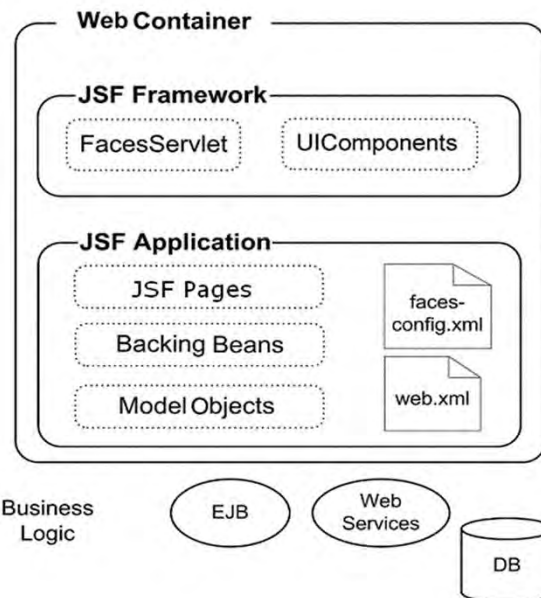
Client side



Client  
Devices



Server side



Web browsers don't know anything about JSF components or events. When a user clicks a button in a JSF application, it causes a request to be sent from your web browser to the server, or more precisely to the FacesServlet. JSF is responsible for translating that request into an event that can be processed by your application logic on the server (usually by the backing bean). JSF is also responsible that every UIComponent you have used on your pages is properly displayed on your browser.

JSF applications run on the server and can integrate with other subsystems like EJBs, web services or databases.

There is a central configuration file for JSF applications usually called `faces-config.xml`. Here, we can define the supported locales, the used backing beans, navigation rules, custom validators or converters and so on. In the new JSF2.0 standard many of these configurations are replaced by annotations in the Java code.



# A simple JSF Page

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

  <h:head>
    <title>Hello World</title>
    <h:outputStylesheet name="styles.css"/>
  </h:head>
  <h:body> <h:form id="main">
    <h:outputText value="Hello World" styleClass="header"/>
    <h:panelGrid columns="2">
      <h:outputLabel for="name" value="Your Name"/>
      <h:inputText id="name" value="#{helloBean.name}" required="true"/>
      <h:outputLabel for="age" value="Your Age"/>
      <h:inputText id="age" value="#{helloBean.age}" required="true"/>
    </h:panelGrid>
    <h:commandButton value="Say Hello" action="#{helloBean.sayHello}"/>
    <h:outputText value="#{helloBean.greeting}"/>
  </h:form></h:body>
</html>
```

## Hello World

Your Name	<input type="text" value="Felix"/>
Your Age	<input type="text" value="25"/>
<input type="button" value="Say Hello"/>	
Good Morning Felix!	

This is an example of a simple page with labels, input text fields and a submit button („Say Hello“).

For the web page designer, a JSF page looks similar to a normal HTML page.

We have to define the necessary namespaces (JSF core, JSF html, ...) so that the JSF tags can be used.

These pages can also be designed by an IDE like NetBeans, IntelliJ, Sun Java Studio Creator, IBM WebSphere or Oracle JDeveloper.



# Translation to HTML

## Hello World

Your Name

Your Age

Good Morning Felix!

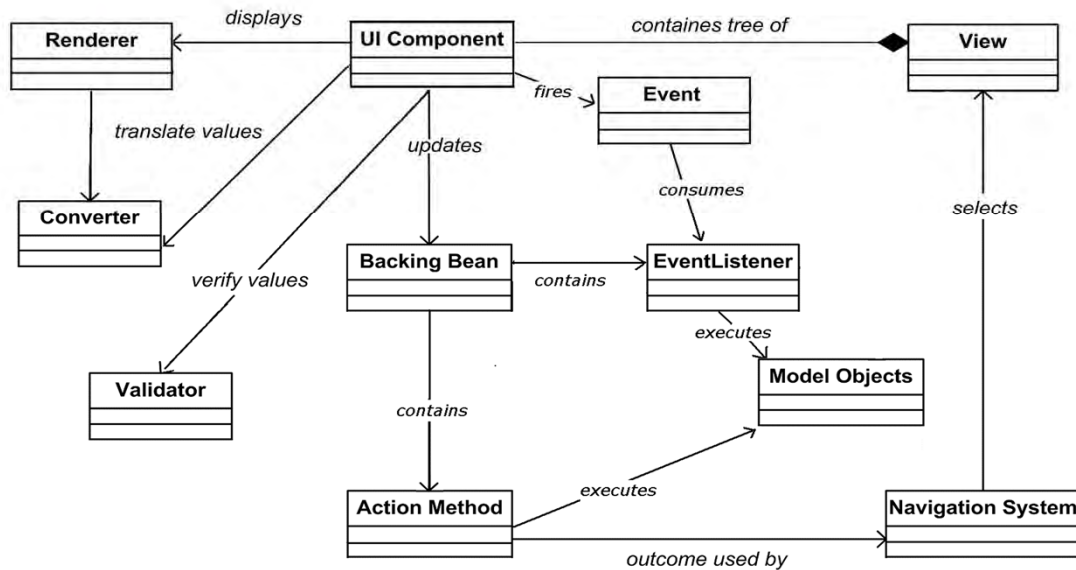
```
<html xmlns="http://www.w3.org/1999/xhtml">
<head> <title>Hello World</title>
  <link type="text/css" rel="stylesheet"
        href="/helloworld1/javax.faces.resource/styles.css.xhtml" /></head>
<body> <form id="main" name="main" method="post" action="/helloworld1/home.xhtml"
  enctype="application/x-www-form-urlencoded">
  <input type="hidden" name="main" value="main" />
  <span class="header">Hello World</span>
  <table> <tbody>
    <tr> <td><label for="main:name">Your Name</label></td> </tr>
    <tr> <td><label for="main:age">Your Age</label></td> </tr> </tbody> </table>
  <input type="submit" name="main:j_idt13" value="Say Hello" />
  <input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState"
    value="382741692:-980735" autocomplete="off" />
</form></body>
</html>
```

At runtime, the FacesServlet automatically translates the JSF page to HTML.

OutputText tags are converted to normal text, OutputLabel tags to HTML labels, inputText tags to a HTML input tag and the submit button to a HTML input tag with type „submit“.

Hidden input fields are used to transmit state information about the client or the server.

# JSF Concepts



This simplified UML class diagram shows the different concepts of JSF.

The view object contains a tree of UIComponents (output labels, command buttons, text fields, ...) that are displayed by the dedicated renderers. On user input, the backing beans are automatically updated. Converters translate and format the component's value and generate an error message if necessary. Validators verify the value of a component and generate an error message if necessary.

In general, JSF applications communicate by events. Backing beans contain event listeners and action methods. The outcome of an action method is used to specify the next page to be shown (navigation). Event listeners consume events and can execute model objects, which perform the core application logic.

In JSF applications, model objects don't know anything about the user interface. This enforces a MVC-style architecture.





# The JSF Expression Language

Access bean property or method

- `{myBean.myProperty}`
- `{myBean.myMethod}`

Access for array, list element or map entry

- `{myBean.myList[5]}`
- `{myBean.myMap['key']}`

The Expression Language uses the number sign (#) to mark the beginning of an expression.

EL expressions can be two way: they can be used to retrieve a property value or to update it.

EL expressions can also reference object methods.



# The JSF Expression Language

Boolean Expression     ==, !=, <=, ...

- `{myBean.myValue != 100 }`
- `{myBean.myValue <= 200 }`
- `{not empty myBean.myValue}`

## Usage

```
<h:inputText  
  value="{myBean.clientName}"  
  validator="{myBean.check}"  
  rendered="{not empty myBean.myValue}"/>
```



## Simple Example: Hello World

The screenshot shows a web browser window with the title 'Hello World'. The address bar displays 'http://js...jsf.html'. The page content includes the heading 'Hello World' in green, followed by two input fields: 'Your Name' with the value 'John' and 'Your Age' with the value '25'. Below these is a 'Say Hello' button. The output text 'Good Morning John!' is displayed below the button. At the bottom left of the page, the word 'Fertig' is visible.

We start with an easy but complete example, in order to learn to know all the key elements which belong to a JSF application.



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

  <h:head>
    <title>Hello World</title>
    <h:outputStylesheet name="styles.css"/>
  </h:head>
```

First, we look at the JSF page.

In the `<html>` tag, we declare the namespaces for the different tag libraries. These provide custom tags like text boxes, output labels and forms.

The JSF HTML elements usually render HTML elements, whereas the core tags are independent of a particular render kit (like `<f:view>`, `<f:validator>`, `<f:converter>` and so on).

In the `<h:head>` tag of the HTML page, we also define the stylesheet to be used (`<h:outputStylesheet ... >`).



## home.xhtml (2)

### Hello World

Your Name	<input type="text" value="John"/>
Your Age	<input type="text" value="25"/>
<input type="button" value="Say Hello"/>	
Good Morning John!	

```
<h:body>
  <h:form id="main">

    <h:outputText value="Hello World" styleClass="header"/>
    <h:panelGrid columns="2">
      <h:outputLabel for="name" value="Your Name"/>
      <h:inputText id="name" value="#{helloBean.name}"
        required="true"/>
      <h:outputLabel for="age" value="Your Age"/>
      <h:inputText id="age" value="#{helloBean.age}"
        required="true"/>
    </h:panelGrid>
```

The `<h:form>` tag represents an `HtmlForm` component, which is a container for other components and is used for posting information back to the server. We can have more than one `HtmlForm` on the same page, but all input controls must be nested within a `<h:form>` tag.

The `<h:outputText>` tag creates an `HtmlOutputText` component, which displays read-only text on the screen. This text can be literal or an EL expression (e.g. a backing bean property).

`<h:inputText>` creates a new `HtmlInputText` component that accepts text input.

`<h:panelGrid>` creates a HTML table. We will see the details of this tag in chapter 3.

The usual CSS styles can be used in JSF tags.



## home.xhtml (3)

### Hello World

Your Name	<input type="text" value="John"/>
Your Age	<input type="text" value="25"/>
<input type="button" value="Say Hello"/>	
Good Morning John!	

```
<p/>
<h:commandButton value="Say Hello"
                  action="#{helloBean.sayHello}"/>
<p/>
<h:outputText value="#{helloBean.greeting}"/>
</h:form>
</h:body>

</html>
```

The `<h:commandButton>` tag specifies an `HtmlCommandButton` component. The `value` attribute defines the button label, here "Say Hello". `HtmlCommandButtons` send action events to the application when they are clicked by a user. The `action` attribute references the action method that computes the new greeting text. The outcome of the action method is used to handle navigation.

Here the `<h:outputText>` tag displays the value of the `greeting` property of the `helloBean` backing bean.



# HelloBean.java

---

```
package jsf.examples;

import java.io.Serializable;
import javax.enterprise.context.SessionScoped;
import javax.inject.Named;

@Named("helloBean")
@SessionScoped
public class HelloBean implements Serializable {

    private String name;
    private int age;
    private String greeting;

    /** property methods for name and age */
    public int getAge() { return age; }

    public void setAge(int age) {
        this.age = age;
    }

    ...
}
```

We use an `@Named` annotation to declare this class as a managed bean. All managed beans are created and initialized by the *Managed Bean Creation Facility* the first time they are requested by an EL expression.

The `@Named` annotation has an optional name parameter to define the name used in the JSF page (e.g. `#{helloBean.name}`). It is good practice to choose similar names for the bean class and its object. If the name property is omitted, the system will take the unqualified Java class name (first letter in lowercase, here „helloBean“).

The `@SessionScoped` annotation controls the lifespan of the managed bean.

Managed beans are automatically instantiated and registered in their scope.

Getter and setter methods define properties and provide reading or writing access from the JSF page (e.g. via EL expressions).



## HelloBean.java

---

```
/** getter method for greeting*/
public String getGreeting() {
    return greeting;
}

/** Define the text to say hello. */
public String sayHello() {
    if (age < 11) {
        greeting = "Hello " + name + "!";
    } else {
        greeting = "Good Morning " + name + "!";
    }
    return null;
}
}
```

The sayHello() action method is called, whenever the „Say Hello“ button is pressed. It computes the new value of the greeting property. Its return value is used to define the next page to be displayed (navigation), a null value implies „no navigation“.





# web.xml

---

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" . . . >

<display-name>Hello World</display-name>
<description>Welcome to JavaServer Faces</description>

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

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

All JavaEE web applications are configured by a deployment descriptor file (**web.xml**).

JSF applications require that you specify the FacesServlet, which is the main servlet (front controller) of the application.

The servlet name is arbitrary, but the servlet class must be javax.faces.webapp.FacesServlet. The servlet mapping makes sure, that every request to a resource of the form \*.xhtml is sent to the FacesServlet.



```
<welcome-file-list>
  <welcome-file>home.xhtml</welcome-file>
</welcome-file-list>

<context-param>
  <param-name>javax.faces.PROJECT_STAGE</param-name>
  <param-value>Development</param-value>
</context-param>

</web-app>
```

The welcome file list is optional and defines the possible entry points of your JSF application.

The context parameter *ProjectStage* provides the options Production, Development, UnitTest and SystemTest.

At runtime, you can query the Application object for the configured value by calling `Application.getProjectStage()`.

If not explicitly configured, the default will be `ProjectStage.Production`.

