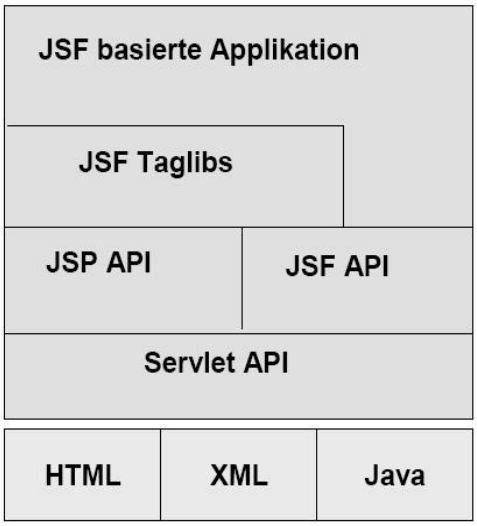
JAVA SERVER FACES

2521 Web Applikationen I

JAVA SERVER FACES

Chapter 01 - Introduction

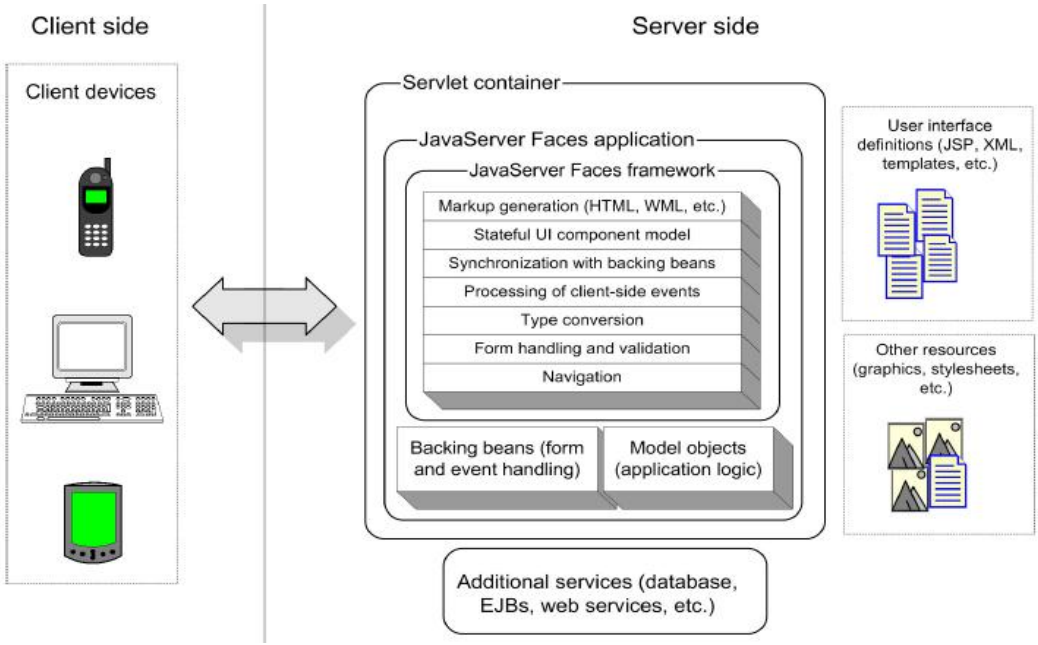
WEB FRAMEWORKS



- JSF (Java Server Faces): Standard Web UI Technologie nach MVC
- Taglib: Bibliothek für JSP-Tags zur Minimierung des Java-Codes in JSPs
- JSP (Java Server Pages): Java-Code in HTML-Seiten eingebettet
- Servlet: Klassenbibliothek zum Empfangen von HTTP-Requests und Generieren von HTML-Seiten
- HTML (Hypertext Markup Language): Darstellung von Web-Inhalten
- XML (Extensible Markup Language): Metasprache zur Definition von Auszeichungssprachen

3

JSF SCOPE



4

A SIMPLE JSF PAGE

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

<h:head>
    <title>Welcome to JSF</title>
    <h:outputStylesheet name="styles.css" />
</h:head>

<h:body>
    <h:form id="main">
        <h:outputText value="Welcome to JSF" styleClass="header" />
        <hr />
        <table>
            <tr>
                <td><h:outputText value="Your Name" /></td>
                <td><h:inputText id="name" value="#{welcomeBean.name}"
                        required="true" /></td>
            </tr>
            <tr>
                <td><h:outputText value="Your JSF experience from 0-10" /></td>
                <td><h:inputText id="experience" value="#{welcomeBean.experience}" /></td>
            </tr>
        </table>
        <p />
        <h:commandButton value="Let's Go" action="#{welcomeBean.letsGo}" />
        <p />
        <hr />
        <h:outputText value="#{welcomeBean.welcomeText}" styleClass="welcomeText" />
    </h:form>
</h:body>
```


TRANSLATION TO HTML

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

<head>
    <title>Welcome to JSF</title>
    <link type="text/css" rel=„stylesheet" href="/example01/javax.faces.resource/styles.css.xhtml" />
</head>

<body>
    <form id="main" name="main" method=„post" action="/example01/welcome.xhtml"
        enctype="application/x-www-form-urlencoded">
        <input type="hidden" name="main" value="main" /> <span class=„header">Welcome to JSF</span>
        <hr />
        <table>
            <tr>
                <td>Your Name</td>
                <td><input id="main:name" type="text" name="main:name" /></td>
            </tr>
            <tr>
                <td>Your JSF experience from 0-10</td>
                <td><input id="main:experience" type="text" name="main:age" value="0" /></td>
            </tr>
        </table>
        <p></p>
        <input type="submit" name="main:j_idt16" value="Let's Go" />
        <p></p>
        <hr />
        <span class="welcomeText"></span><input type=„hidden" name=„javax.faces.ViewState"
            id="javax.faces.ViewState"
            value="8042293025274202304:3330456492363994226" autocomplete="off" />
    </form>
</body>
</html>
```

JSF EXPRESSION LANGUAGE

- Access managed bean property or method
  - #{managedBean.myProperty}
  - #{managedBean.myMethod}
- Access array, list element or map entry
  - #{managedBean.myList[2]}
  - #{managedBean.myMap['key']}

7

JSF EXPRESSION LANGUAGE

- Boolean Expression ==, !=, >=, <=, not empty, ...
  - #{managedBean.myValue != 20}
  - #{managedBean.myValue >= 20}
- conditional view
  - rendered="#{not empty managedBean.myValue}"/>

8

SIMPLE EXAMPLE

WELCOME.XHTML

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

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

<h:head>
    <title>Welcome to JSF</title>
    <h:outputStylesheet name="styles.css" />
</h:head>

<h:body>
    <h:form id="main">
        <h:outputText value="Welcome to JSF" styleClass="header" />
        <hr />
        <table>
            <tr>
                <td><h:outputText value="Your Name" /></td>
                <td><h:inputText id="name" value="#{welcomeBean.name}"
                        required="true" /></td>
            </tr>
            <tr>
                <td><h:outputText value="Your JSF experience from 0-10" /></td>
                <td><h:inputText id="age" value="#{welcomeBean.experience}" /></td>
            </tr>
        </table>
        <p />
        <h:commandButton value="Let's Go" action="#{welcomeBean.letsGo}" />
        <p />
        <hr />
        <h:outputText value="#{welcomeBean.welcomeText}" styleClass="welcomeText" />
    </h:form>
</h:body>
```

WELCOME.XHTML (1)

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

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

<h:head>
   <title>Welcome to JSF</title>
   <h:outputStylesheet name="styles.css" />
</h:head>

.
.
.
```


WELCOME.XHTML (2)

```
…
<h:body>
   <h:form id="main">
       <h:outputText value="Welcome to JSF" styleClass="header" />
       <hr />
       <table>
          <tr>
             <td><h:outputText value="Your Name" /></td>
             <td><h:inputText id="name" value="#{welcomeBean.name}"
                    required="true" /></td>
          </tr>
          <tr>
             <td><h:outputText value="Your JSF experience from 0-10" /></td>
             <td><h:inputText id="age" value="#{welcomeBean.experience}" /></td>
          </tr>
       </table>
…
```

WELCOME.XHTML (3)

```
…
       <p />
       <h:commandButton value="Let's Go" action="#{welcomeBean.letsGo}" />
       <p />
       <hr />
       <h:outputText value="#{welcomeBean.welcomeText}" styleClass="welcomeText" />
    </h:form>
</h:body>
</html>
```


WELCOMEBEAN.JAVA (1)

```
package ch.hftm.examples;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name = "welcomeBean")
@SessionScoped
public class WelcomeBean {

    private String name;
    private int experience;
    private String welcomeText;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getExperience() {
        return experience;
    }
…
```

WELCOMEBEAN.JAVA (2)

```
…
    public void setExperience(int experience) {
        this.experience = experience;
    }

    public String getWelcomeText() {
        return welcomeText;
    }

    public void setWelcomeText(String welcomeText) {
        this.welcomeText = welcomeText;
    }

    public String letsGo() {
        if (experience < 5) {
            welcomeText = name + ", Welcome to JSF basic's !!!";
        } else {
            welcomeText = name + ", Welcome to JSF advanced !!!";
        }
        return null;
    }

}
```


WEB.XML

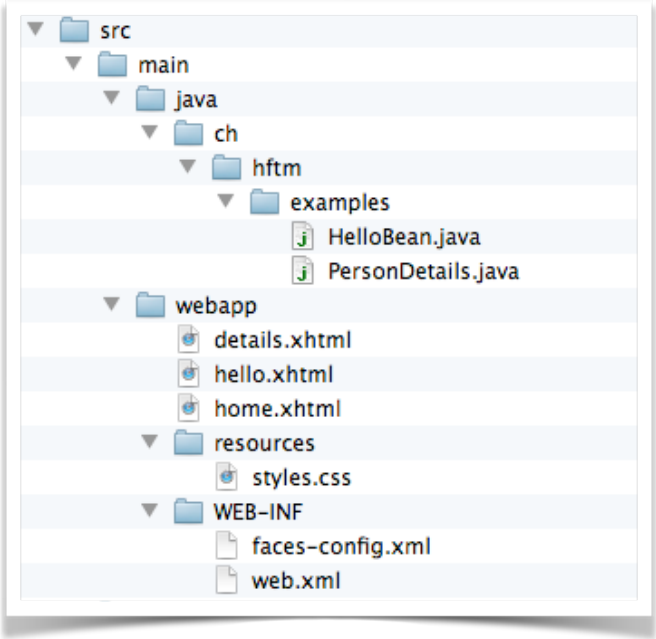
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns=„…“>

    <display-name>Example 01</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>

    <listener>
        <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
    </listener>