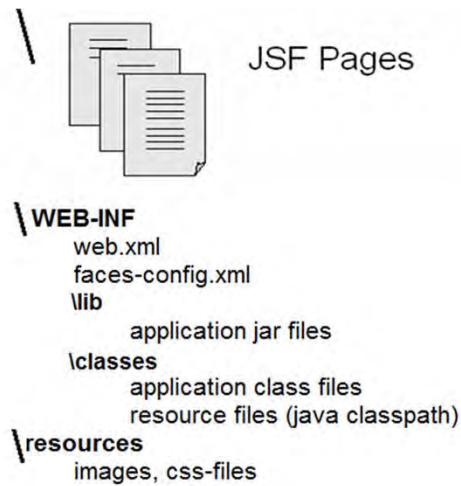
It's recommended to define the *Development* project stage during the development phase.



## 2 Application Configuration

# Directory Structure

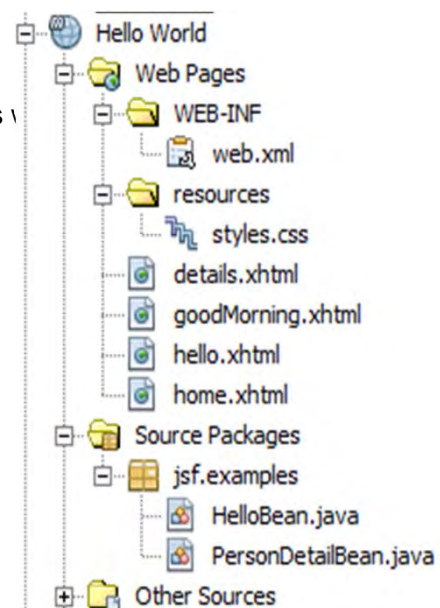
The standard Java web application directory structure.

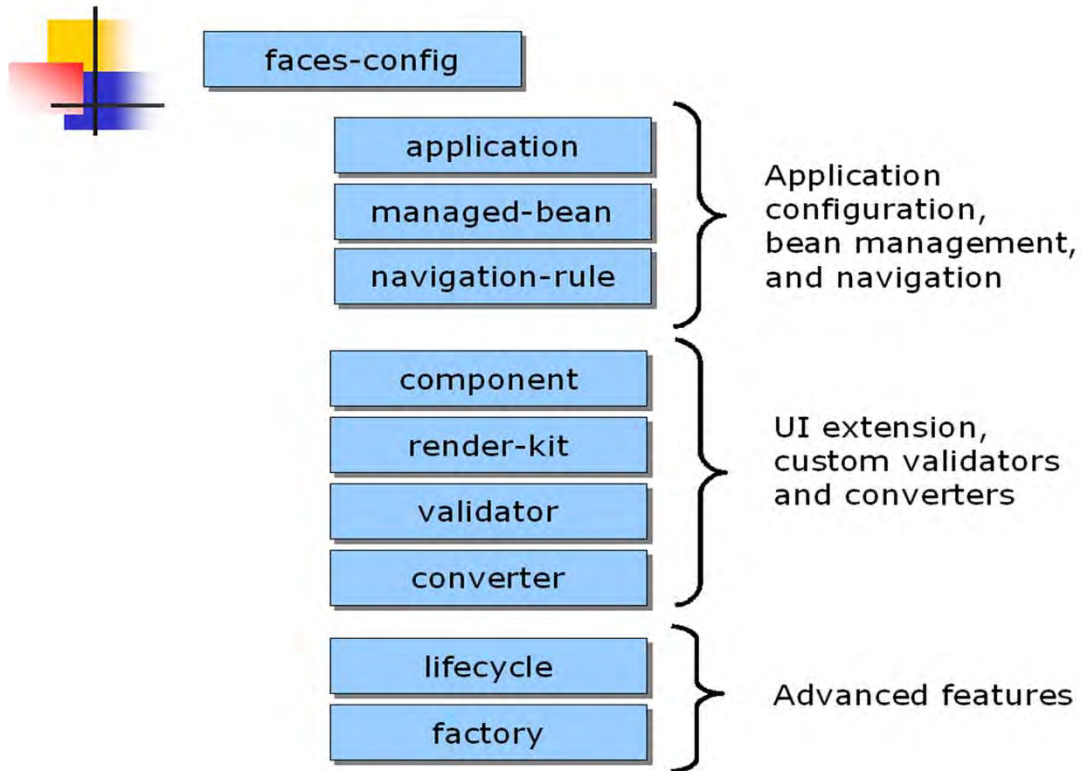


Because JSF applications are standard Java web applications, all of the necessary files must be packed in a directory structure that can be deployed to the web container.

For JSF applications, there are two additions: the faces-config.xml file (where certain JSF configurations are placed) and (if your web container doesn't already support JSF) the JAR files of your JSF implementation. The resources directory contains resources like images or css style sheets.

Netbeans builds the correct target structure, as long as the netbeans project.





The main configuration file for a JSF application is the **faces-config.xml** file.

All top level elements (`application`, `managed-bean`, ..., `factory`) are optional and can be included once or more than once.

The first group of elements is used for general application configuration, to declare the backing beans and for the navigation configuration.

The `application` element contains the supported locales and message resource bundle. Strings defined in this bundle can replace standard validation and conversion error messages.

The second group of elements are used to define custom behaviour (e.g. custom validators or converters).

The third group of elements is used to define advanced features like the registration of phase listeners.



## Configuration and Navigation Example





# home.xhtml

Your Name

```
<h:form id="main">
  <h:outputText value="Hello World" styleClass="header"/>
  <h:panelGrid columns="2">
    <h:outputLabel for="name" value="Your Name"/>
    <h:inputText id="name" value="#{helloBean.name}"
      required="true"/>
  </h:panelGrid>
</p>

<h:commandButton value="Details" action="details"/>
<h:commandButton value="Say Hello"
  action="#{helloBean.sayHello}"/>
</h:form>
```

We extend our first simple example by introducing two new features: navigation rules and managed properties.

The home.xhtml page obtains one additional button („Details“).

By pressing the „Details“ button, the user loads the persons detail.xhtml page, the „Say Hello“ button leads to the hello.xhtml or goodMorning.xhtml page, depending on the age of the person.



# details.xhtml

## Person Details

Your Age	<input type="text" value="25"/>
Your Country	<input type="text" value="Switzerland"/>
<input type="button" value="Home"/>	

```
<h:form id="main">
  <h:outputText value="Person Details" styleClass="header"/>
  <h:panelGrid columns="2">
    <h:outputLabel for="age" value="Your Age"/>
    <h:inputText id="age" value="#{personDetails.age}"
      required="true"/>
    <h:outputLabel for="country" value="Your Country"/>
    <h:inputText id="country"
      value="#{personDetails.country}"/>
  </h:panelGrid>
<p/>
<h:commandButton value="Home" action="home"/>
</h:form>
```

The details.xhtml page is used to change the age and the country property of the given person.

The initial value of the country property is read at creation time from the faces-config.xml file or from the backing bean.

The „Home“ button leads back to the home.xhtml page.





# hello.xhtml

---

Hello John from Switzerland!

[Home](#)

```
<h:form id="main">
  <h:outputText value="Hello #{helloBean.name}
    from #{personDetails.country}!"
    styleClass="header"/>
  <br/>
  <h:commandLink value="Home" action="home"/>
</h:form>
```

The hello.xhtml page is used to print out the greeting, that is the name property of the HelloBean backing bean and the country property of the personDetails Bean.

The „Home“ link leads us back to the home.xhtml page.



## goodMorning.xhtml

---

Good morning John from Switzerland!

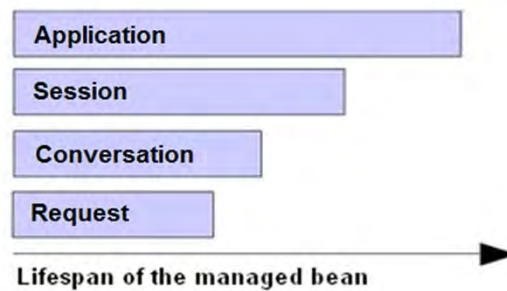
[Home](#)

```
<h:form id="main">
  <h:outputText value="Good morning #{helloBean.name}
    from #{personDetails.country}!" styleClass="header"/>
  <br/>
  <h:commandLink value="Home" action="home"/>
</h:form>
```

The goodMorning.xhtml page is almost the same page, except for the „Good morning“ text.

# HelloBean.java

```
@Named("helloBean")
@SessionScoped
public class HelloBean implements Serializable {
    private String name;
    private String helloText;
```



The `@{Request, Session, Application}-Scoped` annotation on the managed bean class defines how long the instance of the managed bean will be accessible to other parts of the program:

`@RequestScoped`: This managed bean is accessible for one HTTP request (default scope).

`@ConversationScoped`: The lifespan of this managed bean is user defined.

`@SessionScoped`: This managed bean lives as long as the user's session.

`@ApplicationScoped`: There is only one instance of this managed bean for the whole application.

Confer chapter 2.4.2 of

<http://jsfatwork.irian.at>



## HelloBean.java

---

```
/** The reference to the person details backing bean */
```

```
@Inject
```

```
private PersonDetailsBean personDetails;
```

```
@PostConstruct
```

```
private void init() { // custom initialization }
```

The `@Inject` annotation can be used to define a default value of a property. Here we use it to initialize the property with a reference to a `PersonDetailsBean` backing bean (dependency injection).

The `@PostConstruct` method is called after the constructor has finished and can be used to initialize certain values (e.g. from the database).

The `@PreDestroy` method is called at the end of the lifespan of this object.



## HelloBean.java

---

```
/** Define the text to say hello. */  
  
public String sayHello() {  
    if (personDetails.getAge() < 11) {  
        return "hello";  
    } else {  
        return "goodMorning";  
    }  
}
```

The sayHello() action method is called as soon as the „Say Hello“ button is pressed. The action method computes the helloText property and determines the next page to be displayed..

The returned „hello“ string leads to the hello.xhtml page.



## PersonDetailsBean.java

---

```
@Named("personDetails")
@SessionScoped
public class PersonDetailBean implements Serializable {

    private int age;
    private String country = "Switzerland";

    public int getAge() { return age; }

    public void setAge(int age) { this.age = age; }

    public String getCountry() {
        ...
    }
}
```



## XML configuration

```
<faces-config version="2.2"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee" . . . >
  <managed-bean>
    <managed-bean-name>
      helloBean</managed-bean-name>
    <managed-bean-class>
      jsf.examples.HelloBean</managed-bean-class>
    <managed-bean-scope>
      session</managed-bean-scope>
    <managed-property>
      <property-name>
        personDetails</property-name>
      <value>
        #{personDetails}</value>
      </managed-property>
    </managed-bean>
```

As an alternative to the `@Named`, `@SessionScoped` and `@Inject` annotation, we can also make a declaration in the `faces-config.xml` configuration file.

Here we define the class and the scope of the `helloBean` backing bean as well as the value of the `personDetails` property.



## XML configuration

```
<managed-bean>
  <managed-bean-name>
    personDetails</managed-bean-name>
  <managed-bean-class>
    jsf.examples.PersonDetailsBeans </managed-bean-class>
  <managed-bean-scope>
    session</managed-bean-scope>
  <managed-property>
    <property-name>
      country</property-name>
    <value>
      Switzerland</value>
    </managed-property>
  </managed-bean>
```

The personDetails property references a PersonDetails backing bean, equally in session scope.

The <value> element contains the default value of the property specified in the <property-name> element.

Here the country property has „Switzerland“ as its default value.

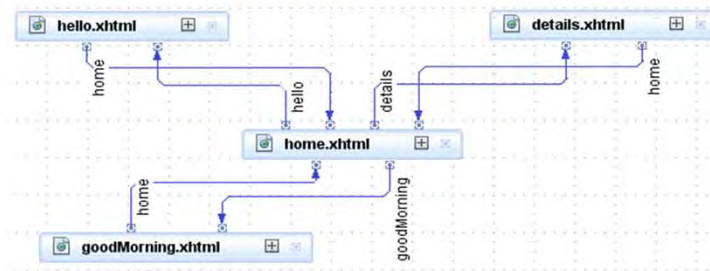
The Bean Creation Facility will apply the corresponding setter method to initialize the property with its default value (Property Injection: cf. Setter Injection pattern).

Collections (maps or lists) can only be initialized by <managed-property> elements in the Faces configuration file, but not by annotations in the Java code.





## faces-config.xml (navigation)



```
<navigation-rule>
  <from-view-id>/home.xhtml</from-view-id>
  <navigation-case>
    <from-action>#{helloBean.sayHello}</from-action>
    <from-outcome>hello</from-outcome>
    <to-view-id>/hello.xhtml</to-view-id>
  </navigation-case>
```

Navigation can be fully defined by the return values of the action methods. The only disadvantage of this approach is, that we have to know the name of the JSF pages in the backing beans.

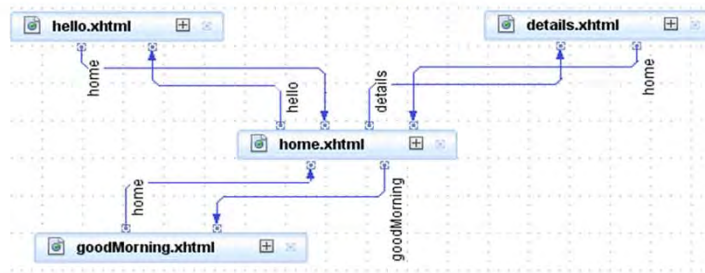
Therefore, we sometimes prefer to define the navigation rules in a faces-config.xml file. In this example, we see the navigation cases for the pages home.xhtml, details.xhtml, and hello.xhtml

By pressing the „Say Hello“ button, the sayHello action is processed which produces the outcome „hello“ or goodMorning. Therefore either the hello.xhtml or the goodMorning page will be loaded.

```
public String sayHello() {
    if (personDetails.getAge() < 11) {
        return "hello";
    } else {
        return "goodMorning";
    }
}
```



## faces-config.xml (navigation)



```
<navigation-case>
  <from-outcome>details</from-outcome>
  <to-view-id>/details.xhtml</to-view-id>
</navigation-case>
</navigation-rule>

<navigation-rule>
  <navigation-case>
    <from-outcome>home</from-outcome>
    <to-view-id>/home.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

Pressing the „Details“ button leads us to the details.xhtml page.  
From the details and hello page we can go back to the hello.xhtml page by pressing the „Home“ button or link.

The second navigation rule has no <from-view-id> element. This means, we navigate to the home.xhtml page whenever an action method returns „home“ (independend from the actual view).



## 3 The Standard Components



Chapter 3 of  
<http://jsfatwork.irian.at>



You find a detailed description of all standard components in chapter 3 of the book:

Martin Marinschek / Michael Kurz / Gerald Müllan JavaServer Faces 2.2  
Grundlagen und erweiterte Konzepte  
dpunkt.verlag  
<http://jsfatwork.irian.at>



## Shared Attributes

---

<code>id</code>	String	identifier
<code>label</code>	ValueExpression	name for this component (e.g. for error messages)
<code>value</code>	ValueExpression	the components value
<code>rendered</code>	ValueExpression	false suppresses rendering
<code>required</code>	ValueExpression	true requires a value to be entered into the input field
<code>styleClass</code>	ValueExpression	CSS class name

We start with the basic attributes, which are applicable to most of the components.

**id:** A string identifier for a component

**label:** A representable name for this component (e.g. for error messages)

**rendered:** A boolean value, if set to false suppresses rendering of this component

**value:** The components value, typically a value binding

**required:** A boolean value, if true a value is required for this input field

**styleClass:** The name of the CSS class which should be used for this component



# Layout Components

```
<h:panelGrid columns="2" border="1"
    headerClass="page-header" cellpadding="1" width="40%">
    <f:facet name="header">
        <h:outputText value="This is the table header"/>
    </f:facet>
    <h:outputText value="(1,1)"/>
    <h:outputText value="(1,2)"/>
    <h:outputText value="(2,1)"/>
    <h:outputText value="(2,2)"/>
</h:panelGrid>
```

This is the table header	
(1,1)	(1,2)
(2,1)	(2,2)

HtmlPanelGrid is useful for creating arbitrary, static component layouts (it maps to the `<table>` element). You can also configure header and footer with facets that map to the `<thead>` and `<tfoot>` table subelements, respectively.

Child components are organized according to the specified number of columns. When we specify two columns, the first two components form the first row (one per column), the next two form the second row, and so on. The width, border, and cellpadding properties are passed through to the HTML table. Unlike the HTML table, you don't have to explicitly denote columns and rows.



# Grouping of Components

```
<h:panelGrid columns="2" . . . . >
  <h:outputText value="(1,1)"/>
  <h:panelGroup>
    <h:outputText value="(1,2.1)"/>
    <h:outputText value="(1,2.2)"/>
  </h:panelGroup>
  <h:panelGroup>
    <h:outputText value="(2.1,1)"/>
    <h:outputText value="(2.2,1)"/>
  </h:panelGroup>
  <h:outputText value="(2,2)"/>
</h:panelGrid>
```

(1,1)	(1,2.1) (1,2.2)
(2.1,1) (2.2,1)	(2,2)

The `HtmlPanelGroup` component groups a set of components together, so that they can be treated as a single entity.



# Output Components

```
<h:outputLabel for="age" value="Your Name:"/>
```



```
<h:outputText value="#{helloBean.helloText}"  
              styleClass="header"/>
```

Good Morning John!

The purpose of the output components is to display data. You can specify the data that they display literally or define properties to be displayed from backing beans (by using value-binding expressions).

The most basic output component is `HtmlOutputText` (`<h:outputText>`).

`HtmlOutputText` converts the given value to a string and displays it with optional CSS style support.

`HTMLOutputLabel` produces a HTML label, which can be used as a label of an input component.





# Input Components

---

```
<h:inputText value="#{helloBean.name}" size="30" required="true"/>
```

```
<h:inputText value="#{helloBean.age}" size="4" disabled="true"/>
```

Your Name:

Your Age:

The `HtmlInputText` component is used for basic user input. It maps to a simple text field—the HTML `<input>` element with type “text”.

The `size` and `disabled` properties are passed through to the HTML input tag.

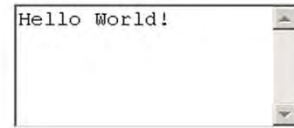
In this example, the component is associated via a value-binding expression with the „name“ property of `helloBean`.

`HtmlInputText` is the most common input control. It displays a single input text field.

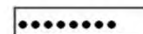


## Further Input Components

```
<h:inputTextarea value="#{helloBean.helloText}" rows="5" />
```



```
<h:inputSecret value="#{user.password}" size="10"/>
```



```
<h:selectBooleanCheckbox value="#{helloBean.accepted}"/>
```

Accepted ☒

If you need a multi line input field, you can use the `HtmlInputTextarea` component.

The `HtmlInputSecret` component is used to display a password input field. Any text the user types is displayed using the asterisk or some other character (depending on the browser).

For the input of boolean values, we can use a `HtmlSelectBooleanCheckbox` component. The „accepted“ property should therefore be a boolean type.



# Command Components

```
<h:commandButton value="Say Hello" action="#{helloBean.sayHello}"/>
```

```
<h:commandButton value="Goodbye"  
    action="#{helloBean.sayGoodbye}" immediate="true"/>
```



```
<h:commandLink value="Goodbye" action="#{personDetails.select}"/>
```

```
<h:outputLink value="http://jsf.net">  
    <h:outputText value="to the jsf page"/> to the jsf page  
</h:outputLink/>
```

```
<h:link value="Goodbye" outcome="select">
```

The command components represent an action initiated by the user. The two standard command components are `CommandButton` (for displaying a button) and `CommandLink` (for displaying a hyperlink).

As they perform the same operation, their usage is quite similar. They both trigger an action via a POST request and must therefore be embedded in a HTML-form.

`OutputLink` creates a simple HTML link. It has no action attribute, so no action method can be called and no JSF navigation is used. It can be used to leave the JSF application. Here we use an `<h:outputText>` child, but we can use any HTML (or JSF) tag.

The `<h:link>` tag has a value (which is rendered as the anchor text) and an outcome attribute, used for the JSF navigation without action method. It creates a HTTP GET request.



## Choice Input Components

```
<h:selectOneMenu id="country" value="#{helloBean.country}">
  <f:selectItem itemValue="FR" itemLabel="France" />
  <f:selectItem itemValue="CH" itemLabel="Switzerland" />
  <f:selectItem itemValue="DE" itemLabel="Germany" />
  <f:selectItem itemValue="IT" itemLabel="Italy" />
</h:selectOneMenu>
```

Your Country

Your Country   
France  
Switzerland  
Germany  
Italy

User interfaces often allow a user to select one or more items from a set of possible choices.

In JSF, a selection item represents a single choice, and has a value and a label. Components in the SelectMany and SelectOne families, like `HtmlSelectManyCheckbox` and `HtmlSelectOneListbox`, display lists of items.

A `UISelectItem` component is used to display items in a list and to select one or more items from this list.

The value of the `selectOneMenu` contains the selected item, whereas the value of `selectItems` contains the list of selectable items.



## Choice Input Components

```
<h:selectOneMenu id="country" value="#{personDetails.country}">
    <f:selectItems value="#{personDetails.countries}"
        var="country" itemValue="#{country.code}"
        itemLabel="#{country.name}" />
</h:selectOneMenu>
```

Your Country

Your Country   
France  
Switzerland  
Germany  
Italy

Usually, the values of a selection list come from a database or from the backing bean. In this case, we iterate through the appropriate elements from the backing bean to create the item elements of the list.



## The Backing Bean (1)

```
public enum Country {  
    CH("Switzerland"), DE("Germany"), FR("France"), IT("Italy");  
    private String value;  
    private Country(String value) { this.value = value; }  
    public String getCode() { return name(); }  
    public String getName() { return value; }  
}  
  
// provide the list of country objects  
public List<Country> getCountries() {  
    return Arrays.asList(Country.values());  
}
```

In the backing bean, you have to provide the corresponding values, i.e. the corresponding country objects for the selection list.



## The Backing Bean(2)

---

```
// get the selected country
public Country getCountry() {
    return country;
}

// set the selected country
public void setCountry(Country country) {
    this.country = country;
}
```

Then, we have to provide the corresponding get and set methods as well as the default value for the selected country.





# Data Tables

```
<h:dataTable id="table" value="#{personDetails.friends}"
    var="friend" headerClass="friendsHeader">
```

```
    <h:column>
        <f:facet name="header">
            <h:outputText value="Name"/>
        </f:facet>
        <h:outputText value="#{friend.name}"/>
    </h:column>
```

Name	Profession
Peter	Pilot
Alice	Cook
George	Philosopher
Julia	Actress

```
    <h:column>
        <f:facet name="header">
            <h:outputText value="Profession"/>
        </f:facet>
        <h:outputText value="#{friend.profession}"/>
    </h:column>
</h:dataTable>
```

The `HtmlDataTable` component displays an HTML `<table>`.

The `value` attribute defines an array or a list of Java beans whose properties are displayed one by one in the rows of the table.

The columns of the table are specified with column components. For each row, the column components are used as template for the different columns.

If the facet tag has a `name="header"` attribute, it is translated to a `<thead>` (table header) HTML tag.

The `var` property defines the name of the index variable (here `friend`).





## Friend.java

```
public class Friend {  
    String name;  
    String profession;  
  
    public Friend (String name, String profession) {  
        this.name = name;  
        this.profession = profession;  
    }  
  
    public String getName(){  
        return name;  
    }  
  
    public void setName(String name){  
        this.name = name;  
    }  
  
    ...  
}
```

Name	Profession
Peter	Pilot
Alice	Cook
George	Philosopher
Julia	Actress

In the backing bean, we have to provide a corresponding list of friends with a name and a profession property.



## Row Selection

```
<h:dataTable value="#{personDetails.friends}" var="friend"
              styleClass="friendsTable" >

  <h:column>
    <f:facet name="header">
      <h:outputText value="Name"/>
    </f:facet>
    <h:commandLink
      action="#{personDetails.selectBestFriend(friend)}"
      value="#{friend.name}" />
    </h:column>
    ...

  /* action method in personDetails managed bean */
  public String selectBestFriend(Friend friend) {
    bestFriend = friend;
    return "details";
  }
```

Name	Profession
<a href="#">Peter</a>	Pilot
<a href="#">Alice</a>	Cook
<a href="#">George</a>	Philosopher
<a href="#">Julia</a>	Actress

Your Best Friend: George

In JSF 2.0, we can invoke methods with parameters in the EL expression. This is usable, for example, to select an item in a table (to show a persons details).



# Images and Styles

---

Include stylesheet

```
<h:head>  
  <title>Hello World</title>  
  <h:outputStylesheet library ="styles" name="styles.css"/>  
</h:head>
```

Include image

```
<h:graphicImage library ="images" name ="icon.png" />
```

The images and stylesheets are stored in the resources directory (cf. page 20)





## 4 Facelets Templating

Confer chapter 4.3 of  
<http://jsfatwork.irian.at>



## Why Templating?

---

- Avoiding repetition in JSF pages
- Support for standard look and feel

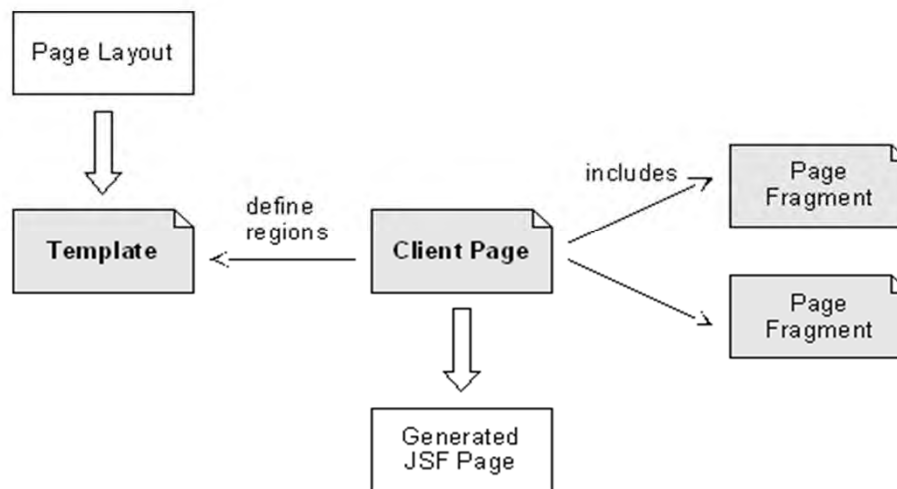
Templating is a useful Facelets feature that allows you to create a page that will act as the base (or template) for other pages in an application.

By using templates, you can reuse code and avoid recreating similarly constructed pages.

Templating also helps in maintaining a standard look and feel in an application with a large number of pages.



# JSF Templating Overview



A Facelets template is an XHTML document that defines the general layout of multiple JSF pages, the so-called client pages of the template. For this, the template contains variable regions (placeholders) which are defined by the client pages, either internally or by including external page fragments. The JSF framework then generates JSF pages by replacing the variable regions by the concrete contents specified by the client pages.



## Basic Steps

---

- Define a **page template**
  - define page layout
  - write static content
  - specify exchangeable content
- Define a **client page** of the template
  - specify the template
  - override content for each replaceable region

The static content that will appear in all client pages has to be entered into the template. Content that can (or will) be replaced in the client pages is marked with **ui:insert** (with default values for the clients pages that do not supply content).

The client page specifies the template it is using by **ui:composition**.

With **ui:define**, the content for each replaceable region in the template is replaced..



# Templating Pages

Template (**template.xhtml**)