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

    <welcome-file-list>
        <welcome-file>welcome.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>
```

JAVA SERVER FACES

Chapter 02 - Application Configuration & Navigation

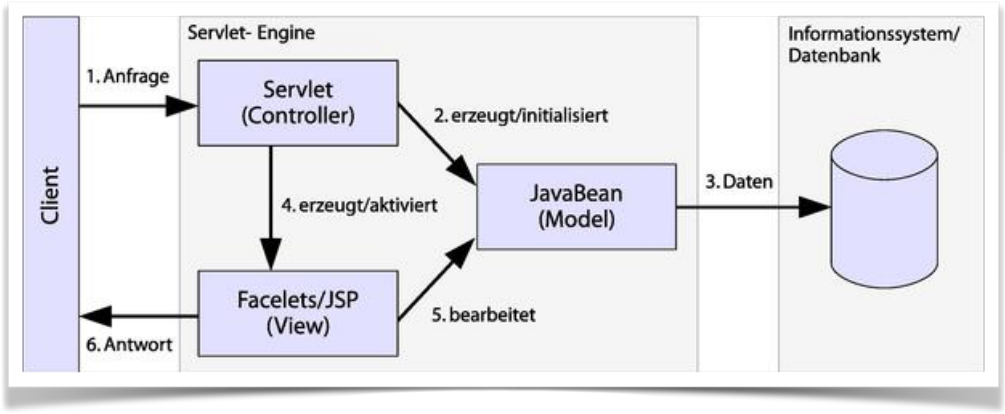
1

DIRECTORY STRUCTURE



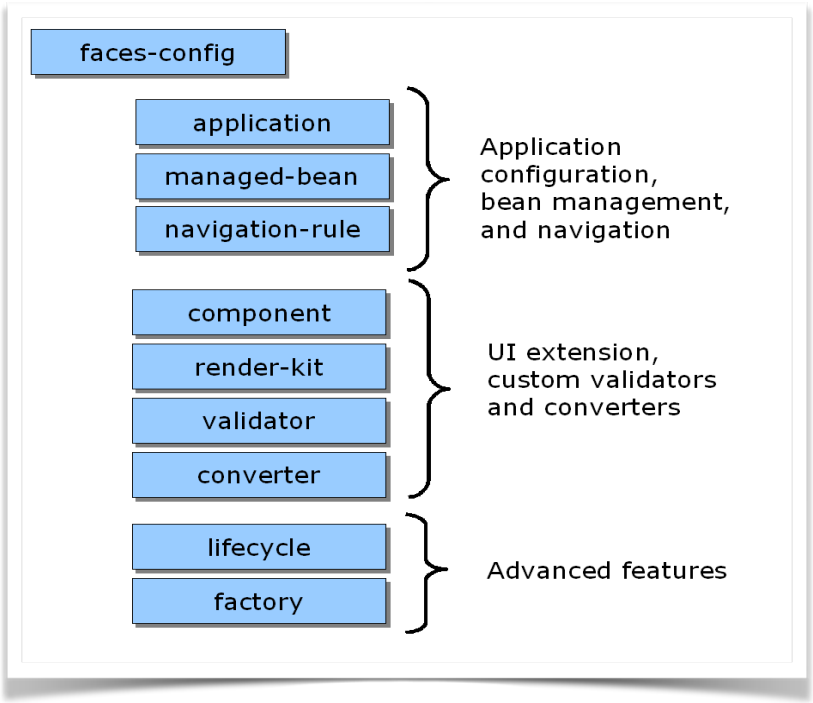
2

JSF MVC



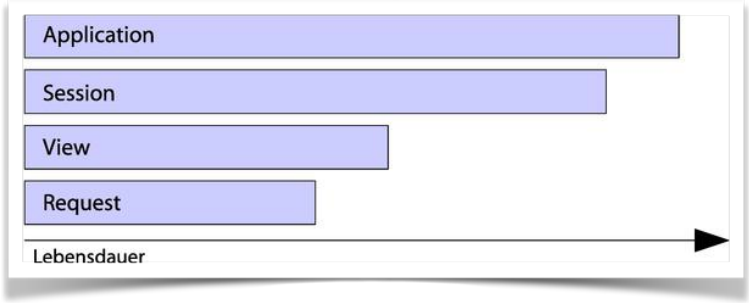
3

JSF CONFIGURATION



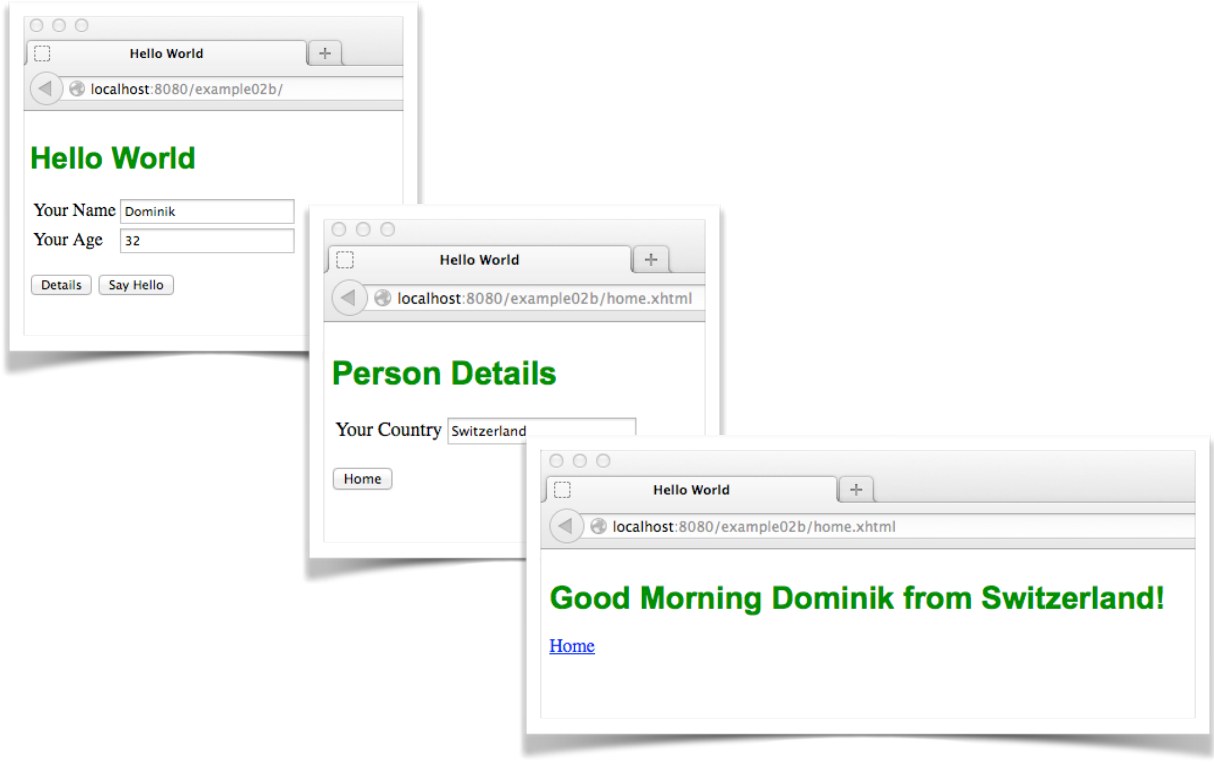
4

JSF LIFECYCLE



- *None-Scope* (`none` , `@NoneScoped`):
  Die *Managed-Bean* wird nicht gespeichert und bei jedem Aufruf neu erstellt.
- *Request-Scope* (`request` , `@RequestScoped`):
  Die *Managed-Bean* lebt für die Zeitdauer einer HTTP-Anfrage.
- *View-Scope* (`view` , `@ViewScoped`):
  Die Lebensdauer der *Managed-Bean* ist an die Ansicht geknüpft, in der sie verwendet wird.
- *Session-Scope* (`session` , `@SessionScoped`):
  Die *Managed-Bean* lebt für die Dauer einer Sitzung, in der der Benutzer mit der Anwendung verbunden ist.
- *Application-Scope* (`application` , `@ApplicationScoped`):
  Für die gesamte Lebensdauer der Anwendung ist nur eine für alle Benutzer gleiche Instanz dieser *Managed-Bean* vorhanden.

SIMPLE EXAMPLE



EXAMPLE02A

@ with annotations

7

HOME.XHTML

```
<h:form id="main">
   <h:outputText value="Hello World" styleClass="header" />
   <table>
      <tr>
         <td><h:outputText value="Your Name" /></td>
         <td><h:inputText id="name" value="#{helloBean.name}"
                 required="true" /></td>
      </tr>
      <tr>
         <td><h:outputText value="Your Age" /></td>
         <td><h:inputText id="age" value="#{helloBean.age}" /></td>
      </tr>
   </table>
   <p />
   <h:commandButton value="Details" action="details" />
   <h:outputText value=" " />
   <h:commandButton value="Say Hello" action="#{helloBean.sayHello}
   />
</h:form>
```

8

DETAILS.XHTML

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

9

HELLO.XHTML

```
<h:form id="main">
   <h:outputText value="#{helloBean.helloText}" styleClass="header"/>
   <br/>
   <h:commandLink value="Home" action="home"/>
</h:form>
```

10

HELLOBEAN.JAVA

```
@ManagedBean(name = "helloBean")
@SessionScoped
public class HelloBean {

   private String name;
   private int age;
   private String helloText;
   @ManagedProperty(value = "#{personDetails}")
   private PersonDetails personDetails;

   public String sayHello() {
      helloText = name + " from " + personDetails.getCountry() + "!";
      if (age < 11) {
         helloText = "Hello " + helloText;
      } else {
         helloText = "Good Morning " + helloText;
      }
      return "hello";
   }
```

11

PERSONDETAILS.JAVA

```
@ManagedBean(name = "personDetails")
@RequestScoped
public class PersonDetails {

   private String country = "Switzerland";

   public String getCountry() {
      return country;
   }

   public void setCountry(String country) {
      this.country = country;
   }
}
```

12

EXAMPLE02B

using faces-config.xml

13

HOME.XHTML

```
<h:form id="main">
   <h:outputText value="Hello World" styleClass="header" />
   <table>
      <tr>
         <td><h:outputText value="Your Name" /></td>
         <td><h:inputText id="name" value="#{helloBean.name}"
                 required="true" /></td>
      </tr>
      <tr>
         <td><h:outputText value="Your Age" /></td>
         <td><h:inputText id="age" value="#{helloBean.age}" /></td>
      </tr>
   </table>
   <p />
   <h:commandButton value="Details" action="details" />
   <h:outputText value=" " />
   <h:commandButton value="Say Hello" action="#{helloBean.sayHello}
   />
</h:form>
```

14

DETAILS.XHTML

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

15

HELLO.XHTML

```
<h:form id="main">
   <h:outputText value="#{helloBean.helloText}" styleClass="header"/>
   <br/>
   <h:commandLink value="Home" action="home"/>
</h:form>
```

16

HELLOBEAN.JAVA

```
public class HelloBean {

   private String name;
   private int age;
   private String helloText;
   private PersonDetails personDetails;

   public String sayHello() {
      helloText = name + " from " + personDetails.getCountry() + "!";
      if (age < 11) {
         helloText = "Hello " + helloText;
      } else {
         helloText = "Good Morning " + helloText;
      }
      return "success";
   }
```


PERSONDETAILS.JAVA

```
public class PersonDetails {

   private String country;

   public String getCountry() {
      return country;
   }

   public void setCountry(String country) {
      this.country = country;
   }
}
```

FACES-CONFIG.XML (0)

```
<faces-config version="2.2"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=„http://xmlns.jcp.org/xml/ns/javaee
           http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">
.
.
.
```

19

FACES-CONFIG.XML (1)

```
<managed-bean>
   <managed-bean-name>helloBean</managed-bean-name>
   <managed-bean-class>ch.hftm.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>
<managed-bean>
   <managed-bean-name>personDetails</managed-bean-name>
   <managed-bean-class>ch.hftm.examples.PersonDetails</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>country</property-name>
      <value>Switzerland</value>
   </managed-property>
</managed-bean>
```

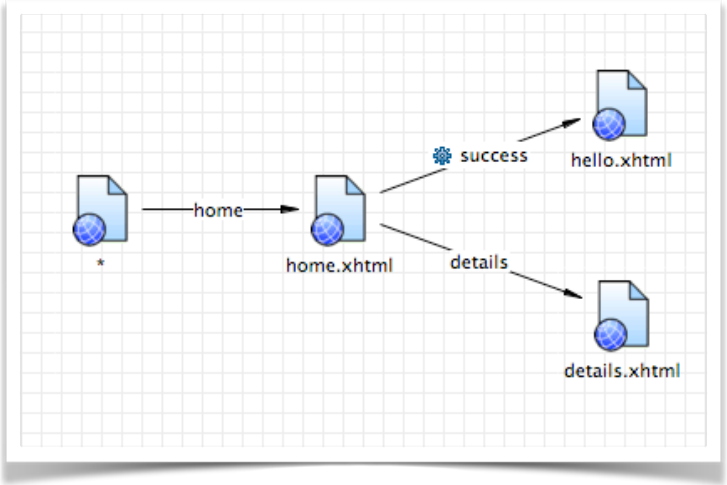
20

FACES-CONFIG.XML (3)