```
<html>
  <head>...</head>
  <body>
    <ui:insert name="header"/>
    <ui:insert name="content"/>
  </body>
</html>
```

Client Page (**page.xhtml**)

```
<ui:composition
  template="template.xhtml">
  <ui:define name="header">
    <ui:include src="header.xhtml"/>
  </ui:define>
  <ui:define name="content">
    <ui:include src="content.xhtml"/>
  </ui:define>
</ui:composition>
```

Page Fragment (**header.xhtml**)

```
<ui:composition>
  <h:outputText .../>
</ui:composition>
```

Page (**content.xhtml**)

```
<html>
  <head>...</head>
  <body>
    <ui:composition>
      <h:form>
        ...
      </h:form>
    </ui:composition>
  </body>
</html>
```



# Templating Example

## Hello World

Your Name

[Details](#)

[Say Hello](#)

## Person Details

Your Age

Your Country

Name	Profession
<a href="#">Peter</a>	Pilot
<a href="#">Alice</a>	Cook
<a href="#">George</a>	Philosopher
<a href="#">Julia</a>	Actress

Your Best Friend: George

[Home](#)

Good morning John from Switzerland!

[Home](#)



# The Template

```
<html xmlns:ui="http://xmlns.jcp.org/jsf/facelets" . . . >

static content {
  <h:head>
    <title>Hello World</title>
    <h:outputStylesheet name="styles.css" />
  </h:head>

  <h:body>
    <ui:insert name="header">
      Hello World
    </ui:insert>
    <ui:insert name="content" />
  </h:body>
</html>
```

exchangeable content

} default value

Our template defines two regions, header and content, that may be replaced by client pages.

The static content in the `<h:head>` element is used for all client pages that use this template.

The header region (`<ui:insert name="header">`) provides a child element which is used as the default content if the client page defines no header.

# A Client Page

```
<ui:composition template="template.xhtml"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

    <ui:define name="header">
        <ui:include src="header.xhtml">
            <ui:param name="greeting" value="Hello World" />
        </ui:include>
    </ui:define>

    <ui:define name="content">
        <ui:include src="home_content.xhtml" />
    </ui:define>

</ui:composition>
```



The home.xhtml client page uses the given template and includes the content for the header and content regions (e.g. `<ui:include src="header.xhtml">`). The `<ui:param>` child element is used to pass a parameter value to the header.xhtml page fragment.



## The Hello Client Page

```
<ui:composition template="template.xhtml"
  xmlns:ui="http://xmlns.jcp.org/jsf/facelets" xmlns:h="http://xmlns.jcp.org/jsf/html">

  <ui:define name="header">
    <ui:include src="header.xhtml">
      <ui:param name="greeting" value="Hello
        #{helloBean.name} from
          #{personDetails.country.name}!"/>
    </ui:include>
  </ui:define>

  inline definition { <ui:define name="content">
    <h:form id="main">
      <h:commandLink value="Home" action="home"/>
    </h:form>
  </ui:define>
  </ui:composition>

  header { Hello John from Switzerland!
  content { Home
```

The goodMorning.xhtml client page uses the same template and includes the content of the header region.

The content region is statically defined as inline code.

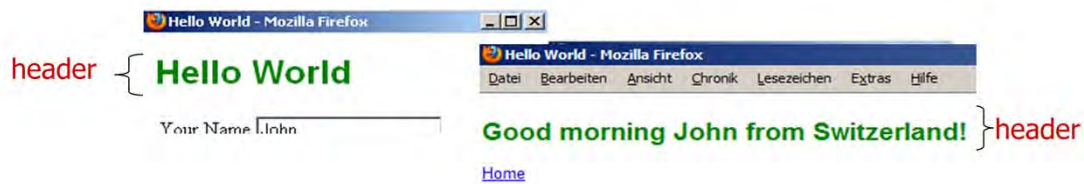


## Header Page Fragment

```
<ui:composition xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
                 xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:outputText value="#{greeting}" styleClass="header" />

</ui:composition>
```



The header.xhtml page fragment is included by the home.xhtml as well as by the goodMorning and the hello.xhtml client page.

The parameter greeting is either set to „Hello World“ or to „Good Morning“ or „Hello“ plus the person name, according to the value in the <ui:param> tag.



## Home Content Fragment

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
  <h:head>
    <title>Hello World</title>
    <h:outputStylesheet name="styles.css" />
  </h:head>
  <h:body>
    <ui:composition>
      ...
    </ui:composition>
  </h:body>
</html>
```

home content

Your Name

The home\_content.xhtml fragment page is inserted into the content region of the home.xhtml client page.

It is a self-contained web page. Only the content of <ui:composition> element is inserted into the <ui:define> region of the home.xhtml page. All contents outside of the <ui:composition> tag is ignored.



## <ui:composition>

```
<h:form id="main">
  <h:panelGrid columns="3">
    <h:outputLabel for="name" value="Your Name" />
    <h:inputText id="name" value="#{helloBean.name}"
      required="true"/>
    ...
  </h:panelGrid>
  <h:commandButton value="Details" action="details"/>
  <h:commandButton value="Say Hello"
    action="#{helloBean.sayHello}"/>
</h:form>
</ui:composition>
```





## Facelets Templating Tags

### **ui:composition**

a reusable piece of markup, that may be included by ui:insert and may use a template attribute. Ignores all content outside of this tag.

### **ui:define**

defines content that is inserted into a page by a template.

### **ui:include**

used in a ui:define tag to insert the content of a page (or page fragment)

### **ui:insert**

inserts content into a template.

### **ui:param**

is used to pass parameters to an included file.

Some further facelets templating tags are:

#### **<ui:component>**

Similar to <ui:composition>. Defines a new component that is created and added to the component tree. Has no template attribute.

#### **<ui:decorate>**

Similar to the <ui:composition> tag. Content outside of this tag is not ignored.

#### **<ui:fragment>**

The fragment tag is identical to the <ui:component> tag, but does not ignore the content outside of the tag.

See also

<https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/>





# 5 Internationalization

## Support for Multiple Locales

### Internationalization (I18N)

means planning and implementing products and services so that they can easily be adapted to specific local languages and cultures

### Localization

is the process of adapting a product or service to a particular (or new) language, culture, or local "look-and-feel."



# Configuring Locales

faces-config.xml: Configuration of supported locales

```
<faces-config . . . >
  <application>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>de</supported-locale>
      <supported-locale>fr</supported-locale>
    </locale-config>
  </application>
  . . .
</faces-config>
```

You have to tell your JSF application which language it should support. The supported locales are specified with the `<locale-config>` element in the Faces configuration file, inside the `<application>` element.

The element `<default-locale>` specifies the default locale which is used if the current locale is not found. With the following `<supported-locale>` elements you define the locales that are supported by your web application.

# Configuring Resource Bundles

faces-config.xml: Definition of the resource bundle

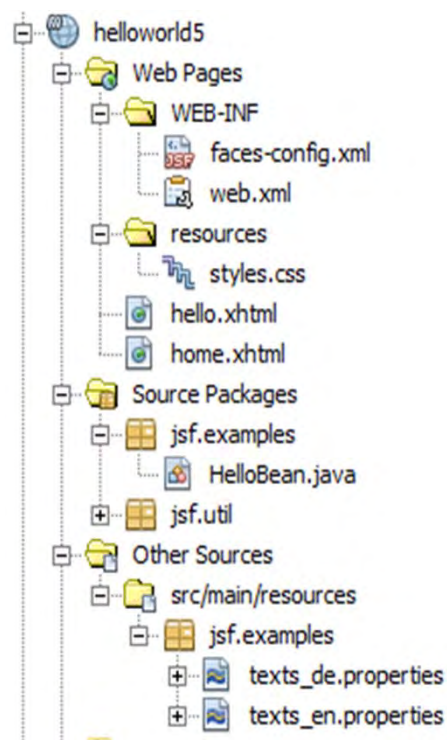
```
<faces-config . . . >
  <application>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>de</supported-locale>
      <supported-locale>fr</supported-locale>
    </locale-config>
    <resource-bundle>
      <base-name>jsf.examples.texts</base-name>
      <var>texts</var>
    </resource-bundle>
  </application>
```

Resource bundles contain the translated texts which are used in the JSF pages or the application code.

They are configured in the faces-config.xml file.

The resource bundle files of the supported locales are read on demand.

Here we declare the texts\_xx.properties files in the directory jsf/examples.





# Creating Resource Bundles

texts\_de.properties

helloWorld=**Hallo Welt**

yourName=**Ihr Name**

yourAge=**Ihr Alter**

sayHello=**Sag Hallo**

hello=**Hallo {0}!**

goodMorning=**Guten Tag {0}!**

home=**Startseite**

**Hallo Welt**

Ihr Name

Ihr Alter

Resource bundles are not a JSF feature, they are a fundamental part of the way Java handles internationalization and localization.

Resource bundles are normal property files with key/value pairs. Each text has a key, which is the same for all locales. So the key „helloWorld“ points to the text that represents the welcome message, whether it is in English, French, or German.



# Creating Resource Bundle

texts\_en.properties

```
helloWorld=Hello World  
yourName=Your Name  
yourAge=Your Age  
sayHello=Say Hello
```

**Hello World**

Your Name   
Your Age

```
hello=Hello {0}  
goodMorning=Good Morning {0}  
home=Home
```

As these keys are used in EL expressions, special characters like the number sign (#), slashes (/), points (.), brackets, spaces, and so on are not allowed.



# Using Resource Bundles

```
<title>#{texts.helloWorld}</title>
```

```
<h:outputLabel for="name" value="#{texts.yourName}"/>
<h:inputText id="name" value="#{helloBean.name}" />
<h:outputLabel for="age" value="#{texts.yourAge}"/>
<h:inputText id="age" value="#{helloBean.age}" />
```

```
<h:outputFormat value="#{texts.goodMorning}"
                styleClass="header">
  <f:param value="#{helloBean.name}"/>
</h:outputFormat>
```

Hallo Welt - Mozilla Firefox

Datei Bearbeiten Ansicht

Hallo Welt

Ihr Name

Ihr Alter

Guten Tag John!

You can refer to any of the defined keys using an EL expression using the variable defined in the faces-config.xml file for the corresponding resource bundle (here „texts“).





## 6 Messaging

Confer chapter 2.13 of  
<http://jsfatwork.irian.at>



## Messages

---

- ... are generated from different sources.
- ... have a severity level (info, warn, error, fatal)
- ... contain a summary and a detail text
- ... can be global or associated with a component
- ... are stored in the Faces context
- ... have request scope

Multiple parts of a JSF application (validators, converters, application exceptions ... ) can generate error or information messages.

The detail text of a message is optional, only a summary text is required. If the detail text is missing, the summary text is printed out instead.

# Displaying Messages

Messages for input controls name and age

```
<h:message for="name" />
<h:message for="age" showDetail="false" showSummary="true"
    errorClass="error" infoClass="info"/>
```

All messages for this page.

```
<h:messages errorClass="errors" layout="table"/>
```



A message associated to a UI component can be displayed with the `<h:message>` tag where the *for* attribute specifies the id of the component.

Whenever possible, these messages should be placed next to the component, so the user knows where the problem arises.

If more than one message is registered for a component `<h:message>` displays only the first one.

`<h:messages>` displays all messages of this page.

The attributes *showDetail* and *showSummary* are optional. In `<h:message>` the default value for *showDetail* is true, that for *showSummary* is false. In `<h:messages>` the default value for *showDetails* is false, that for *showSummary* is true.

You can insert an *infoClass*, *infoStyle*, *warnClass*, *warnStyle*, ... attribute to define a CSS class or style for messages with the according severity.

The *layout* attribute defines how the messages are displayed. Possible values for layout are *list* or *table*, the default value is list.



# Overriding Standard Messages

In the input component tag

```
<h:inputText id="name" value="#{helloBean.name}" required="true"
    requiredMessage="Please enter your name">
<h:message for="name" errorClass="error"/>
```

Hello World

Your Name  Please enter your name

- requiredMessage for required input
- converterMessage for conversion errors
- validatorMessage for validation errors

All of the standard validators and converters have default messages. These are configured in the default application message bundle. In the reference implementation, for example, we can find the following key/value pair, creating a message for an empty input field with a required="true" attribute.

```
javax.faces.component.UIInput.REQUIRED={0}: Validation Error: Value is required.
```

The different parameters are defined in the JSF documentation for the corresponding messages. If there is only one parameter, the label name is assigned to parameter 0.

These default messages may be missing or not very user friendly, so we would like to override them with custom messages.

One possibility is to define an error message in the required-, converter-, or validatorMessage attribute of the corresponding input component.



## Overriding Standard Messages

Or with texts from the resource bundles.

```
<h:inputText id="name" label="#{texts.yourName}"
    value="#{helloBean.name}" required="true"
    requiredMessage="#{texts.enterYourName}" />
<h:message for="name" errorClass="errorMessage"/>
```

Hello World

Your Name  Please enter your name

In order to provide localized messages, the texts for the `requiredMessage`, `converterMessage` or `validatorMessage` can be defined in a resource bundle (here `texts_xx.properties`).



# Overriding Standard Messages

... by a custom message bundle

```
<faces-config ... >
  <application>
    <message-bundle>jsf.examples.messages</message-bundle>
  </application>
```

Another possibility to override the standard messages is to configure a custom message bundle in the faces-config.xml file.

Here, we define the bundle messages\_xx.properties in the directory jsf/examples.

When a JSF component, validator or converter is looking for an error message, it first looks for it in the JSF page and then in the custom message bundle (here messages\_xx.properties).

Only if there is no user defined message, the predefined standard message is used.



## Using Localized Messages

messages\_de.properties

javax.faces.converter.IntegerConverter.INTEGER=

Der eingegebene Wert "{0}" ist keine Zahl.

javax.faces.converter.IntegerConverter.INTEGER\_detail=

Bitte geben Sie eine ganze Zahl von der Form 17 ein.

messages\_en.properties

javax.faces.converter.IntegerConverter.INTEGER=

The given input value "{0}" is not an integer.

javax.faces.converter.IntegerConverter.INTEGER\_detail= . . .

Ihr Name	<input type="text" value="Anton"/>	
Ihr Alter	<input type="text" value="a"/>	Der eingegebene Wert 'a' ist keine Zahl.
Your Name	<input type="text" value="John"/>	
Your Age	<input type="text" value="a"/>	The given input value 'a' is not an integer.

With a custom message bundle, you can selectively override the standard messages, as long as you use the correct keys (see the constant-values.html page of the JSF API documentation for the list of message keys).

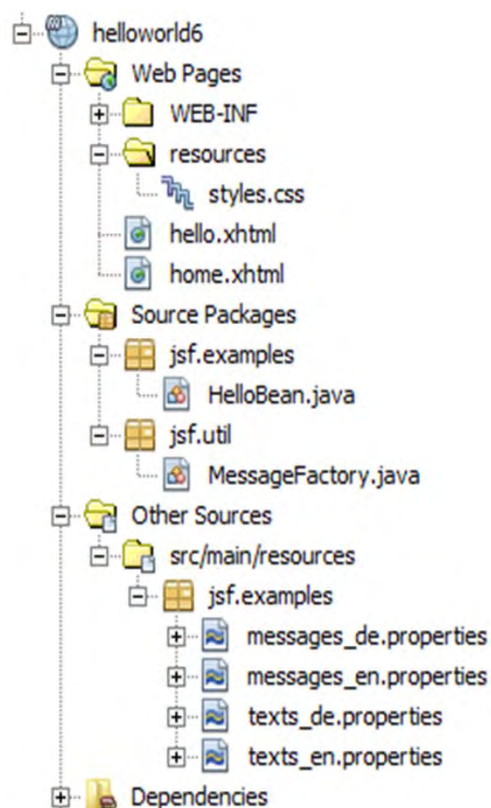
The detail text of a message is (by convention) stored at the same key as the summary text, but with an \_detail extension.

# Directory Structure



Resource files like messages (or texts for localized applications) are deployed to (a subdirectory of) the classes directory.

In our project directory, we put the necessary \*.properties files in the resource directory.







# Application Messages

---

Warnings or error messages for

- Backing Beans Exceptions
- Application Logic Exceptions
- System Errors
- Connection Errors
- . . .

Whenever one of your backing beans throws an exception or when an error occurs in your application logic, you have to inform the users about this.

For this, you create a `FacesMessage` in your backing bean (or validator or converter class).



## FacesMessage in BackingBean

```
public String sayHello() {  
    if (age < 1 || age > 120) {  
        // create hello message and add it to the Faces context  
        FacesMessage message =  
            new FacesMessage(FacesMessage.SEVERITY_ERROR,  
                "The given age must be between 1 and 120");  
        FacesContext context = FacesContext.getCurrentInstance();  
        context.addMessage(null, message);  
    } else { . . . }
```

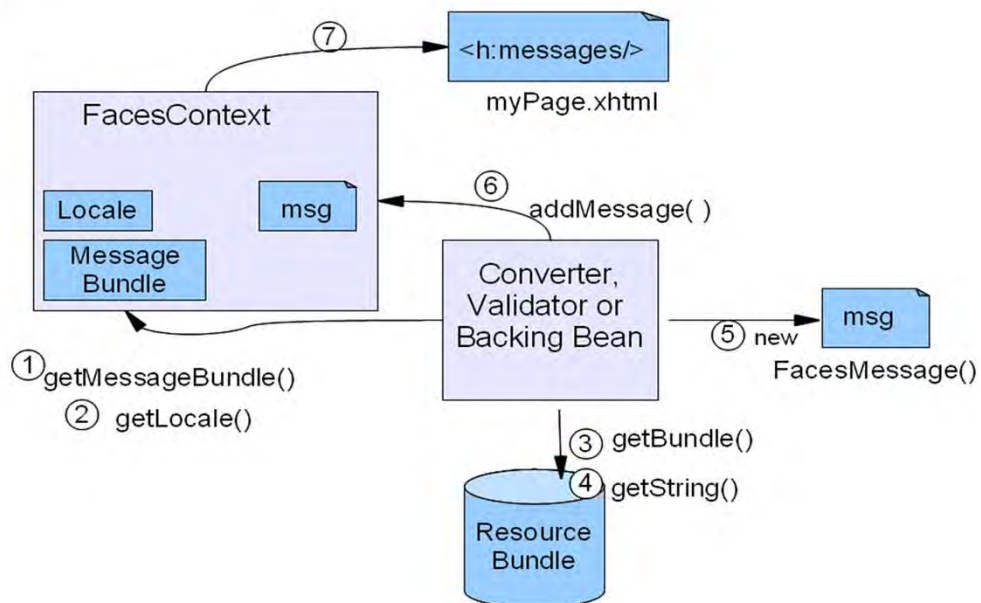
→ Message texts in Java code!

In this sayHello() action method we create a FacesMessage to inform the users about errors in the age input field.

The FacesMessage constructor takes the severity, the message string and an optional detail message string plus parameters as arguments.

Instead of using hard coded messages in your backing beans, you usually want to read all error messages from a message bundle. So, you preferably use a custom message bundle to define your application messages.

# Message Factory



In order to simplify the retrieving of messages from the custom message bundle, we implement the utility class **MessageFactory**.

The diagram shows the necessary steps.



## Message Factory

---

```
public class MessageFactory {  
    public static FacesMessage  
        getMessage(FacesMessage.Severity severity,  
                   String key, Object... params) {  
  
        FacesContext context =  
            FacesContext.getCurrentInstance();  
  
        String name =                               // ( 1 )  
            context.getApplication().getMessageBundle();  
    }  
}
```

We first have to obtain the (name of) the message bundle and the locale from the Faces context.



## Message Factory (2)

---

```

// ( 2 )
Locale locale = context.getViewRoot().getLocale();

ResourceBundle bundle = // ( 3 )
    ResourceBundle.getBundle(name, locale);

String summary = "???" + key + "??";
String detail = null;
```

With this information we can read the message bundle itself.

We initialize the summary string by a default message. This gives us an obvious error string if the error message could not be found.



## Message Factory (3)

```
try { // ( 4 )
    summary = MessageFormat.format(bundle.getString(key),
                                    params);
    detail = MessageFormat.format(bundle.getString(
                                    key + "_detail"), params);
} catch (MissingResourceException e) {
} // ( 5 )
return new FacesMessage(severity, summary, detail);
}

public static void error(String key, Object... params) {
    FacesContext context = FacesContext.getCurrentInstance();
    context.addMessage(null,
        getMessage(FacesMessage.SEVERITY_ERROR, key,
            params));
}
```

The detail string is optional.

If the detail string is not defined, the JSF framework automatically displays out the summary string instead.

Finally the message has to be added to the Faces context, from where the JSF page will retrieve it afterwards.

For this we implement the methods `error()` and `info()`.

`error()` adds an error message (with error severity), `info()` adds an information message (with info severity).



## HelloBean Example

---

```
public String sayHello() {  
    if (age < MIN_AGE || age > MAX_AGE) {  
        MessageFactory.error( AGE_MESSAGE_ID,  
                               MIN_AGE, MAX_AGE);  
        return null;  
    }  
    if (age < 11) {  
        return "hello";  
    } else {  
        return "goodMorning";  
    }  
}
```

Now, we can use this MessageFactory class in our backing beans or validator or converter methods.



## Displaying Application Messages

```
// display only global messages
```

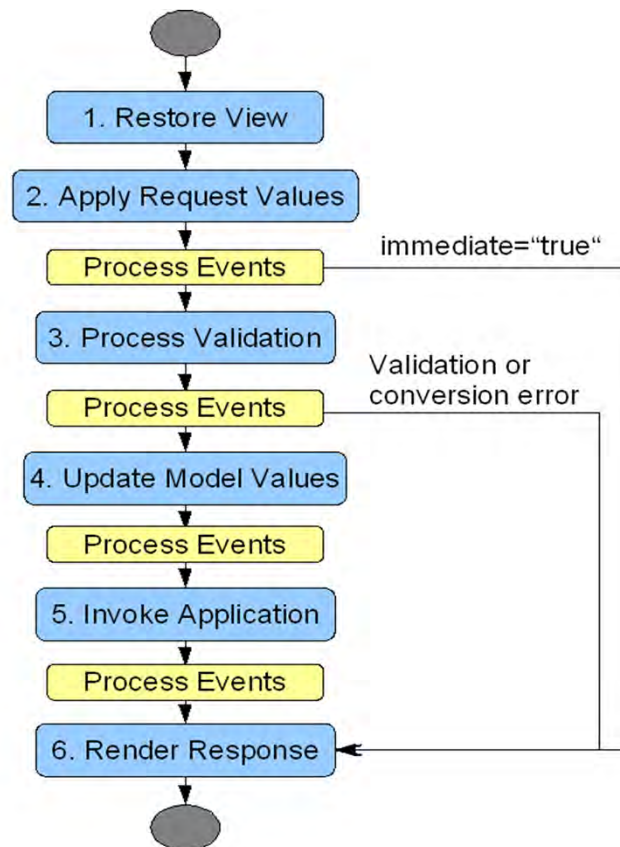
```
<h:messages globalOnly="true">
```

Application messages do not belong to any component on the JSF page. If we want to retrieve the application messages only (i.e. the messages, which belong to no component) we use the `globalOnly` attribute.





## 7 The Request Processing Lifecycle



When a JSF view is requested, the components in the view are asked to perform tasks in a certain order, defined as phases of a request processing life cycle.

This diagram shows the main tasks for each phase.



## Restore View (1)

---

- Extract the view ID → find current view
- Look up the components for the current view, create them if necessary
- Restore or create component values, event listeners, validators and converters
- Initial Request → skip to Render Response Phase (6)

When a request for a JSF page is made (e.g. when a link or a button is clicked) the JSF implementation begins the restore view phase.

During this phase, the JSF implementation builds the view of the page, wires event handlers and validators to components in the view, and saves the view in the FacesContext instance. The FacesContext instance contains all the information needed to process a single request. All the application's component tags, event handlers, converters, and validators have access to the FacesContext instance.

If the request for the page is an initial request, the JSF implementation creates an empty view during this phase and the life cycle advances to the render response phase. The empty view will be populated when the page is processed during a postback.

If the request for the page is a postback, a view corresponding to this page already exists. During this phase, the JSF implementation restores the view by using the state information saved on the client or the server.



## Apply Request Values (2)

- Decoding: extract user input from input components
- Create action events and add them to Faces Context for later processing
  
- immediate = "true" in a command component  
→ Action sources fire action events
- immediate = "true" in an input component  
→ Validation of submitted values

After the component tree is restored, each component in the tree extracts its new value from the request parameters by using its decode method. The value is then stored locally in the component. If the conversion of the value fails, an error message associated with the component is generated and queued on the Faces Context. This message will be displayed during the render response phase, along with any validation errors resulting from the process validations phase.

If events have been queued during this phase, the JSF implementation broadcasts the events to interested listeners.

If some components on the page have their immediate attributes set to true, the validation, the conversion, and all events associated with these components will be processed during this phase.

At the end of this phase, the components are set to their new values, and messages and events have been queued.



## Process Validations (3)

---

- Convert submitted values
- Validate submitted values
  - Conversion or validation errors → Render Response (6), else ...
- Set local value to the converted and validated submitted values
- Fire value change events

During this phase, the JSF implementation processes all validators registered on the components in the tree.

If the local value is invalid, the JSF implementation adds an error message to the Faces Context instance, and the life cycle advances directly to the render response phase so that the page is rendered again with the error messages displayed.

If events have been queued during this phase, the JSF implementation broadcasts them to interested listeners.



## Update Model Values (4)

---

- Find appropriate bean in the scope  
(request, session, application, . . . )
- Set the bean property to the new value

After the JSF implementation determines that the data is valid, it can walk through the component tree and set the corresponding server-side object properties to the components' local values. The JSF implementation will update the bean properties pointed at by an input component's value attribute.

If events have been queued during this phase, the JSF implementation broadcasts them to interested listeners.



## Invoke Application (5)

---

- Handle all fired events
  - Execute default action listener method
  - Call the event handler methods of all registered event listeners
  - Retain the outcome of the fired action method

During this phase, the JSF implementation handles any application-level events, such as submitting a form or linking to a different page.