```
<navigation-rule>
   <description>Navigation from the hello page</description>
   <from-view-id>/home.xhtml</from-view-id>
   <navigation-case>
      <from-action>#{helloBean.sayHello}</from-action>
      <from-outcome>success</from-outcome>
      <to-view-id>/hello.xhtml</to-view-id>
   </navigation-case>
   <navigation-case>
      <from-outcome>details</from-outcome>
      <to-view-id>/details.xhtml</to-view-id>
   </navigation-case>
</navigation-rule>
<navigation-rule>
   <description>Navigation from any page</description>
   <from-view-id>/*</from-view-id>
   <navigation-case>
      <from-outcome>home</from-outcome>
      <to-view-id>/home.xhtml</to-view-id>
   </navigation-case>
</navigation-rule>
```

21

NAVIGATION RULE



22

JAVA SERVER FACES

Chapter 03 - Standard Component Tags

1

DETAILED DESCRIPTION

- http://jsfatwork.irian.at/semistatic/standard_components.html

2

JSF-TAG-LIBRARY

Name	alte Namespaces	JSF 2.2	Präfix
HTML-Custom-Tag-Library	http://java.sun.com/jsf/html	http://xmlns.jcp.org/jsf/html	h
Core-Tag-Library	http://java.sun.com/jsf/core	http://xmlns.jcp.org/jsf/core	f
Facelets	http://java.sun.com/jsf/facelets	http://xmlns.jcp.org/jsf/facelets	ui
Composite Components	http://java.sun.com/jsf/composite	http://xmlns.jcp.org/jsf/composite	cc

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

3

BASIC ATTRIBUTES

Attribut	Typ	Beschreibung
id	String	Eindeutiger Bezeichner der Komponente; falls id nicht angegeben ist, wird ein Bezeichner generiert.
immediate	boolean	Ist diese Eigenschaft true gesetzt, werden Eingabe- und Steuerkomponenten schon in Phase 2 des Lebenszyklus abgearbeitet.
value	Object	Der Wert der Komponente direkt als Text oder als Value-Expression
rendered	boolean	Bestimmt die Sichtbarkeit der Komponente
converter	Converter	Bestimmt den Konverter für die Darstellung
validator	Method-Expression	Referenziert eine Validierungsmethode
styleClass	String	CSS-Klassennamen der Komponente (durch Leerzeichen getrennt)
style	String	CSS-Eigenschaften für gerendertes HTML-Element

4

CORE-TAG-LIBRARY

- f:facet
  Fügt eine Komponente mit einer spezifischen, benannten Beziehung zur Elternkomponente hinzu. Die in dem f:facet -Tag verschachtelte Komponente wird angezeigt. Wird in komplexeren Komponenten wie in h:panelGrid oder h:dataTable verwendet.

- f:loadBundle
  Ein Tag zur Lokalisierung, das die Eigenschaften eines Resource-Bundles in einer Map speichert. Der Inhalt der Map kann über Value-Expressions angesprochen werden.

- f:ajax
  Fügt Ajax-Funktionalität in die Ansicht ein.

5

CORE-TAG-LIBRARY

- f:selectItem
  Steht für ein Objekt in der Auswahlliste einer UISelectOne - oder UISelectMany -Komponente.

- f:selectItems
  Steht für eine Objektsammlung in der Auswahlliste einer UISelectOne - oder UISelectMany -Komponente.

6

CORE-TAG-LIBRARY

- f:convertDateTime
  Fügt einen DateTime -Konverter zu einer Komponente hinzu.
- f:converter
  Fügt einen frei wählbaren Konverter zu einer Komponente hinzu.
- f:convertNumber
  Fügt einen Number -Konverter zu einer Komponente hinzu.

7

CUSTOM-TAG-LIBRARY

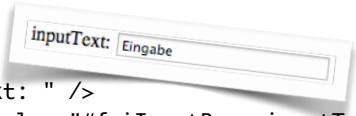
- f:validateDoubleRange
  Fügt einen Validator zu einer Komponente hinzu, der eine Fließkommazahl auf einen bestimmten Wertebereich überprüft
- f:validateLength
  Fügt einen Validator zu einer Komponente hinzu, der die Länge einer Zeichenkette validiert
- f:validateLongRange
  Fügt einen Validator zu einer Komponente hinzu, der eine Ganzzahl auf einen Wertebereich überprüft
- f:validateRegex
  Fügt einen Validator zu einer Komponente hinzu, der eine Zeichenkette mit einem regulären Ausdruck vergleicht
- f:validateRequired
  Fügt einen Validator zu einer Komponente hinzu. Dieser überprüft, ob der Benutzer einen Wert angegeben hat
- f:validator
  Fügt einen im System registrierten, benutzerdefinierten Validator zu einer Komponente hinzu

8

UIINPUT (JAVA)

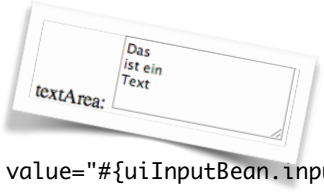
```
@ManagedBean
@SessionScoped
public class UiInputBean {

    private String inputText;
    private String inputSecret;
    private String inputTextarea;

    public String getInputText() {
        return inputText;
    }

    public void setInputText(String inputText) {
        this.inputText = inputText;
    }
...
```


UIINPUT (XHTML)




```
<h:outputText value="inputText: " />
<h:inputText id="inputText" value="#{uiInputBean.inputText}" size="30" maxlength="40" />
```


```
<h:outputText value="inputSecret: " />
<h:inputSecret id="inputSecret" redisplay="false" value="#{uiInputBean.inputSecret}" />
```




```
<h:outputText value="textArea: " />
<h:inputTextarea id="textArea" rows="4" cols="7" value="#{uiInputBean.inputTextarea}" />
```

UIOUTPUT (JAVA)

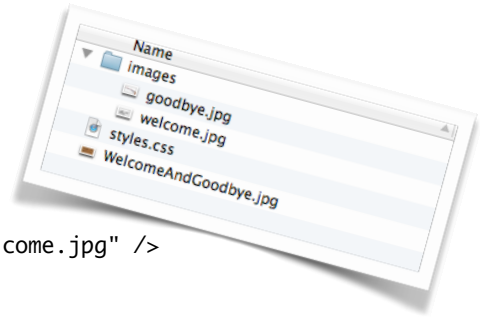
```
@ManagedBean
@SessionScoped
public class UiOutputBean {

    private String string01 = "String01";
    private String string02 = "String02";

    public String getString01() {
        return string01;
    }

    public String getString02() {
        return string02;
    }

}
```


UIOUTPUT (XHTML)

```
<h:outputText value="#{uiOutputBean.string01}" />
```


```
<h:outputLink value="http://hftm.ch">
    <h:outputText value="HFTM" />
</h:outputLink>
```

UIGRAPHIC (XHTML)




```
<h:graphicImage id="welcome" url="/resources/images/welcome.jpg" />


<h:graphicImage id="welcomeAndgoodbye" name="WelcomeAndGoodbye.jpg" />


<h:graphicImage id="goodbye" library="images" name="goodbye.jpg" />
```


UIDATA (JAVA)

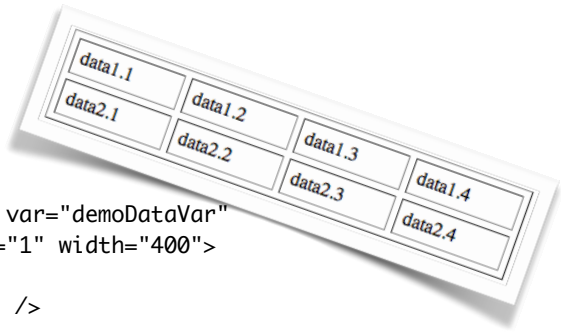
```
@ManagedBean
@SessionScoped
public class UiDataBean {

    private List<DemoData> demoDataList = new ArrayList<DemoData>();

    public List<DemoData> getDemoDataList() {
        if (demoDataList.size() == 0) {
            DemoData demoData01 = new DemoData("data1.1", "data1.2", "data1.3", "data1.4");
            DemoData demoData02 = new DemoData("data2.1", "data2.2", "data2.3", "data2.4");

            demoDataList.add(demoData01);
            demoDataList.add(demoData02);
        }
        return demoDataList;
    }
}
```

UIDATA (JAVA)

```
public class DemoData {
    private String data1;
    private String data2;
    private String data3;
    private String data4;

    public DemoData(String data1, String data2, String data3, String data4) {
        this.data1 = data1;
        this.data2 = data2;
        this.data3 = data3;
        this.data4 = data4;
    }

    public String getData1() {
        return data1;
    }

    public void setData1(String data1) {
        this.data1 = data1;
    }
...
```

15

UIDATA (XHTML)

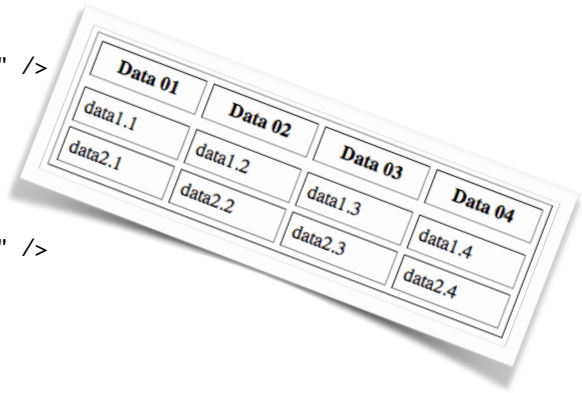



```
<h:dataTable value="#{uiDataBean.demoDataList}"  var="demoDataVar"
          cellpadding="5" cellspacing="5" border="1" width="400">
    <h:column>
        <h:outputText value="#{demoDataVar.data1}" />
    </h:column>
    <h:column>
        <h:outputText value="#{demoDataVar.data2}" />
    </h:column>
    <h:column>
        <h:outputText value="#{demoDataVar.data3}" />
    </h:column>
    <h:column>
        <h:outputText value="#{demoDataVar.data4}" />
    </h:column>
</h:dataTable>
```

16

UIDATA (XHTML)

```
<h:dataTable value="#{uiDataBean.demoDataList}" var="demoDataVar"
          cellpadding="5" cellspacing="5" border="1" width="400">
    <h:column>
        <f:facet name="header">
            <h:outputText value="Data 01" />
        </f:facet>
        <h:outputText value="#{demoDataVar.data1}" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Data 02" />
        </f:facet>
        <h:outputText value="#{demoDataVar.data2}" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Data 03" />
        </f:facet>
        <h:outputText value="#{demoDataVar.data3}" />
    </h:column>
...
```

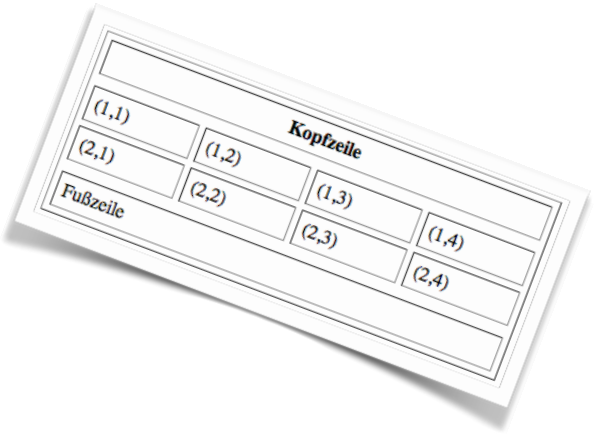


ROWSELECTION (XHTML)

```
XHTML:
<h:dataTable value="#{uiDataBean.demoDataList}"  var="demoDataVar">
    <h:column>
        <h:commandLink action="#{uiDataBean.selectRow(demoDataVar)}"
        value="#{demoDataVar.data1}" />
    </h:column>
...