If the view being processed was reconstructed from state information from a previous request and if a component has fired an event, these events are broadcast to interested listeners.

When processing this event, a default ActionListener implementation retrieves the outcome from the component's action attribute. The listener passes the outcome to the default NavigationHandler. The NavigationHandler matches the outcome to the proper navigation rule to determine which page needs to be displayed next. The JSF implementation then sets the response view to that of the new page. Finally, the JSF implementation transfers control to the render response phase.



## Render Response (6)

---

- Determine the next page (Navigation)
- Display the next view
  - Render tree of UIComponents
  - with the updated values from the backing beans
- Invoke converters
- Save the state of the new view
  - for later use in the Restore View Phase

If this is an initial request, the components represented on the page will be added to the component tree as the container executes the page.

If this is not an initial request, the components are already added to the tree so they needn't be added again. In either case, the components will render themselves as the view container traverses the tags in the page.

If the request is a postback and errors were encountered during the apply request values phase, process validations phase, or update model values phase, the original page is rendered during this phase. If the pages contain message or messages tags, any queued error messages are displayed on the page.

After the content of the view is rendered, the state of the response is saved so that subsequent requests can access it and it is available to the restore view phase.





# 8 Converters and Validators

Type Conversion  
and Input Validation

Confer chapter 2.11 and 2.12 of  
<http://jsfatwork.irian.at>



## Conversion and Validation

- Conversion
  - Translation of input values (String to Object)
  - Translation of backing bean properties (Object to String)
  - Using a converter class
- Validation
  - Check for validity of converted input values
  - Using one or more validator methods or validator classes

JSF supports validation and conversion through validator methods of backing beans and validator and converter classes.

Converters try to translate user input strings to backing bean properties and vice versa.

Validation checks the value of a control to see if its current value is acceptable. The associated property in the backing bean is updated only if the value is accepted.

Otherwise, an error message is generated for that specific component, and the associated property is not modified.



# Standard Validators

- f:validateLength
- f:validateDoubleRange
- f:validateLongRange

```
<h:inputText id="age" value="#{helloBean.age}">
  <f:validateLongRange minimum="1" maximum="120"/>
</h:inputText>
<h:message for="age"/>
```

Ihr Name  Bitte geben Sie Ihren Namen ein  
Ihr Alter  Der eingegebene Wert muss zwischen 1 und 120 liegen.

JSF includes a few standard validators for the most common validation problems like input length or value range.

## **f:validateLength**

Validates the (string) length of a component's value (the number of letters or digits)

## **f:validateDoubleRange**

Validates a double range for a component's value  
(e.g. range -3.141 to 3.141)

## **f:validateLongRange**

Validates a long range for a component's value  
(e.g. range -17 to 123456789)

All these validators use the attributes minimum and maximum.



# Standard Validators

---

## New in JSF2.0

```
<h:inputText value="#{helloBean.name}">
  <f:validateRegex pattern="[A-Z][a-z]*/>
</h:inputText>

<f:validateRequired>
  <h:outputLabel for="name" value="#{texts.yourName}" />
  <h:inputText id="name" value="#{helloBean.name}" />
</f:validateRequired>
```

**f:validateRegex** checks the value of the corresponding input component against the specified pattern using the Java regular expression syntax.

The example on the left allows only names which start with capitals.

**f:validateRequired** enforces the presence of a value. It has the same affect as setting the required attribute on an input component (or a group of input components) to true.



## Validator Methods

```
<h:inputText id="name" value="#{helloBean.name}"
              validator="#{helloBean.nameCheck}">
    <f:validateLength minimum="2" maximum="10"/>
</h:inputText>
```

### Backing Bean: validator method

```
public void nameCheck(FacesContext context,
                     UIComponent component, Object value) {
    if (!value.toString().trim().matches("[A-Z][a-z]*")) {
        FacesMessage message =
            MessageFactory.getMessage(
                INVALID_VALUE_MESSAGE_ID);
        throw new ValidatorException(message);
    }
}
```

For any input component you can register one or more validators.

An input field can also be associated with a validation method of a backing bean. These validation methods are generally used for application specific validation and can not be reused for different applications.

If you register more than one validator, all of them are executed. You can show all the generated error messages by a <h:messages/> tag.



## Validator Class

```
@FacesValidator("jsf.examples.AgeValidator")
public class AgeValidator implements Validator {

    public void validate(FacesContext context,
        UIComponent component, Object value) {
        int age = (Integer) value;
        if (age < 1 || age > 120) {
            FacesMessage message = MessageFactory.getMessage( . . . );
            throw new ValidatorException(message);
        }
    }
}
```

Validator classes raise some additional work. On the other hand, validator classes are more generic and designed to be used in different applications. The validator interface demands for only one method: `validate()`.

The **@FacesValidator** annotation defines the identifier of a validator which is used in the JSF page to reference the validator.

Instead of the annotation, you can also configure your validator in the `faces-config.xml` file:

```
<validator>
  <validator-id>AgeValidator</validator-id>
  <validator-class>jsf.examples.AgeValidator</validator-class>
</validator>
```



## Use of a Validator

---

home.xhtml

```
<h:inputText id="age" value="#{helloBean.age}" >  
  <f:validator validatorId="jsf.examples.AgeValidator" />  
</h:inputText>
```

This validator can then be used in any input component of a JSF page.



## Automatic Conversion

---

All basic Java types are automatically converted between string values and objects.

- Integer, int
- Short, short
- Double, double
- Float, float
- Long, long
- BigDecimal
- BigInteger
- Boolean, boolean
- Byte, byte
- Character, char

When users see an object on the screen, they see it in terms of recognizable text, like May 23, 1980. A Java Date object, on the other hand, is much more than just a string. (Two Date objects can be compared and we can compute the elapsed time between the two dates.)

To create a string representation of an object, and to create an object representation of a string is the purpose of a converter.

If you don't specify a converter, JSF will try to select one for you. JSF provides a set of standard converters to satisfy the basic type conversion needs.

You can also write your own converters, and third-party vendors will provide them as well.

On failure, converters create converter exceptions and send them back to the user as FacesMessages.





## Standard Converters

```
<h:inputText value="#{helloBean.dateOfBirth}">
  <f:convertDateTime type="date"/>
</h:inputText>
```

Date of Birth

```
Price: CHF
<h:outputText value="#{helloBean.price}">
  <f:convertNumber pattern="#,##0.00"/>
</h:outputText>
```

Price: CHF 3,324.00

The converter **f:convertDateTime** can be used to convert a `Date` object to a string and vice versa. The `type` attribute is optional and its value can be either *date* or *time* or *both*, depending on what you want to print out. The default type is *date*.

The number converter **f:convertNumber** is useful for displaying numbers in basic formats like a currency or a percentage.

Further attributes of **f:convertNumber** are *type*, *integerOnly*, *minFractionDigits*, *maxIntegerDigits*, *minIntegerDigits*, *pattern*, ... (cf. JSF Tag Library Documentation, or chapter 2.11.1 of <http://jsfatwork.irian.at>).



## Converter Class

---

```
@FacesConverter("jsf.examples.AgeConverter")
public class AgeConverter implements Converter {

    @Override
    public String getAsString(FacesContext ctxt, UIComponent cmpt,
                             Object value) throws ConverterException {
        return value.toString();
    }
}
```

In many cases, the standard converters will be sufficient; they handle all of the standard Java data types, and also provide sophisticated formatting functionality for dates and numbers.

You have to develop custom converters when you want to make it easy for a front-end developer to display a special data type, or accept input for a special data type.

It is not possible to define the converter methods in a backing bean, as conversion needs two translation methods (string to object and object to string). So, the only way to write a custom converter is by implementing a converter class.

The `@FacesConverter` annotation defines an identifier for the converter. The identifier is used in the JSF page to reference the converter.

The `@FacesConverter` annotation can also have a `forClass` attribute, which means this converter is used for all objects of this class.



## Converter Class

```
@Override
public Object getAsObject(FacesContext ctxt, UIComponent c,
                        String value) throws ConverterException {
    int radix = 10;
    if (value.startsWith("0x")) {
        radix = 16; value = value.substring(2);
    }
    if (value.startsWith("0")) {
        radix = 8; value = value.substring(1);
    }
    try {
        return Integer.parseInt(value, radix);
    } catch (NumberFormatException e) {
        FacesMessage message = MessageFactory.getMessage( . . . );
        throw new ConverterException(message);
    }
}
```

Instead of the `@FacesConverter` annotation in the converter class, you can also declare your converters in the `faces-config.xml` file:

```
<converter>
  <converter-id>
    jsf.examples.AgeConverter
  </converter-id>
  <converter-class>
    jsf.examples.AgeConverter
  </converter-class>
</converter>
```



## Use of Converter Class

---

```
<h:outputLabel for="age" value="#{texts.yourAge}" />
<h:inputText id="age" value="#{helloBean.age}" >
  <f:converter converterId="jsf.examples.AgeConverter" />
</h:inputText>
<h:message for="age" errorClass="errorMessage" />
```

Ihr Name	<input type="text" value="Anton"/>
Ihr Alter	<input type="text" value="0b101"/>

In our example, the user can insert any octal, hex or decimal number, which is automatically converted into the corresponding age value.



## 9 Custom Tags

How to create converter and validator tags

As we have already seen, JSF has several services that enable you to build web applications quickly. Many standard components are available, and different IDE vendors provide many additional components.

However, the real power of JSF lies in its sophisticated component model and an extensible architecture that allows you to create your own user interface extensions, like custom components, renderers, validators, and converters.



## No custom tag is necessary to

- ... format an existing component → CSS
- ... display a value in a specific way → Converter
- ... validate user input → Validator

### Before you write a custom tag

- [jsfcentral.com/products](http://jsfcentral.com/products)
- [myfaces.apache.org/tomahawk](http://myfaces.apache.org/tomahawk)
- [icesoft.org](http://icesoft.org)
- [jboss.org/richfaces](http://jboss.org/richfaces)
- ...

Before you write a new custom tag or custom component, you should make sure that there is no alternative way you can reach your goal: you can visit the JSF Central page ([jsfcentral.com/products](http://jsfcentral.com/products)), for example, to find out about different JSF products and libraries.



## Custom Validator or Converter Tag

- Validator or Converter without attributes →  
Custom Validator or Converter Class
- Validator or Converter with attributes →  
Custom Tag

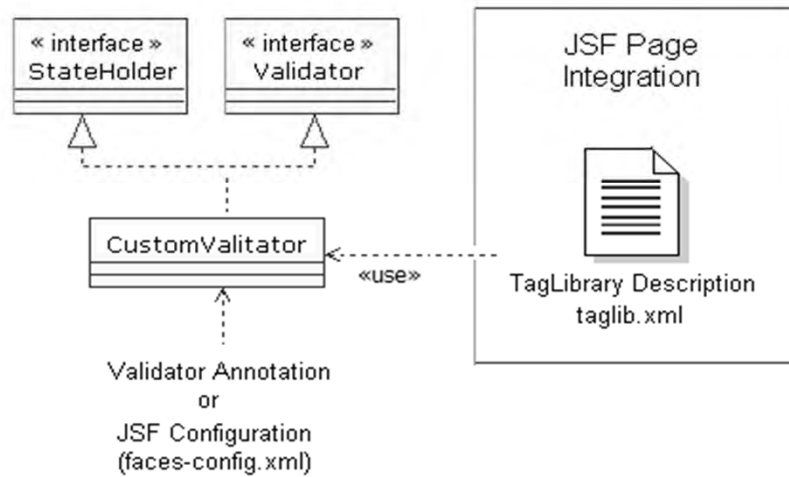
```
<html ... xmlns:hello="http://jsf.examples/hello">
...
<h:inputText id="age" value="#{helloBean.age}">
  <hello:convertNumber radix="2" />
  <hello:validateRange min="1" max="120" />
</h:inputText>
...
</html>
```

With custom tags we are able to provide tag attributes whose values are used as custom validator or converter configuration properties (e.g. a min or max attribute to limit the input value).

Custom tags for validators and converters are implemented in the same way.



## The Parts of a Custom Validator Tag



We explain the implementation of a custom tag on the basis of a simple custom validator.

To create a custom validator tag with a property value, we have to write a CustomValidator class which implements the Validator and StateHolder interfaces.

The validator interface requires a public validate() method.

If the Validator class has configuration properties the class must also implement the StateHolder interface such that the property values are saved and restored with the view.





## Tag Library Description

```
<facelet-taglib version="2.0"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee" ... >

  <namespace>http://jsf.examples/hello</namespace>

  <tag>
    <tag-name>validateRange</tag-name>
    <validator>
      <validator-id>
        jsf.examples.RangeValidator
      </validator-id>
    </validator>
  </tag>
  <tag>
    ...
  </tag>
</facelet-taglib>
```

} tag

For the JSF integration we need a tag library description of the new tag. Here we determine the name and the namespace of the tag as well as its validator-id. The validator-id is a name which is equally defined in the validator annotation of the validator class. Usually, validator-id is the same as the validators class name. The namespace ensures the uniqueness of the tag names.



## Validator Class

---

```
package jsf.examples;
import . . .

@FacesValidator("jsf.examples.RangeValidator")
public class RangeValidator implements Validator, StateHolder {

    private int min = Integer.MIN_VALUE;
    private int max = Integer.MAX_VALUE;

    private boolean transientValue = false;
```

The `@FacesValidator` annotation which associates the validator id is placed at the head of the validator class.