JAVA:
@ManagedBean
@SessionScoped
public class UiDataBean {

    private List<DemoData> demoDataList = new ArrayList<DemoData>();
    private DemoData selectetObject;


    public String selectRow(DemoData selectetObject){
        this.selectetObject = selectetObject;
        return null;
    }
...
```

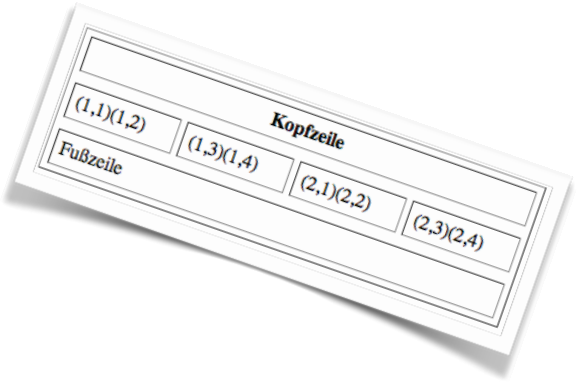
UIPANEL (XHTML)

```
<h:panelGrid columns="4" cellpadding="5" cellspacing="5" border="1"
    width="400" styleClass="table-background"
    columnClasses="table-odd-column, table-even-column"
    footerClass="table-footer" headerClass="page-header">
    <f:facet name="header">
        <h:outputText value="Kopfzeile" />
    </f:facet>
    <h:outputText value="(1,1)" />
    <h:outputText value="(1,2)" />
    <h:outputText value="(1,3)" />
    <h:outputText value="(1,4)" />
    <h:outputText value="(2,1)" />
    <h:outputText value="(2,2)" />
    <h:outputText value="(2,3)" />
    <h:outputText value="(2,4)" />
    <f:facet name="footer">
        <h:outputText value="Fußzeile" />
    </f:facet>
</h:panelGrid>
```



UIPANEL (XHTML)

```
<h:panelGrid columns="4" cellpadding="5" cellspacing="5" border="1"
    width="400" styleClass="table-background"
    columnClasses="table-odd-column, table-even-column"
    footerClass="table-footer" headerClass="page-header">
    <f:facet name="header">
        <h:outputText value="Kopfzeile" />
    </f:facet>
    <h:panelGroup>
        <h:outputText value="(1,1)" />
        <h:outputText value="(1,2)" />
    </h:panelGroup>
    <h:panelGroup>
        <h:outputText value="(1,3)" />
        <h:outputText value="(1,4)" />
    </h:panelGroup>
    <h:panelGroup>
        <h:outputText value="(2,1)" />
        <h:outputText value="(2,2)" />
    </h:panelGroup>
    <h:panelGroup>
        <h:outputText value="(2,3)" />
        <h:outputText value="(2,4)" />
    </h:panelGroup>
    <f:facet name="footer">
        <h:outputText value="Fußzeile" />
```



UISELECTBOOLEAN (JAVA)

```
@ManagedBean
@SessionScoped
public class UiSelectBooleanBean {

    private boolean booleanValue01;

    public boolean isBooleanValue01() {
        return booleanValue01;
    }

    public void setBooleanValue01(boolean booleanValue01) {
        this.booleanValue01 = booleanValue01;
    }
```


UISELECTBOOLEAN (XHTML)

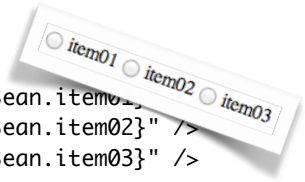

```
<h:outputText value="BooleanValue01: " />
<h:selectBooleanCheckbox id="booleanValue01" value="#{uiSelectBooleanBean.booleanValue01}"
/>
```

UISELECTONE (JAVA)

```
@ManagedBean
@SessionScoped
public class UiSelectOneBean {

   private String item01 = "itemValue01";
   private String item02 = "itemValue02";
   private String item03 = "itemValue03";
   private String selectionRadio;
   private String selectionListbox;
   private String selectionMenu;

   public String getItem01() {
      return item01;
   }

   public void setItem01(String item01) {
      this.item01 = item01;
   }
...
```

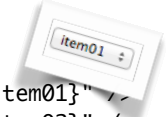

UISELECTONE (XHTML)

```
<h:selectOneRadio id="itemRadio" required="true"
   value="#{uiSelectOneBean.selectionRadio}">
   <f:selectItem itemLabel="item01" itemValue="#{uiSelectOneBean.item01}" />
   <f:selectItem itemLabel="item02" itemValue="#{uiSelectOneBean.item02}" />
   <f:selectItem itemLabel="item03" itemValue="#{uiSelectOneBean.item03}" />
</h:selectOneRadio>
```




```
<h:selectOneListbox id="itemListbox" required="true"
   value="#{uiSelectOneBean.selectionListbox}">
   <f:selectItem itemLabel="item01" itemValue="#{uiSelectOneBean.item01}" />
   <f:selectItem itemLabel="item02" itemValue="#{uiSelectOneBean.item02}" />
   <f:selectItem itemLabel="item03" itemValue="#{uiSelectOneBean.item03}" />
</h:selectOneListbox>
```


```
<h:selectOneMenu id="itemMenu" required="true"
   value="#{uiSelectOneBean.selectionMenu}">
   <f:selectItem itemLabel="item01" itemValue="#{uiSelectOneBean.item01}" />
   <f:selectItem itemLabel="item02" itemValue="#{uiSelectOneBean.item02}" />
   <f:selectItem itemLabel="item03" itemValue="#{uiSelectOneBean.item03}" />
</h:selectOneMenu>
```

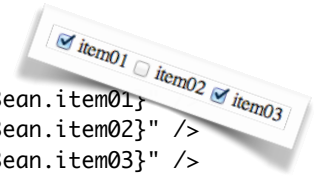


UISELECTMANY (JAVA)

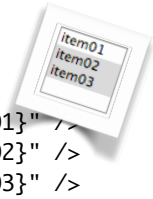
```
@ManagedBean
@SessionScoped
public class UiSelectManyBean {

    private String item01 = "itemValue01";
    private String item02 = "itemValue02";
    private String item03 = "itemValue03";
    private List<String> selectionCheckboxList = new ArrayList<String>();
    private List<String> selectionListboxList = new ArrayList<String>();
    private List<String> selectionMenuList = new ArrayList<String>();

    public String getItem01() {
        return item01;
    }

    public void setItem01(String item01) {
        this.item01 = item01;
    }
...
```


UISELECTMANY (XHTML)

```
<h:selectManyCheckbox id="itemCheckbox"
    value="#{uiSelectManyBean.selectionCheckboxList}">
    <f:selectItem itemLabel="item01" itemValue="#{uiSelectManyBean.item01}" />
    <f:selectItem itemLabel="item02" itemValue="#{uiSelectManyBean.item02}" />
    <f:selectItem itemLabel="item03" itemValue="#{uiSelectManyBean.item03}" />
</h:selectManyCheckbox>
```




```
<h:selectManyListbox id="itemListbox"
    value="#{uiSelectManyBean.selectionListboxList}">
    <f:selectItem itemLabel="item01" itemValue="#{uiSelectManyBean.item01}" />
    <f:selectItem itemLabel="item02" itemValue="#{uiSelectManyBean.item02}" />
    <f:selectItem itemLabel="item03" itemValue="#{uiSelectManyBean.item03}" />
</h:selectManyListbox>
```

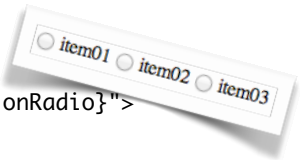



```
<h:selectManyMenu id="itemMenu"
    value="#{uiSelectManyBean.selectionMenuList}">
    <f:selectItem itemLabel="item01" itemValue="#{uiSelectManyBean.item01}" />
    <f:selectItem itemLabel="item02" itemValue="#{uiSelectManyBean.item02}" />
    <f:selectItem itemLabel="item03" itemValue="#{uiSelectManyBean.item03}" />
</h:selectManyMenu>
```

UISELECTITEMS (JAVA)

```
@ManagedBean
@SessionScoped
public class UiSelectItemsBean {

    private List<SelectItem> selectItems;
    private String selectionRadio;
    private String selectionRadioPlus;
    private String selectionListbox;
    private String selectionMenu;

    public List<SelectItem> getSelectItems() {
        if (selectItems == null) {
            selectItems = new ArrayList<SelectItem>();
            selectItems.add(new SelectItem("itemValue01", "item01"));
            selectItems.add(new SelectItem("itemValue02", "item02"));
            selectItems.add(new SelectItem("itemValue03", "item03"));
        }
        return selectItems;
    }
...
```


UISELECTITEMS (XHTML)




```
<h:selectOneRadio id="itemRadio" value="#{uiSelectItemsBean.selectionRadio}">
    <f:selectItems value="#{uiSelectItemsBean.selectItems}" />
</h:selectOneRadio>
```


```
<h:selectOneListbox id="itemListbox" value="#{uiSelectItemsBean.selectionListbox}">
    <f:selectItems value="#{uiSelectItemsBean.selectItems}" />
</h:selectOneListbox>
```


```
<h:selectOneMenu id="itemMenu" value="#{uiSelectItemsBean.selectionMenu}">
    <f:selectItems value="#{uiSelectItemsBean.selectItems}" />
</h:selectOneMenu>
```

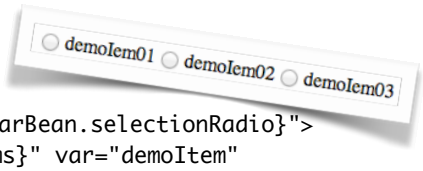
UISELECTITEMSVAR (JAVA)

```
@ManagedBean
@SessionScoped
public class UiSelectItemsVarBean {

    private List<DemoItem> demoItems;
    private String selectionRadio;
    private String selectionRadioPlus;
    private String selectionListbox;
    private String selectionMenu;

    public UiSelectItemsVarBean() {
        if (demoItems == null) {
            demoItems = new ArrayList<DemoItem>();
            demoItems.add(new DemoItem("demoIem01", 01));
            demoItems.add(new DemoItem("demoIem02", 02));
            demoItems.add(new DemoItem("demoIem03", 03));
        }
    }
..
```


UISELECTITEMSVAR (JAVA)

```
public class DemoItem {
    private String name;
    private int value;

    public DemoItem(String name, int value) {
        this.name = name;
        this.value = value;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
...
```

UISELECTITEMSVAR (XHTML)




```
<h:selectOneRadio id="itemRadio" value="#{uiSelectItemsVarBean.selectionRadio}">
   <f:selectItems value="#{uiSelectItemsVarBean.demoItems}" var="demoItem"
      itemLabel="#{demoItem.name}" itemValue="#{demoItem.value}" />
</h:selectOneRadio>
```


```
<h:selectOneListbox id="itemListbox" value="#{uiSelectItemsVarBean.selectionListbox}">
   <f:selectItems value="#{uiSelectItemsVarBean.demoItems}" var="demoItem"
      itemLabel="#{demoItem.name}" itemValue="#{demoItem.value}" />
</h:selectOneListbox>
```


```
<h:selectOneMenu id="itemMenu" value="#{uiSelectItemsVarBean.selectionMenu}">
   <f:selectItems value="#{uiSelectItemsVarBean.demoItems}" var="demoItem"
      itemLabel="#{demoItem.name}" itemValue="#{demoItem.value}" />
</h:selectOneMenu>
```

31

JAVA SERVER FACES

Chapter 3.2 - Enumeration as selectItems

32

MODEL GENDER.JAVA (ENUM)

```
public enum Gender {
    MALE("male"), FEMALE("female");

    private String label;

    private Gender(String label) {
        this.label = label;
    }

    public String getLabel() {
        return label;
    }
}
```

33

GENDERBEAN.JAVA

```
@ManagedBean
@RequestScoped
public class genderBean  {

    private Gender gender;

    public Gender[] getGenders() {
        return Gender.values();
    }

    public Gender getGender() {
        return gender;
    }

    public void setGender(Gender gender) {
        this.gender = gender;
    }

}
```

34

LOCALES-PARAMETER

texts_en.properties

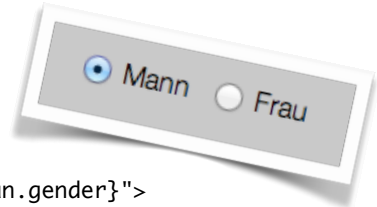
```
male=Male
female=Female
```

texts_de.properties

```
male=Mann
female=Frau
```

35

UISELECTONE (XHTML)




```
<h:selectOneRadio id="selectGender" value="#{genderBean.gender}">
    <f:selectItems value="#{genderBean.genders}" var="gender"
        itemValue="#{gender}" itemLabel="#{texts[gender.label]}" />
</h:selectOneRadio>
```

36

JAVA SERVER FACES

Chapter 04 - Facelets Templating

1

NAMESPACE

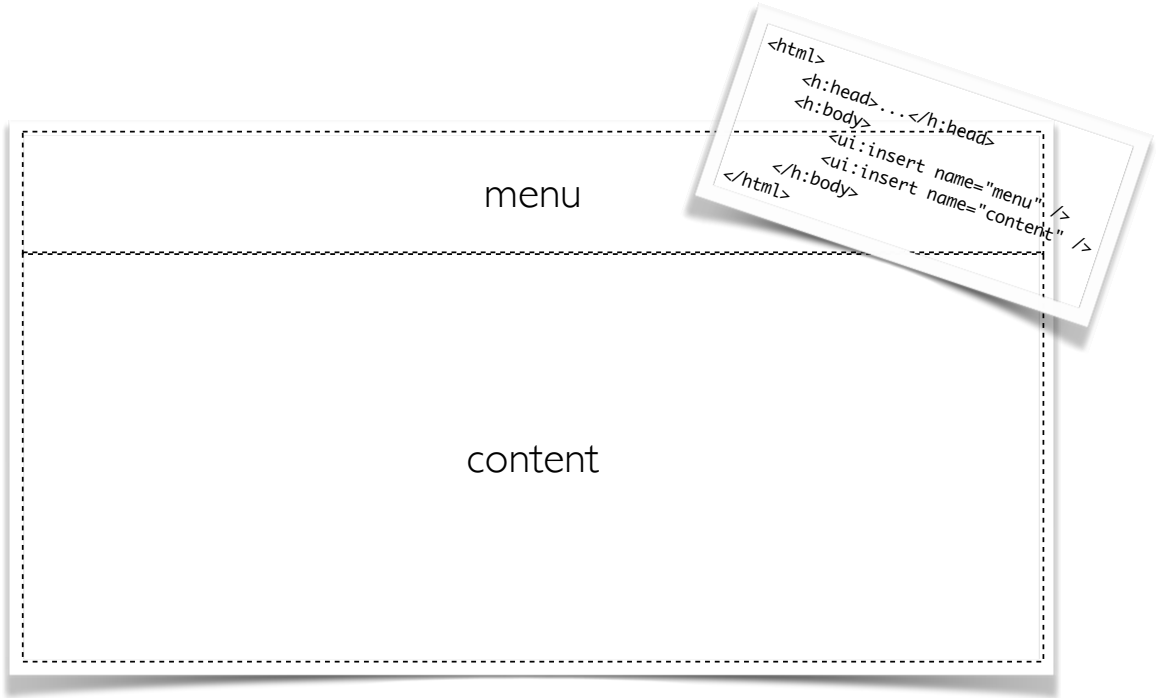
```
xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
```

2

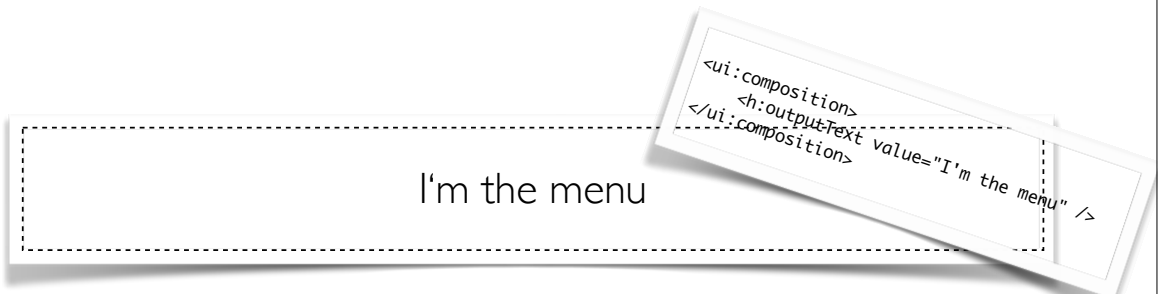
FACELETS TEMPLATING TAGS

- ui:insert
  fügt Inhalt in ein Template ein
- ui:define
  definiert den Inhalt welcher in eine Seite eingefügt wird
- ui:include
  fügt „externen" Inhalt innerhalb eines ui:define tag's ein
- ui:composition
  wiederverwendbarer code-Teil welcher von einem ui:insert eingefügt werden kann (code ausserhalb diese Tags wird nicht berücksichtigt)
- ui:param
  wird verwendet um Parameter an ein eingefügtes File zu übergeben

THE TEMPLATE



THE MENU



5

THE CONTENT

6

CLIENT PAGE (ALL TOGETHER)

7

OVERVIEW

template.xhtml

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

home.xhtml

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

menu.xhtml

```
<ui:composition>
      <h:outputText value="I'm the menu" />
</ui:composition>
```

content.xhtml

```
<ui:composition>
      <h:outputText value="I'm the content" />
</ui:composition>
```

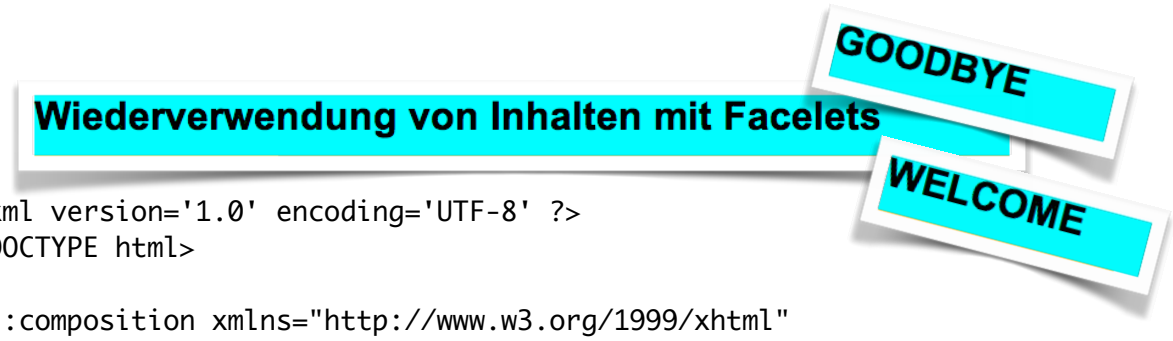
8

EXAMPLE04

TEMPLATE

```
<div class="header">
    <ui:insert name="header" id="header">
        <ui:include src=„header.xhtml“ />
    </ui:insert>
</div>

<div class="menu">
    <ui:insert name="menu" id="menu">
        <ui:include src="menu.xhtml" />
    </ui:insert>
</div>

<div class="content">
    <ui:insert name="content" id="content" />
</div>
```

HEADER




```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html>

<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html"
   xmlns:f="http://xmlns.jcp.org/jsf/core"
   xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

   <h:outputText value="#{pageTitle}" styleClass="title" />

</ui:composition>
```

11

MENU


```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html>

<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html"
   xmlns:f="http://xmlns.jcp.org/jsf/core"
   xmlns:ui="http://xmlns.jcp.org/jsf/facelets">


   <h:form class="menu">
      <ul>
         <li><h:commandLink value="Welcome" action="welcome" /></li>
         <li><h:commandLink value="Goodbye" action="goodbye" /></li>
      </ul>
   </h:form>

</ui:composition>
```

12

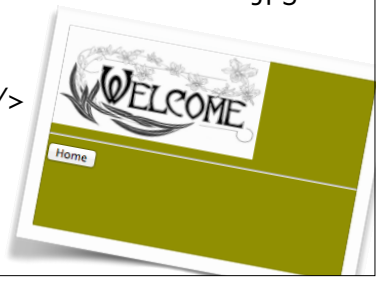
HOME

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html>

<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html"
   xmlns:f="http://xmlns.jcp.org/jsf/core"
   xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
   template="./template.xhtml">


   <ui:param name="pageTitle"
      value="Wiederverwendung von Inhalten mit Facelets" />

   <ui:define name="content">
      <h:outputText
         value="Textausgabe Facelets …/>
   </ui:define>

</ui:composition>
```

CLIENT PAGE (HOME.XHTML)

WELCOME

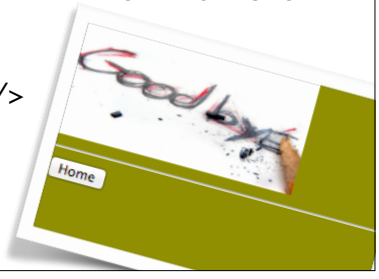
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html>

<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html"
   xmlns:f="http://xmlns.jcp.org/jsf/core"
   xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
   template="./template.xhtml">

   <ui:param name="pageTitle" value="WELCOME" />

   <ui:define name="content">
      <h:form>
         <h:graphicImage id="welcome" library="images" name="welcome.jpg"
            width="200" height="100" />
         <hr />
         <h:commandButton value="Home" action="home" />
      </h:form>
   </ui:define>

</ui:composition>
```



15

GOODBYE

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html>

<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html"
   xmlns:f="http://xmlns.jcp.org/jsf/core"
   xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
   template="./template.xhtml">

   <ui:param name="pageTitle" value="GOODBYE" />

   <ui:define name="content">
      <h:form>
         <h:graphicImage id="goodbye" library="images" name="goodbye.jpg"
            width="200" height="100" />
         <hr />
         <h:commandButton value="Home" action="home" />
      </h:form>
   </ui:define>

</ui:composition>
```



16

JAVA SERVER FACES

Chapter 05 - Internationalization

1

SUPPORT MULTIPLE LOCALES

2

CONFIGURING....

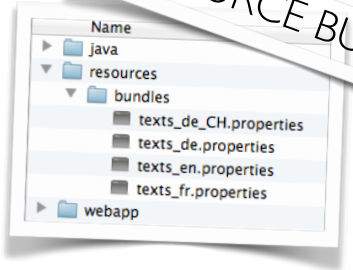
```
<faces-config ...>
    <application>

        <locale-config>
            <default-locale>de</default-locale>
            <supported-locale>en</supported-locale>
            <supported-locale>fr</supported-locale>
            <supported-locale>de_CH</supported-locale>
        </locale-config>

        ..................
            ..................
            ..................
        ..................

    </application>
</faces-config>
```

LOCALES

CONFIGURING....

```
<faces-config ...>
    <application>

        ..................
            ....................................
            ....................................
            ....................................
            ....................................
        ..................

        <resource-bundle>
            <base-name>bundles.texts</base-name>
            <var>texts</var>
        </resource-bundle>

    </application>
</faces-config>
```

RESOURCE BUNDLES



RESOURCE BUNDLES

texts_en.properties

```
home=Home

infoText=Internationalization of the application texts

titleDefault=Internationalization
titleWelcome=Welcome
titleGoodbye=Goodbye
```

texts_de.properties

```
home=Hauptseite

infoText=Internationalisierung der Anwendungstexte

titleDefault=Internationalisierung
titleWelcome=Willkommen
titleGoodbye=Auf Wiedersehen
```

5

USING RESOURCE BUNDLES

```
<ui:define name="header">
    <ui:include src="header.xhtml">
        <ui:param name="pageTitle" value="#{texts.titleWelcome}" />
    </ui:include>
</ui:define>

<ui:define name="content">
    <h:form>
        <h:graphicImage id="welcome" library="images" name="welcome.jpg"
            width="200" height="100" />
        <br />
        <h:commandButton value="#{texts.home}" action="home" />
    </h:form>
</ui:define>
```

6

LOCALES-PARAMETER

texts_en.properties

```
welcomeText=Welcome {0}!
```

texts_de.properties

```
welcomeText=Willkommen {0}!
```


USING LOCALES-PARAMETER


```
<h:panelGrid columns="2">

    <h:inputText value="#{welcomeBean.name}" />
    <h:commandButton value="${texts.save}" />

    <h:outputFormat value="#{texts.welcomeText}" rendered="#{welcomeBean.name != null}">
        <f:param value="#{welcomeBean.name}" />
    </h:outputFormat>

</h:panelGrid>
```

LANGAGE SELECTOR

9

LANGUAGEBEAN.JAVA

```
@ManagedBean
@SessionScoped
public class LanguageBean implements Serializable{

    private Locale locale = FacesContext.getCurrentInstance().getViewRoot().getLocale();

    public String english() {
        setLocale(Locale.ENGLISH);
        return null;
    }

    public String german() {
        setLocale(Locale.GERMAN);
        return null;
    }

    public String french() {
        setLocale(Locale.FRENCH);
        return null;
    }
    public void setLocale(Locale locale) {
        this.locale = locale;
        FacesContext.getCurrentInstance().getViewRoot().setLocale(locale);
    }

    public Locale getLocale() {
        return locale;
    }
}
```

10

XHTML SELECTOR

```
<h:panelGrid columns="3">
    <h:commandLink action="#{languageBean.english}">
        <h:graphicImage library="icons" name="en.gif"
            styleClass="languageIcon" />
    </h:commandLink>
    <h:commandLink action="#{languageBean.german}">
        <h:graphicImage library="icons" name="de.gif"
            styleClass="languageIcon" />
    </h:commandLink>
    <h:commandLink action="#{languageBean.french}">
        <h:graphicImage library="icons" name="fr.gif"
            styleClass="languageIcon" />
    </h:commandLink>
</h:panelGrid>
```

TEMPLATE.XHTML

```
<f:view locale="#{languageBean.locale}">
    <h:body>

        <div class="header">
            <…>
        </div>


        <div class="menu">
            <…>
        </div>

        <div class="content">
            <…>
        </div>

    </h:body>
</f:view>
```

BUNDLES IN MANAGED BEANS


```
public static String getResourceText(String bundleName, String key, Object... args) {
    String text;
    try {
        FacesContext context = FacesContext.getCurrentInstance();
        Application app = context.getApplication();
        ResourceBundle bundle = app.getResourceBundle(context, bundleName);
        text = bundle.getString(key);
    } catch (MissingResourceException e) {
        return "???" + key + "???";
    }
    if (args != null)
        text = MessageFormat.format(text, args);
    return text;
}
```