We also define the two configuration properties of the `RangeValidator`, `min` and `max`. `min` and `max` are optional attributes of the tag, so we define some default values.

`min` and `max` are not transient, so we define the boolean `transientValue` to be `false`.



## Validator Class: Properties

---

```
public void setMin(int min) {  
    this.min = min;  
}
```

```
public void setMax(int max) {  
    this.max = max;  
}
```

The setter methods of the validator attributes are used to store the values of max and min defined in the JSF-page into their corresponding configuration properties.

If the attribute value for min or max in the JSF-page is a value expression, this expression is automatically evaluated and the corresponding result is stored in the max/min property.



## Validator Class: validate()

```
@Override
public void validate( FacesContext context,
                     UIComponent component, Object value) {
    int age = (Integer) value;
    if (age < min || age > max) {
        FacesMessage message = MessageFactory.getMessage(
            FacesMessage.SEVERITY_ERROR,
            INVALID_VALUE_MESSAGE_ID, min, max);
        throw new ValidatorException(message);
    }
}
```

The validator interface requires a public validate() method. Here we compare the input value against the two properties min and max.



## Validator Class: StateHolder

```
@Override
public Object saveState(FacesContext context) {
    return new int[]{min, max};
}

@Override
public void restoreState(FacesContext context, Object state) {
    min = ((int[]) state)[0];
    max = ((int[]) state)[1];
}

@Override
public boolean isTransient() {
    return transientValue;
}

@Override
public void setTransient(boolean transientValue) {
    this.transientValue = transientValue;
}
}
```

If the Validator class uses property values which should to be saved and restored together with the view, the class must also implement the StateHolder interface. This means we have to implement the methods `saveState()` and `restoreState()`, to define which values have to be saved and how they are restored. The method `isTransient()` returns true, if all properties are transient (nothing has to be saved). Therefore, in this example the default value for `transientValue` is false.



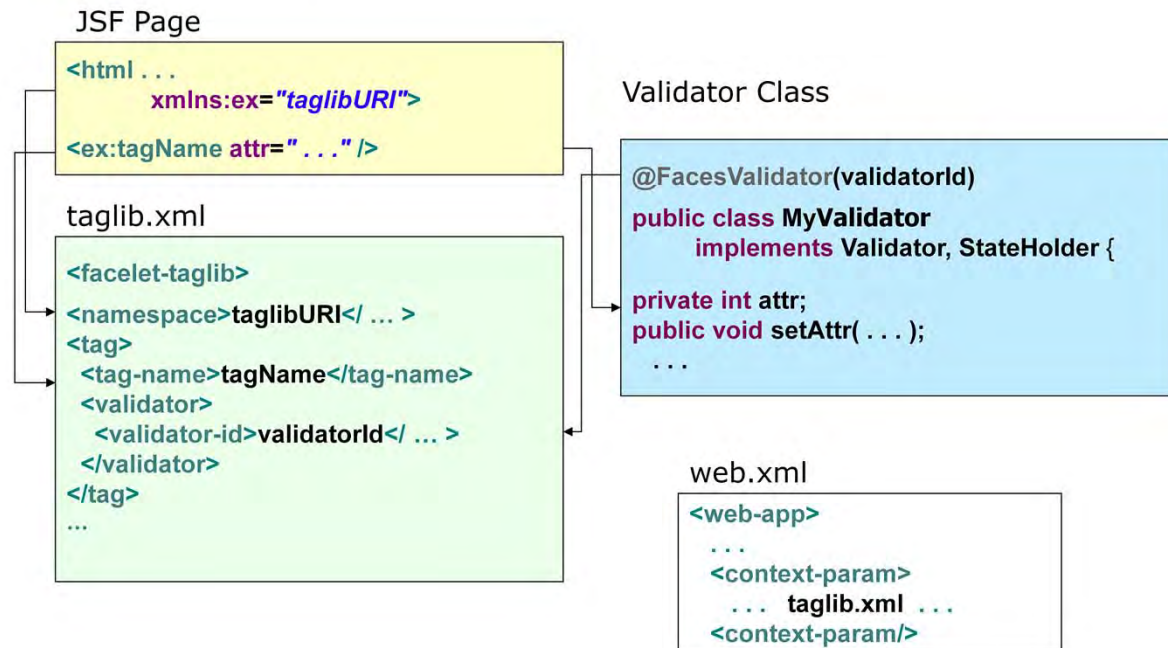
## Tag Library Definition in web.xml

```
<web-app version="3.0" . . . >  
    . . .  
    <context-param>  
        <param-name>  
            javax.faces.FACELETS_LIBRARIES  
        </param-name>  
        <param-value>  
            /WEB-INF/jsf.examples.taglib.xml  
        </param-value>  
    </context-param>  
</web-app>
```

The tag library definition has to be declared in the web.xml file.



# Configuration Summary









# 10 Composite Components

## Facelets Component

So far, we have used Facelets as the page declaration language and as an alternative to JSP. However, Facelets also offers a wide range of features that make the development of a JSF application easier.

In this section, we present the possibility to compose new components from existing ones.



## What is a Composite Component?

- Composition of existing JSF components and XHTML code
- Is defined by some special markup tags
- Can be parametrized by attributes and actions
- Can have validators, converters, and action listeners
- Can be stored in a library

Composite components is one of the most important new features of JSF 2.0. With this new feature, developers have the possibility to build components from almost any page fragments.

A composite component is a special type of template. It has a customized functionality and can have validators, converters, and listeners attached to it like any other JSF component.

Using the resources facility, the composite component can be stored in a library that is available to the application from the resources location.



# Composite Component Structure

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:composite="http://xmlns.jcp.org/jsf/composite">

  <body>
    <composite:interface>
      ...
    </composite:interface>
    <composite:implementation>
      ...
    </composite:implementation>
  </body>
</html>
```

A composite component consists of two major parts:

**composite:interface** defines the interface of the composite component and declares all the features (attributes, actions, ...) that are relevant for the usage of this component.

**composite:implementation** defines the implementation of the composite component.



## Attribute Access with cc.attrs

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:composite="http://xmlns.jcp.org/jsf/composite">
  <body>
    <composite:interface>
      <composite:attribute name="myAttribute"/>
      ...
    </composite:interface>
    <composite:implementation>
      <h:outputText value="#{cc.attrs.myAttribute}" />
      ...
    </composite:implementation>
  </body>
</html>
```

The names of the attributes defined in the interface part can be accessed by using the keyword `cc.attrs` in EL expressions.



## Composite Component Tags

**composite:interface**

declares the usage contract for a composite component

**composite:implementation**

contains the implementation of the composite component

**composite:attribute**

declares an attribute

**composite:actionSource**

declares an ActionSource

**composite:valueHolder**

declares a ValueHolder

**composite:editableValueHolder**

declares an EditableValueHolder

The different composite component tags are:

**composite:interface** declares the usage contract for a composite component, i.e. the components attributes, action sources, value holders, ...

**composite:attribute** declares an attribute that may be given to an instance of the composite component in which this tag is declared.

**composite:actionSource** declares that the belonging composite component exposes an implementation of ActionSource.

**composite:valueHolder** exposes a component that holds a value. This allows the JSF page author to register converters to (some part of) the composite component.

**composite:editableValueHolder** exposes a component that holds an editable value. This allows the JSF page author to register converters and validators to (some part of) the composite component.



## Composite Component Example

A simple icon component



We show an implementation of a simple icon component with some required fields.  
The icon provides a command link, which triggers an action method as soon as the user clicks on the icon.



# Composite Interface

```
<html ...  
  xmlns:composite="http://xmlns.jcp.org/jsf/composite">  
  
  <composite:interface>  
    <composite:attribute name="image" required="true" />  
    <composite:attribute name="doValidation"  
      default="true" />  
    <composite:attribute name="actionMethod"  
      method-signature="java.lang.String action()" />  
  </composite:interface>
```

The icon composite component is defined in the icon.xhtml file with the following attributes:

- required="true" forces the page authors to provide an image for the icon.
- The doValidation attribute is optional and has a default value. It allows the page author to determine the validation.
- The page author can assign an actionMethod, which is invoked as soon as the user clicks on the icon.



# Composite Implementation

```
<composite:implementation>
  <h:commandLink action="#{cc.attrs.actionMethod}"
                 immediate="#{not cc.attrs.doValidation}">
    <h:graphicImage url="#{cc.attrs.image}" />
  </h:commandLink>
</composite:implementation>

</html>
```

We refer to the attributes (image, doValidation and actionMethod) defined in the interface by **`#{cc.attrs. . . }`** expressions.

The actionMethod is executed as soon as the user clicks on the icon.

For commandLinks the immediate attribute is false, per default. If the attribute value of immediate is set to true, no validation of the input values is performed.

The commandLink is rendered with the image which is found at the given URL.





# Namespace and Tag Name

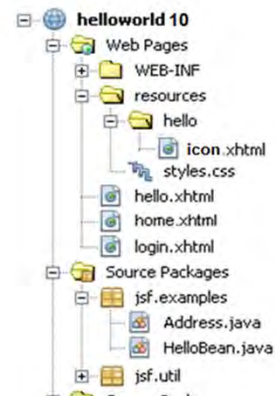
The helloworld directory structure

The name of the tag:

```
<hello:icon image="..." />
```

The namespace:

```
xmlns:hello =  
    "http://xmlns.jcp.org/jsf/composite/hello"
```



The namespace of the composite component and the name of the component's tag are defined by the name of the resources library and the name of the XHTML document, respectively.

The name of the tag is defined by the file name of the composite component (and the namespace prefix), here **hello:icon**.



## Usage of the Component

```
<html ...  
  xmlns:hello="http://xmlns.jcp.org/jsf/composite/hello">  
<h:head> ... </h:head>  
<h:body>  
  <h:form id="main">  
    . . .  
    <hello:icon image="#{...}" actionMethod="#{...}"  
      doValidation="true"/>  
    . . .  
  </h:form>  
</h:body>
```



We can use the icon tag after defining the appropriate namespace (here `xmlns:hello="http://xmlns.jcp.org/jsf/composite/hello"`) and providing the attributes defined in the interface of the component (`image`, `doValidation` and `actionMethod`).



## Composite Component Example

A simple address component

Name	<input type="text" value="Anton"/>
Street	<input type="text"/>
City	<input type="text"/>
Country	<input type="text"/>

We show an implementation of a simple address component with some required fields. A light blue background color shows which input fields belong to required input values.



# Composite Interface

```
<html ...  
    xmlns:composite="http://xmlns.jcp.org/jsf/composite">  
  
    <composite:interface>  
        <composite:attribute name="value" required="true"/>  
        <composite:attribute name="nameRequired"  
            default="true"/>  
        <composite:attribute name="streetRequired"  
            default="false"/>  
        <composite:attribute name="cityRequired"  
            default="false"/>  
        <composite:attribute name="countryRequired"  
            default="false"/>  
    </composite:interface>
```

The inputAddress composite component is defined in the inputAddress.xhtml file with the following attributes:

- **value** defines the value of the component. This value must be an Address type.
- **nameRequired**, **streetRequired** and **cityRequired** are boolean values. The page author can define whether the different fields in the address are optional or required input fields. Required fields have a light blue background (cf. inputAddress.css).



# Composite Implementation

---

```
<composite:implementation>
  <h:outputStylesheet library="hello"
    name="inputAddress.css"/>
  <h:panelGrid columns="3">
    ...

  </h:panelGrid>
</composite:implementation>

</html>
```

The components style definitions are stored in the components own style sheet, usually named after the component.

The name/path of the stylesheet is defined in the implementation part of the inputAddress component.

The general layout of the component is defined here by a h:panelGrid tag.



## Composite Implementation

```
<h:panelGrid columns="3">
  <h:outputLabel for="name" value="#{cc.resourceBundleMap.name}"/>
  <h:inputText id="name" value="#{cc.attrs.value.name}"
    required="#{cc.attrs.nameRequired}"
    requiredMessage="#{cc.resourceBundleMap.requiredMessage}"
    styleClass="required#{cc.attrs.nameRequired}"/>
  <h:message for="name" errorClass="errorMessage"/>
  <h:outputLabel value="#{cc.resourceBundleMap.city}"/>
  ...
</h:panelGrid>
```

We refer to the attributes defined in the interface by **`#{cc.attrs. . . }`** expressions. Therefore, we can refer to the value attribute by **`#{cc.attrs.value}`**.

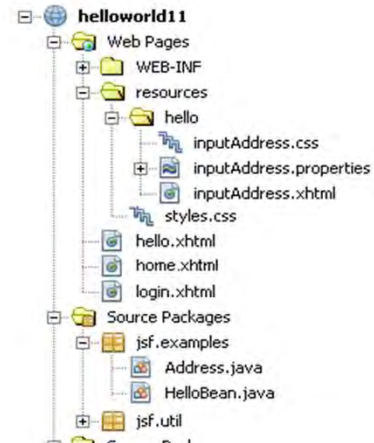
The labels are read from the resource bundle map using a **`#{cc.resourceBundleMap. . . }`** expression.

This refers to a `*.properties` file with the same name as the component itself, here to the `inputAddress.properties` file in the components directory.



# Namespace and Tag Name

The helloWorld directory structure



The name of the tag:

```
<hello:inputAddress value="..." />
```

The namespace:

```
xmlns:hello = "http://xmlns.jcp.org/jsf/composite/hello"
```

The namespace of the composite component and the name of the component's tag are defined by the name of the resources library and the name of the XHTML document, respectively.

In our example, the namespace is **http://xmlns.jcp.org/jsf/composite/hello**. The name of the tag is defined by the file name of the composite component (and the namespace prefix), here **hello:inputAddress**.