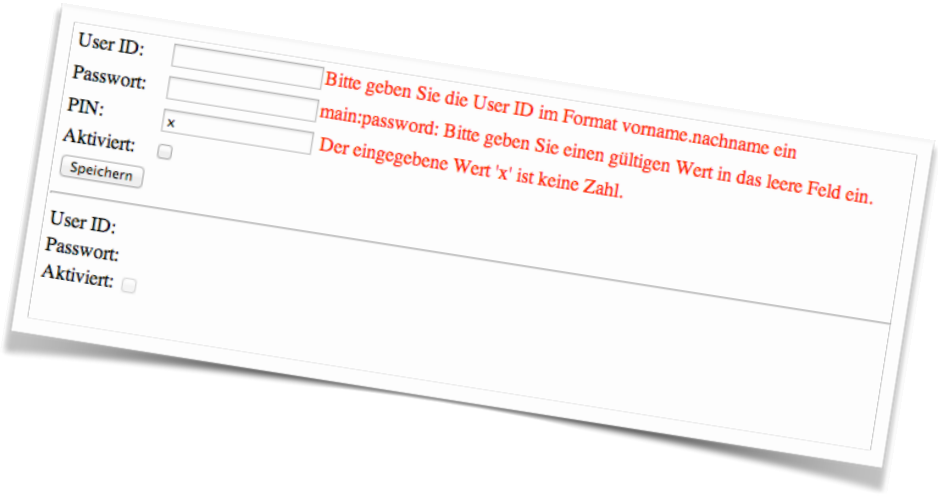
13

JAVA SERVER FACES

Chapter 06 - Messages

1

CUSTOMIZED MESSAGES



2

DISPLAYING MESSAGES

Messages for „userId“

```
<h:message for="userId"
          errorClass="errorMessage" infoClass="infoMessage"
          showDetail="true" showSummary="false" />
```

All messages for this page

```
<h:messages globalOnly="false"
          errorClass="errorMessage" infoClass="infoMessage"
          layout="table" />
```


OVERRIDING STANDARD

1. In the input component

```
<h:inputText id="userId" value="#{loginBean.userId}" required="true"
    requiredMessage="Please enter your user id" />
```

2. With texts from bundle

```
<h:inputText id="userId" value="#{loginBean.userId}" required="true"
    requiredMessage="#{texts.enterId}" />
```

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

OVERRIDING STANDARD

3. By a custom message bundle

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


MESSAGE BUNDLES

messages_en.properties

```
javax.faces.component.UIInput.REQUIRED=Please enter a value
javax.faces.component.UIInput.REQUIRED_detail={0}: Please insert a valid value into the empty field
javax.faces.converter.IntegerConverter.INTEGER=The given input value ''{0}'' is not an integer.
javax.faces.converter.IntegerConverter.INTEGER_detail={2}: Please insert an integer value
```

messages_de.properties

```
javax.faces.component.UIInput.REQUIRED=Dieses Feld darf nicht leer sein.
javax.faces.component.UIInput.REQUIRED_detail={0}: Bitte geben Sie einen gültigen Wert in das Feld ein
javax.faces.converter.IntegerConverter.INTEGER=Der eingegebene Wert ''{0}'' ist keine Zahl.
javax.faces.converter.IntegerConverter.INTEGER_detail={2}: Bitte geben Sie eine ganze Zahl
```

STANDARD JSF MESSAGES

7

APPLICATION MESSAGES

warnings or error messages for:

- ManagedBean Exceptions
- Application Logic Exceptions
- System Errors
- Connection Errors

8

APPLICATION MESSAGES

1. FacesMessages

```
public void setUserId(String userId) {
    if (userId.matches(pattern)){
        this.userId = userId;
    } else {
        FacesMessage message = new FacesMessage(
               FacesMessage.SEVERITY_ERROR,"FEHLER_TEXT_SUMMARY","FEHLER_TEXT_DETAIL");
        FacesContext context = FacesContext.getCurrentInstance();
        context.addMessage(null, message);
        this.userId = "";
    }
}
```


APPLICATION MESSAGES

2. MessageFactory

```
public class MessageFactory {

    public static FacesMessage getMessage(
            FacesMessage.Severity severity, String key, Object... params) {
        String summary = "???" + key + "???";
        String detail = null;
        try {
            FacesContext context = FacesContext.getCurrentInstance();
            String name = context.getApplication().getMessageBundle();
            if (name == null) {
                name = FacesMessage.FACES_MESSAGES;
            }
            Locale locale = context.getViewRoot().getLocale();
            ResourceBundle bundle = ResourceBundle.getBundle(name, locale);
            summary = MessageFormat.format(bundle.getString(key), params);
            detail = MessageFormat.format(bundle.getString(key + "_detail"), params);
        } catch (MissingResourceException e) {
        }
        return new FacesMessage(severity, summary, detail);
    }

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

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

APPLICATION MESSAGES

using the MessageFactory

```
} catch (UserNotFoundException e) {
   MessageFactory.error("ch.hftm.example.USER_NOT_FOUND");
}
```

messages_de.properties

```
ch.hftm.example.USER_NOT_FOUND=User konnte nicht angemeldet werden
```

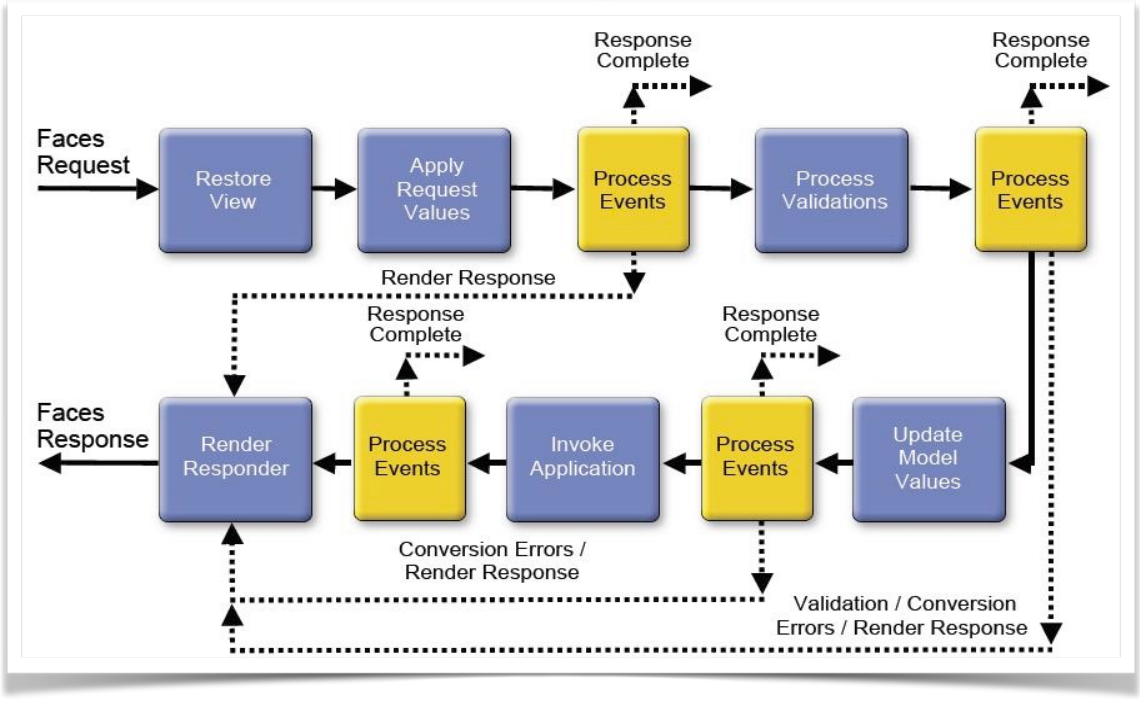
11

JAVA SERVER FACES

Chapter 07 - JSF Lifecycle

1

LEBENSZYKLUS



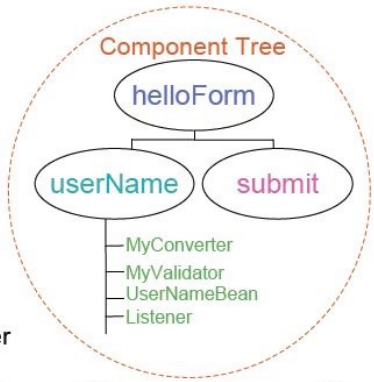
2

REQUEST LIFECYCLE

1. Restore View: Wiederherstellung des Komponentenbaumes
2. Apply Requests: Übernahme der Daten der Anfrage
3. Process Validations: Verarbeitung der Validierung
4. Update Model Values: Aktualisierung der Managed-Bean Daten
5. Invoke Applikation: Aufruf der Applikation
6. Render Response: Darstellen (rendern) der Antwort

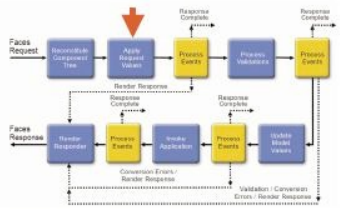
RESTORE VIEW

- Wird ausgelöst, wenn eine Seite vom Client angefordert wird.
- Der Component Tree wird aus der zugehörigen JSF Seite erzeugt, Event Handlers und Validators werden verknüpft.
  - Sofern es sich um ein initiales Request handelte, wird ein leeres View der Seite erzeugt und im weiteren Verlauf gefüllt.
  - Ansonsten existiert bereits ein korrespondierendes View, welches mit Hilfe von Zustandsinformationenen, die auf dem Client oder Server gespeichert sind, wieder hergestellt wird.
- Entspricht der Verschachtelung der Tags, wie sie auf der JSP Seite eingetragen wurden, ist also die serverseitige Repräsentation einer JSF-Seite.
- Speichert den View in den *Faces-Context*.



APPLY REQUEST VALUES

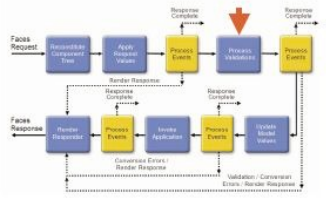
- Nach dem Aufbau des Komponentenbaums holt sich jede Komponente den neuen Wert aus dem Request.
  - Wert wird **lokal** in der Komponente gespeichert.
  - Mögliche Fehler werden im *FacesContext* gespeichert.
- Außerdem finden hier Datentypkonvertierungen der Werte statt.
- Registrierte Event-Listener werden in dieser Phase bereits benachrichitgt.



5

PROCESS VALIDATIONS

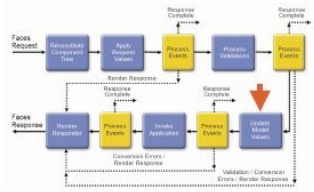
- Alle Validatoren, welche in der Apply Request Phase auf den Komponenten registriert wurden, werden ausgeführt.
- Überprüfung, ob lokal gespeicherte Komponentenwerte den Regeln der individuell gesetzten Validatoren entsprechen.
- Falls Validation fehlschlägt,
  - Fehlermeldung wird im *FacesContext* gespeichert
  - Es wird direkt zur Response Phase verzweigt.
  - Beispiel: Wenn eingegebene Zahl zwischen 0 und 10 liegen sollte und dieses nicht erfüllt ist.



6

UPDATE MODEL VALUES

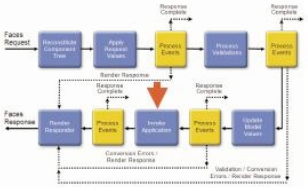
- Die JSF-Implementierung durchwandert den Component Tree und setzt die korrespondierenden serverseitigen Objekteigenschaften auf die lokalen Werte der Komponenten.
  - Updaten der Bean-Eigenschaften, die auf einen Eingabewert der Komponente zeigen.



7

INVOKE APPLICATION

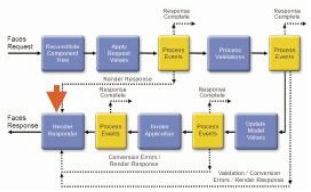
- In dieser Phase wickelt die JSF-Implementierung alle Events auf Anwendungsebene ab, wie z.B. das Abschicken eines Formulars oder die Verlinkung zu einer anderen Seite.



8

RENDER RESPONSE

- Durch die *encode*-Methode wird der Komponentenbaum, der in *FacesContext* gespeichert ist, gerendert.
- Wird nun durch alle Komponenten iteriert, stellt sich so Stück für Stück die Antwortseite zusammen.
- Traten in irgendwelchen anderen Phasen Fehler auf, wird die alte Seite neu gerendert mit entsprechend generierten Fehlermeldungen.



There are four forms of JSF validation:

1. Built-in validation components
2. Application level validations
3. Custom validation components using Validator interface
4. Validation methods in backing beans

JAVA SERVER FACES

Chapter 08 - Validation

1

VALIDATION

- Validierung versteht sich als Test auf Plausibilität, bei dem ein konkreter Wert darauf geprüft wird, ob er zu einem bestimmten Datentyp gehört oder in einem vorgegebenen Wertebereich oder einer vorgegebenen Wertemenge liegt.
- Viele Programmfehler und Sicherheitsprobleme sind auf fehlende Plausibilisierung von Eingabewerten zurückzuführen.
- Für die Validierung gilt die goldene Regel: never trust the user.
- Die Validierung von Werten kann an verschiedenen Punkten der Lebenszeit einer Software stattfinden

2

JSF-VALIDATION

`f:validateLength` :
Dieser Validator überprüft die Länge einer Zeichenkette basierend auf den Werten der Attribute minimum und maximum . Das folgende Beispiel erlaubt die Eingabe einer Zeichenkette mit der minimalen Länge drei und der maximalen Länge sieben:

```
<h:inputText value="#{managedBean.property}">
   <f:validateLength minimum="3" maximum="7" />
</h:inputText>
```

`f:validateLongRange` :
Für die Überprüfung, ob eine Ganzzahl in einem gewissen Bereich liegt, bietet JSF den LongRange Validator an. Auch hier sind die gültigen Attribute minimum und maximum . Ein Beispiel für die Überprüfung einer Zahl im Bereich 0-1000:

```
<h:inputText value="#{managedBean.property}">
   <f:validateLongRange minimum="0" maximum="1000" />
</h:inputText>
```

`f:validateDoubleRange` :
Für die Überprüfung, ob eine Fliesskommazahl in einem gewissen Bereich liegt, bietet JSF den DoubleRange Validator an. Wieder sind die gültigen Attribute minimum und maximum . Ein Beispiel für die Überprüfung einer Zahl im Bereich 0.0 - 1.0 :

```
<h:inputText value="#{managedBean.property}">
   <f:validateDoubleRange minimum="0.0" maximum="1.0" />
</h:inputText>
```

3

JSF-VALIDATION

`f:validateRegex` :
Dieser Validator überprüft, ob der Zeichenkettenwert dem im Attribut pattern angegebenen regulären Ausdruck entspricht. Eine detaillierte Erklärung zu regulären Ausdrücken in Java finden Sie in der Dokumentation zur Java-API. Hier ein kleines Beispiel, bei dem der Wert einer Komponente aus einem Großbuchstaben, gefolgt von einer beliebigen Folge von Buchstaben bestehen muss:

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

`f:validateRequired` :
Dieser Validator überprüft, ob der Benutzer überhaupt einen Wert angegeben hat.

```
<h:inputText value="#{backingBean.property}">
   <f:validateRequired/>
</h:inputText>
```

4

VALIDATOR METHODS

loginBean.java

```
public void yesNoValidate(FacesContext ctx, UIComponent comp, Object value) throws ValidatorException {
    if (value instanceof String) {
        String strValue = (String) value;
        if (!(strValue.equals("ja")) && !(strValue.equals("nein"))) {
            throw new ValidatorException(new FacesMessage("messageText", null));
        }
    } else {
        throw new ValidatorException(new FacesMessage("messageText", null));
    }
}
```

login.xhtml

```
<h:inputText id="yesno" value="#{loginBean.yesNo}" validator="#{loginBean.yesNoValidate}" />
```

5

VALIDATOR CLASS

Validator Class

```
@FacesValidator("ch.hftm.validators.LoginValidator")
public class LoginValidator implements Validator {

    @Override
    public void validate(FacesContext context, UIComponent component, Object value) throws ValidatorException {
        int validatorClassNr = (Integer) value;
        if (validatorClassNr < 50 || validatorClassNr > 100) {
            throw new ValidatorException(new FacesMessage("eingabe muss zwischen 50 und 100 liegen", null));
        }
    }

}
```

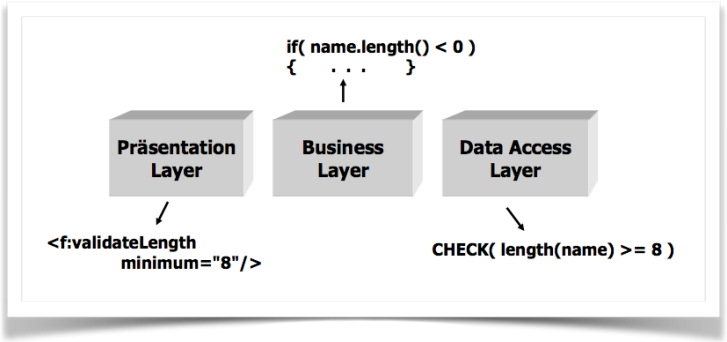
use of the Validator

```
<h:inputText id="validatorClassNr" value="#{loginBean.validatorClassNr}">
    <f:validator validatorId="ch.hftm.validators.LoginValidator" />
</h:inputText>
```

6

BEAN-VALIDATION (JSR-303)

- Validierung der Daten ist eine gemeinsame Aufgabe, die auf verschiedenen Ebenen in der Anwendung auftritt, beginnend bei der Präsentations-Schicht bis auf die Persistenz-Schicht.
- Häufig wird die gleiche Validierungslogik in jeder Schicht implementiert.
- Um duplizierter Code zu vermeiden kann die Validierungslogik direkt in das Domain-Modell geschrieben werden.



7

STANDARD-CONSTRAINTS

- `@AssertFalse`
  Das annotierte Element muss `false` sein.
- `@AssertTrue`
  Das annotierte Element muss `true` sein.
- `@DecimalMax`
  Das annotierte Element muss kleiner oder gleich dem in `value` angegebenen Wert sein. Beachten Sie, dass `value` ein String ist, der als `BigDecimal` interpretiert wird.
- `@DecimalMin`
  Das annotierte Element muss größer oder gleich dem in `value` angegebenen Wert sein. Beachten Sie, dass `value` ein String ist, der als `BigDecimal` interpretiert wird.
- `@Digits`
  Das annotierte Element darf nicht mehr als in `integer` angegebene Stellen vor dem Komma und nicht mehr als in `fraction` angegebene Stellen nach dem Komma haben.
- `@Future`
  Das annotierte Element muss ein Datum in der Zukunft sein.
- `@Past`
  Das annotierte Element muss ein Datum in der Vergangenheit sein.

8

STANDARD-CONSTRAINTS

- `@Max`
  Das annotierte Element muss kleiner oder gleich dem in `value` angegebenen Wert sein. Im Gegensatz zu `@DecimalMax` hat `value` hier den Typ `long` .
- `@Min`
  Das annotierte Element muss größer oder gleich dem in `value` angegebenen Wert sein. Im Gegensatz zu `@DecimalMin` hat `value` hier den Typ `long` .
- `@NotNull`
  Das annotierte Element darf nicht `null` sein.*
- `@Null`
  Das annotierte Element muss `null` sein.*
- `@Pattern`
  Das annotierte Element muss dem in `regexp` angegebenen regulären Ausdruck entsprechen.
- `@Size`
  Die Größe des annotierten Elements muss zwischen `min` und `max` liegen. Mit diesem Constraint können Strings und alle Collection-Typen validiert werden.

NULL-CONSTRAINTS

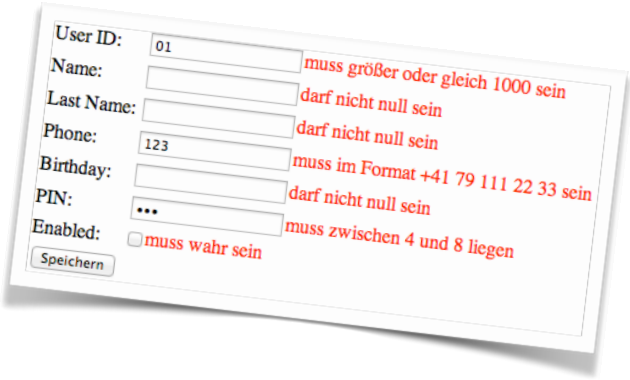
Wichtige Voraussetzung für den Einsatz von @NotNull bzw. @Null

Damit fehlende Benutzereingaben von JavaServer Faces auch wirklich als null -Werte interpretiert werden, muss in der web.xml der Kontextparameter javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL auf true gesetzt werden.

```
<context-param>
   <param-name>javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL</param-name>
   <param-value>TRUE</param-value>
</context-param>
```

EXAMPLE08

```
public class User {

    @NotNull @Min(value = 1000) @Max(value = 9999)
    private int userID;

    @NotNull @Pattern (regexp = "[A-Z][a-z]+", message = "erster Buchstabe......" )
    private String name;

    @NotNull
    private String lastname;

    @Pattern (regexp = "\\+[0-9]{2}\\s[0-9]{2}\\s[0-9]{3}\\s[0-9]{2}\\s[0-9]{2}", message = "muss im......")
    private String phone;

    @NotNull @Past
    private Date birthday;

    @Size(min = 4, max = 8)
    private String pin;

    @AssertTrue
    private boolean enabled;
```



11

VALIDATION MESSAGES

Wine.java (Model)

```
@NotNull (message="{ch.myWinery.REQUIRED}")
private String name;

@Past (message= "{ch.myWinery.PAST}")
private Date year;

@Min(value=1, message= "{ch.myWinery.MIN}")
private BigDecimal price;
```

ValidationMessages_de.properties