## Usage of the Component

```
<html ...  
  xmlns:hello="http://xmlns.jcp.org/jsf/composite/hello">  
<h:head> ... </h:head>  
<h:body>  
  <h:form id="main">  
    . . .  
    <hello:inputAddress value="#{helloBean.address}"  
      countryRequired="true"/>  
    . . .  
  </h:body>
```

Name	<input type="text" value="Anton"/>
Street	<input type="text"/>
City	<input type="text"/>
Country	<input type="text"/>

We can use the component `inputAddress` tag after defining the appropriate namespace (here `xmlns:hello="http://xmlns.jcp.org/jsf/composite/hello"`) and providing the attributes defined in the interface of the component.

By default, name is a required field and therefore has a light blue background color. The country field is set required by the page author.





# Composite Component Example

A simple login component

## Login

Name	<input type="text"/>
PIN	<input type="text"/>
<input type="button" value="Login"/>	

We show an implementation of a simple login component with two value holders and an action source.



## The userLogin Interface

---

```
<composite:interface>
  <composite:attribute name="name" required="true"/>
  <composite:attribute name="pin" required="true"/>
  <composite:attribute name="login"
    method-signature="java.lang.String action()"/>
  <composite:editableValueHolder name="all"
    targets="loginName loginPin"/>
  <composite:editableValueHolder name="userName"
    targets="loginName"/>
  <composite:editableValueHolder name="userPin"
    targets="loginPin"/>
</composite:interface>
```

For the login composite component, we allow the page author to register validators to the name and the pin input text fields, as well as an action listener to the login button.

The type for the pin input field is expected to be an integer.

The login attribute has to be bound to an action method.



## The Implementation: Layout

```
<composite:implementation>
  <h:panelGrid columns="3">
    . . .

  </h:panelGrid>
  <h:commandButton id="login" action="#{cc.attrs.login}"
    value="#{texts.login}"/>
</composite:implementation>
```

The layout is again defined by a panelGrid with 3 columns (for the label, the input text field, and for the error message).

At the bottom of the panel grid we place a „Login“ button with the login action method, as defined by the interface.



## The Implementation: Functionality

```
<h:outputLabel for="loginName" value="#{texts.name}"/>
<h:inputText id="loginName" value="#{cc.attrs.name}"
    required="true"/>
<h:message for="loginName" errorClass="errorMessage"/>

<h:outputLabel for="loginPin" value="#{texts.pin}"/>
<h:inputSecret id="loginPin" value="#{cc.attrs.pin}"
    required="true" />
<h:message for="loginPin" errorClass="errorMessage"/>
```

The inputText and inputSecret component each have an id, which is the target of the corresponding editableValueHolder.

This id can be used by the page author to register a validator or a special converter.



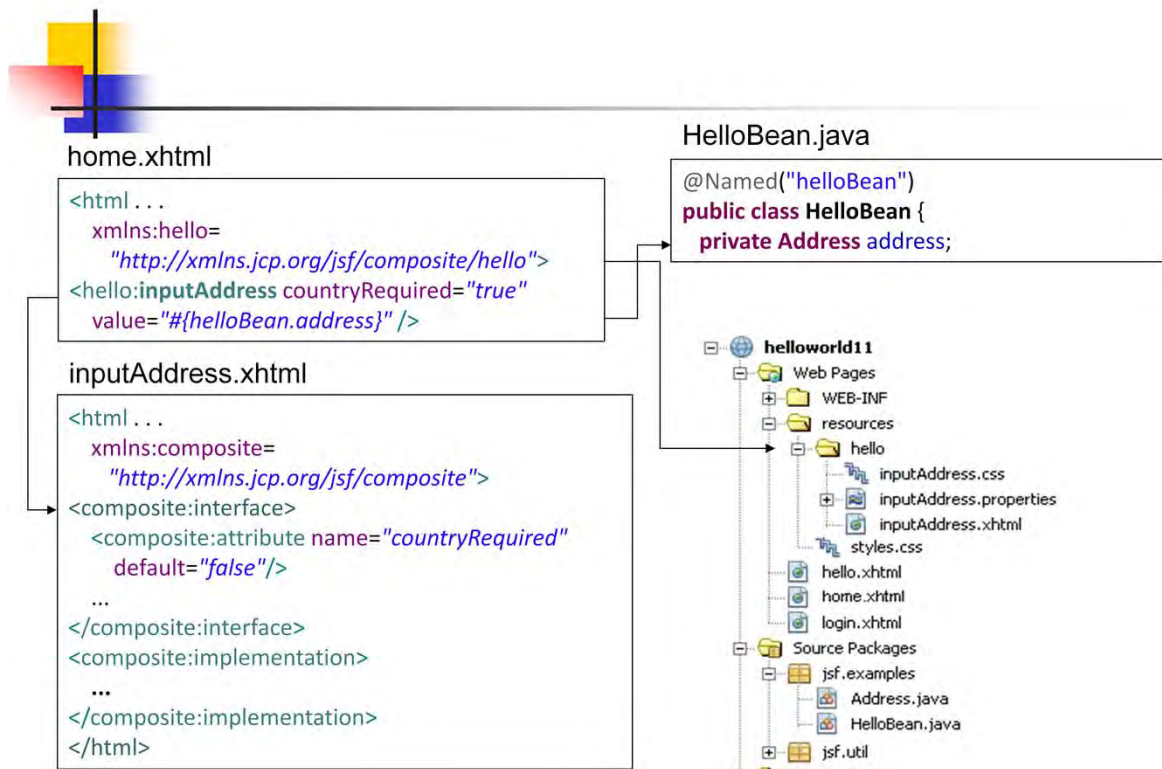
## Usage of userLogin Component

```
<html ...  
  xmlns:hello="http://xmlns.jcp.org/jsf/composite/hello">  
<h:form id="main">  
  . . .  
  
  <hello:userLogin name="#{loginBean.name}"  
    pin="#{loginBean.pin}" login="#{loginBean.login}">  
    <f:validateLength for="all" minimum="5"/>  
    <f:validateRegex for="userName" pattern="[A-Z][a-z]*"/>  
  </hello:userLogin>  
  
  <h:messages ... />  
</h:form>
```

Name	<input type="text" value="George"/>
PIN	<input type="password" value="....."/>
<input type="button" value="Login"/>	

Because of the `editableValueHolder` interface, the page user can register validators for the name and the pin input text field.

The action method for the "Login" button is assigned by the value expression of the login attribute.



Here we can see the different parts which are necessary to build a custom component and their dependencies.



# 11 AJAX Support

Asynchronous JavaScript and XML



# Introduction

---

## AJAX

- allows to automatically update parts of a web page
- provides users with rich agile interfaces
- uses asynchronous communication between browser and web server

Classic web pages reload the entire web page if (part of) its content changes.

AJAX is a technique that allows web pages to be updated asynchronously by exchanging small amounts of data with the server behind the scenes. This means that it is possible to update parts of a web page without reloading the whole page.

JSF 2.0 defines a JavaScript library that covers basic AJAX operations such as sending a request and process the response. This standard interface ensures that all components provide the same functionality.

JSF 2.0 extends the JSF life cycle so that only the relevant parts of the component tree are processed and rendered (partial-view processing, partial-view rendering).





## Based on Internet Standards

---

AJAX uses

- a XMLHttpRequest object
- JavaScript
- DOM
- XML

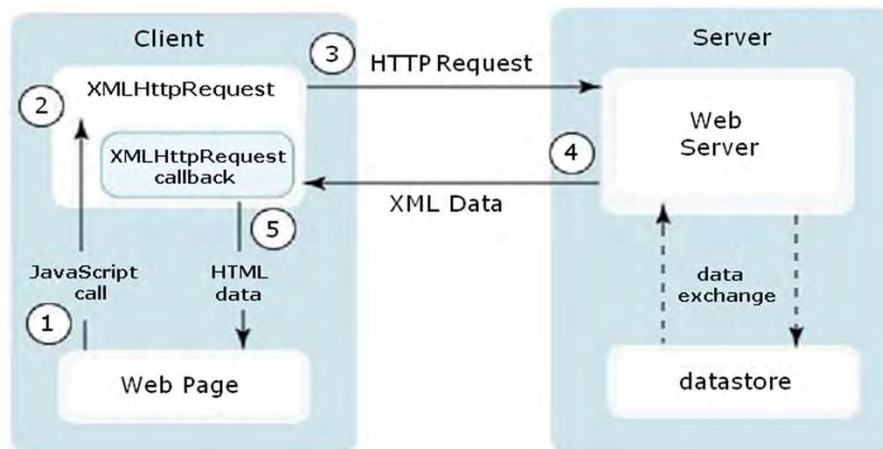
AJAX applications are browser and platform independent

AJAX is based on the following Internet standards:

- XMLHttpRequest objects for the asynchronous communication between the browser and the web server
- JavaScript for responding to events and the interaction with the browser
- DOM for accessing and manipulating the structure of the HTML page
- XML for the data passed between the browser and the web server



## Sequence of an AJAX Request



1. The user generates an event that results in a JavaScript call.
2. An XMLHttpRequest object is created. The request parameter includes the ID of the component that generated the event and any values the user may have entered.
3. The XMLHttpRequest object makes an asynchronous request to the server. In the case of an AJAX-aware JSF component, a PhaseListener object receives the request and processes it.
4. The PhaseListener object returns an XML document containing any updates that need to go to the client.
5. The XMLHttpRequest object receives the XML data, processes it, and updates the HTML DOM representing the web page with the new data.



Register an AJAX behavior instance with one or more UIComponents

```
<f:ajax event="JavaScriptEvent" execute="executeComponent"
        render="renderComponent"/>
```

## Example

```
<f:ajax event="keyup" execute="@this" render="ageMessage" />
```

The **<f:ajax>** tag may be nested within a single component (enabling AJAX for a single component), or it may be wrapped around multiple components (enabling AJAX for many components). The tag has the following attributes:

**event:** The event that triggers the AJAX request. Possible values are all JavaScript events without the prefix *on*. Examples are *change* or *keyup* for input components or *action* for control components.

The default value of this attribute is determined by the component type (confer chapter 6.2 of <http://jsfatwork.irian.at>).

**execute:** A space-separated list of IDs of the components to be executed while processing the AJAX request. Other possible values are the constants *@this* (the element itself), *@form* (the form of the element), *@all* (all elements) and *@none* (no element). The default value is *@this*.

**render:** The space-separated list of IDs of the components to be rendered while processing the AJAX request. Other possible values are *@this*, *@form*, *@all* and *@none*. The default value is *@none*.



Install an event listener

```
<f:event type="SystemEvent" listener="renderComponent"/>
```

Example

```
<f:event type="postValidate" listener="#{helloBean.doIt}" />
```

The **<f:event>** tag is also a new feature of JSF 2.0 and allows to install an event listener which is called at a specific time during the JSF lifecycle.

The value of the type attribute is the name of the event for which the listener is installed. Possible values are preRenderComponent, postAddToView, preValidate, and postValidate.

The value of the listener attribute is the name of the listener method to be called.



# HelloWorld AJAX Example

## Hello World

Your Name	<input type="text" value="beat"/>	The first letter of the name must be in upper case.
Your Age	<input type="text" value="123"/>	The age value must be between 1 and 120.
Your Country	<input type="text" value="Germany"/>	
<input type="button" value="Say Hello"/>		



# hello.xhtml

## Immediate validation and error message processing

```
<h:inputText id="name" label="#{texts.yourName}"
    value="#{helloBean.name}" required="true"
    requiredMessage="#{texts.enterYourName}"
    validatorMessage="#{texts.invalidName}">
    <f:validateRegex pattern="[A-Z][a-z]*" />
    <f:ajax event="blur" render="nameMessage" />
</h:inputText>
<h:message id="nameMessage" for="name" />
```

Your Name  The first letter of the name must be in upper case.

In the first part of our form, we use AJAX for the early validation of the inserted name. The regex validator produces an error message, if the name does not start with a capital letter. This error message shall be produced as soon as we leave the name input field.

Here, the `<f:ajax>` tag has the attribute `event="blur"` which defines that the AJAX request is triggered as soon as the input field loses the focus. The `render` attribute defines the message element to be rendered during the AJAX request.



## Immediate validation and error message processing

```
<h:inputText id="age" label="#{texts.yourAge}"
  value="#{helloBean.age}" required="true"
  validatorMessage="#{texts.invalidAge}">
  <f:validateLongRange minimum="1" maximum="120" />
  <f:ajax event="keyup" render="ageMessage" />
</h:inputText>
<h:message id="ageMessage" for="age" />
```

Ihr Alter  Das Alter muss zwischen 1 und 120 liegen.

In a similar way, we force an early validation of the age input field. Here, the event we use is *keyup*, which causes the inserted value to be validated as soon as the user has typed a key.



# hello.xhtml

---

## Input completion

```
<h:inputText id="country" label="#{texts.yourCountry}"
  value="#{helloBean.country}"
  required="true"
  requiredMessage="#{texts.enterYourCountry}">
  <f:event type="preValidate"
    listener="#{countryCompleter.complete}" />
  <f:ajax event="keyup" render="country" />
</h:inputText>
```

Ihr Land

In this example, the input is automatically completed to a country name.

With the <f:ajax> tag we force an early processing of the country text field.

With the <f:event> tag we install a listener method (here the complete() method of the CountryCompleter backing bean). This method is called before the third life cycle phase (validation and conversion) and tries to complete the given input keys to a well known country name.