```
ch.myWinery.REQUIRED=Please enter a value
ch.myWinery.PAST=Year has to be in the past!
ch.myWinery.MIN=Minimal value: {value}
```

12

JAVA SERVER FACES

Chapter 09 - (Custom)Converter

CONVERTER

- Die Konvertierung ist ein wichtiger Aspekt im JSF-Lebenszyklus, da Datentypen...
  - Client-Seitig Werte die Form von *Zeichenketten* haben
  - Server-Seitig sind sie *Java-Typen*
- Den Konvertern fällt dabei die Rolle eines internen Vermittlers zu. Sie kümmern sich um die Umwandlung der vom Benutzer eingegebenen Zeichenketten in Java-Objekte und wandeln Java-Objekte beim Rendern der Ausgabe wieder in Zeichenketten um
- In JSF gibt es für sehr viele Datentypen bereits Standardkonverter, die automatisch zum Einsatz kommen
  - Integer, int
  - Short, short
  - Double, double
  - Float, float
  - Long, long
  - BigDecimal
  - BigInteger
  - Boolean, boolean
  - Byte, byte
  - Character, char

f:convertDateTime

- Der Konverter *f:convertDateTime* kann verwendet werden, um ein Date-Objekt in einen String zu konvertieren und umgekehrt.
- Das Tag *f:convertDateTime* hat folgende Eigenschaften:
  - **type**
    Dieses Attribut definiert, welche Teile des Datumswertes angezeigt werden. Kann die Werte date , `time` oder `both` annehmen.
  - **dateStyle**
    Dieses Attribut gibt den Typ der Datumsanzeige an, wenn type auf date oder both gesetzt ist. Mögliche Werte sind `default` , `short` , `medium` , `long` , `full` .
  - **timeStyle**
    Dieses Attribut gibt den Typ der Zeitanzeige an, wenn type auf `time` oder `both` gesetzt ist. Es sind die gleichen Werte wie für die dateStyle -Angabe gültig.
  - **pattern**
    Statt der Angabe von `type` , `dateStyle` und `timeStyle` kann hier direkt ein Datumsmuster (wie etwa dd.MM.yyyy) übergeben werden.
  - **timeZone**
    Die Zeitzone für die Ausgabe kann mit diesem Attribut umgestellt werden, beispielsweise ist der Standardwert für dieses Attribut `GMT`.
  - **locale**
    Hier kann eine Lokalisierung angegeben werden - sowohl als Zeichenkette als auch als Instanz der Klasse Locale über eine Value-Expression . Beispiele für den Wert sind: `en_US` oder `#{session.locale}`.

3

f:convertNumber

- *f:convertNumber* ist nützlich für das Anzeigen von Zahlen in Basisformate wie eine Währung oder einen Prozentsatz.
- Das Tag *f:convertNumber* hat folgende Eigenschaften:
  - **type**
    Kann die Eigenschaften `currency` oder `percentage` zusätzlich zum Standardwert `number` annehmen.
  - **currencyCode**
    Mit dieser Eigenschaft kann die Währung über einen international gültigen Code eingestellt werden - für den Euro wäre das beispielsweise `EUR` . Alternativ dazu können Sie auch das Attribut `currencySymbol` setzen.
  - **currencySymbol**
    Alternative zum `currencyCode` . Beispiel wäre die Angabe von für den Euro.
  - **groupingUsed**
    Gibt an, ob ein Separator (beispielsweise ein Tausenderseparator) verwendet wird.
  - **locale**
    Hier kann eine Lokalisierung angegeben werden - sowohl als Zeichenkette als auch als Instanz der Klasse Locale über eine Value-Expression. Beispiele für den Wert sind: `en_US` oder `#{session.locale}`.
  - **minFractionDigits / maxFractionDigits**
    Dieses Attribut gibt die minimale / maximale Anzahl an Nachkommastellen an.
  - **minIntegerDigits / maxIntegerDigits**
    Dieses Attribut gibt die minimale / maximale Anzahl an Vorkommastellen an.
  - **pattern**
    Alternativ zur Angabe der anderen Attribute kann mit diesem Attribut direkt ein Muster für die Darstellung festgelegt werden - beispielsweise ist hier die Angabe von `#.###,##` für das "deutsche" Zahlenformat möglich.

4

CUSTOM CONVERTER

- Natürlich kann die Funktionalität des JSF-Frameworks durch Erstellen eigener Konverter erweitert werden
- Es ist möglich...
  - bestehende Konverter durch eigene zu ersetzen
  - neue Java-Datentypen zu verarbeiten
  - für einzelne Komponenten-Tags eigene Konverter zu entwickeln

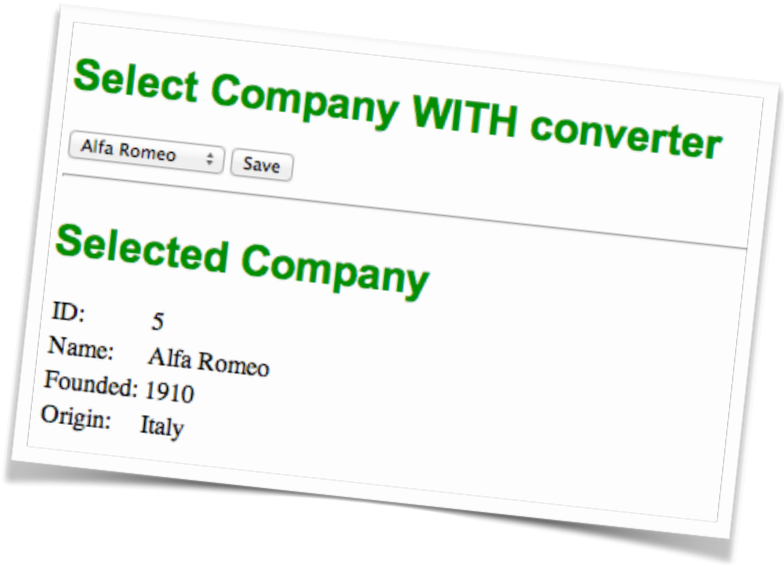
5

HOW TO...

1. Erstellen eine Konverter Klasse durch die Implementierung des `javax.faces.convert.Converter` Interface
2. Überschreibe die Methoden `getAsObject()` und `getAsString()`
3. Annotiere die Klasse mit `@FacesConverter(value= "ID")` und weise ihr eine eindeutige ID zu
4. Verknüpfe den Custom Converter mittels `f:converter` in der JSF Komponente

6

EXAMPLE09



MODEL

Company.java

```
public class Company {

    private long id;
    private String name;
    private int founded;
    private String origin;

    public Company(long id, String name, int founded, String origin) {
        this.id = id;
        this.name = name;
        this.founded = founded;
        this.origin = origin;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

MANAGED BEAN

CompanySelectorBean.java

```
@ManagedBean
@SessionScoped
public class CompanySelectorBean {

    private CompanyCatalog companyCatalog = CompanyCatalog.getInstance();
    private Company selectedCompany;

    public List<Company> getCompanies() {
        return companyCatalog.getAllCompanies();
    }

    public Company getSelectedCompany() {
        return selectedCompany;
    }

    public void setSelectedCompany(Company selectedCompany) {
        this.selectedCompany = selectedCompany;
    }

}
```


CUSTOM CONVERTER

CompanyConverter.java

```
@FacesConverter(value = "ch.hftm.converter.CompanyConverter")
public class CompanyConverter implements Converter {

    private CompanyCatalog companyCatalog = CompanyCatalog.getInstance();

    @Override
    public Object getAsObject(FacesContext context, UIComponent component, String submittedValue) {
        long companyId;

        try {
            companyId = Long.parseLong(submittedValue);
        } catch (NumberFormatException exception) {
            throw new ConverterException();
        }
        return companyCatalog.findCompany(companyId);
    }

    @Override
    public String getAsString(FacesContext context, UIComponent component, Object modelValue) {
        String s = "";

        if (modelValue instanceof Company) {
            Company company = (Company) modelValue;
            s += company.getId();
            return s;
        }
        return s;
    }
}
```

JSF COMPONENT

CompanySelector.xhtml

```
<h:form>
    <h:outputText value="Select Company WITH converter" styleClass="header" />
    <br />
    <h:selectOneMenu id="companySelector2" value="#{companySelectorBean.selectedCompany}">
        <f:selectItem noSelectionOption="true" itemLabel="please select" />
        <f:selectItems value="#{companySelectorBean.companies}" var="company"
            itemLabel="#{company.name}" itemValue="#{company}" />
        <f:converter converterId="ch.hftm.converter.CompanyConverter" />
    </h:selectOneMenu>
    <h:commandButton value="Save" />
    <h:message for="companySelector2" errorClass="errorMessage" />
</h:form>
```

11

JAVA SERVER FACES

Chapter 10 - Composite Component

1

COMPOSITE COMPONENT

Durch Kompositkomponenten erhalten Entwickler durch die Verbindung von Facelets und Ressourcen die Möglichkeit, Komponenten aus beinahe beliebigen Seitenfragmenten aufzubauen - daher auch der Name. Eine Kompositkomponente ist im Grunde nichts anderes als ein XHTML-Dokument, das in einer Ressourcenbibliothek abgelegt ist und die Komponente deklariert.

2

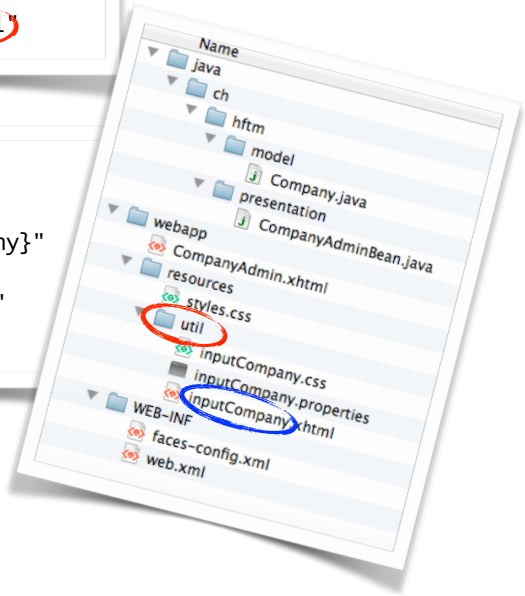
DIRECTORY STRUCTURE

Namespace:

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

Tag Name:

```
<util:inputCompany value="#{companyAdminBean.company}"
                   submitAction="#{companyAdminBean.save}"
                   cancelAction="#{companyAdminBean.cancel}"
                   originRequired="true" />
```



CC STRUCTURE

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

    <cc:interface>
        ...
        ...
    </cc:interface>


    <cc:implementation>
        ...
        ...
    </cc:implementation>

</html>
```

Der Bereich cc:interface definiert die Schnittstelle der Komponente nach aussen und umfasst alle Merkmale, die für Benutzer der fertigen Komponente relevant sind. Neben der Definition von Attributen und Facets ist es auch möglich, das Verhalten einzelner interner Komponenten nach aussen bekanntzugeben.

Der Bereich cc:implementation enthält alle JSF-Tags, HTML-Elemente und anderweitigen Inhalte, aus denen die Kompositkomponente aufgebaut ist.
Im Implementierungsteil gibt es mehrere Möglichkeiten, Inhalte einzufügen, die ein Benutzer der Kompositkomponente innerhalb der Komponenten-Tags angegeben hat.

INTERFACE

- `cc:attribute`
  declares an attribute that may be given to an instance of the composite component in which this tag is declared
  `cc:valueHolder`
  exposes a component that holds a value. This allows the JSF page author to register converters to (some part of) the composite component.
  `cc: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.
  `cc:actionSource`
  declares that the belonging composite component exposes an implementation of ActionSource.

5

EXAMPLE10

6

INTERFACE IMPLEMENTATION

```
<cc:interface>
    <cc:attribute name="value" required="true"/>
    <cc:attribute name="nameRequired"  default="true"/>
    <cc:attribute name="foundedRequired" default="false"/>
    <cc:attribute name="originRequired" default="false"/>
    <cc:attribute name="submitAction"
        method-signature="java.lang.String action()"
        required="true"/>
    <cc:attribute name="cancelAction"
        method-signature="java.lang.String action()"
        required="true"/>
    <cc:editableValueHolder name="name" targets="name"/>
    <cc:editableValueHolder name="founded" targets="founded"/>
</cc:interface>

<cc:implementation>
    ...
    ...
</cc:implementation>
```

7

CC IMPLEMENTATION

```
<cc:implementation>
    <h:outputStylesheet library="util" name="inputCompany.css"/>
    <h:panelGrid columns="3">

        <h:outputText value="#{cc.resourceBundleMap.name}"/>
        <h:inputText id="name" value="#{cc.attrs.value.name}"
                     required="#{cc.attrs.nameRequired}"
                     requiredMessage="#{cc.resourceBundleMap.requiredMessage}"/>
        <h:message for="name" errorClass="errorMessage"/>

        <h:outputText value="#{cc.resourceBundleMap.founded}"/>
        <h:inputText id="founded" value="#{cc.attrs.value.founded}"
                     required="#{cc.attrs.foundedRequired}"
                     requiredMessage="#{cc.resourceBundleMap.requiredMessage}"/>
        <h:message for="founded" errorClass="errorMessage"/>

        <h:outputText value="#{cc.resourceBundleMap.origin}"/>
        <h:inputText id="origin" value="#{cc.attrs.value.origin}"
                     required="#{cc.attrs.originRequired}"
                     requiredMessage="#{cc.resourceBundleMap.requiredMessage}"/>
        <h:message for="origin" errorClass="errorMessage"/>

        <h:commandButton action="#{cc.attrs.submitAction}" value="#{cc.resourceBundleMap.submitAction}"/>
        <h:commandButton action="#{cc.attrs.cancelAction}" value="#{cc.resourceBundleMap.cancelAction}"
            immediate="TRUE"/>

    </h:panelGrid>
</cc:implementation>
```

8

OPTIONAL FILES

inputCompany.properties:

```
name=Name
founded=Founded
origin=Origin
submitAction=Save
cancelAction=Cancel
requiredMessage=This value is required
```

inputCompany.css:

```
.requiredtrue { background-color: azure; }
.requiredfalse { background-color: white; }
.errorMessage { color: red; }
```

9

USAGE OF THE COMPONENT

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


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

<h:head> ... </h:head>

    <h:body>
        <h:form>
            <util:inputCompany value="#{companyAdminBean.company}"
                submitAction="#{companyAdminBean.save}"
                cancelAction="#{companyAdminBean.cancel}"
                originRequired="true" />
            <f:validateRegex for="name" pattern="[A-Z][a-z]*"/>
            <f:validateLongRange for="founded" minimum="1000"/>
        </h:form>
    </h:body>

</html>
```

10

JAVA SERVER FACES

Chapter 11 - AJAX

AJAX

Ajax hat in den letzten Jahren enorm an Bedeutung gewonnen und ist aus der Webentwicklung nicht mehr wegzudenken.

Mit der stark zunehmenden Nutzung des Internets in den letzten Jahren stieg auch der Wunsch nach einer interaktiveren und umfangreicheren Benutzung von Webseiten, wie sie im Bereich herkömmlicher Desktop-Applikationen bereits seit vielen Jahren üblich und Benutzern vertraut ist.

Diese Lücke soll durch neue Ansätze in der Webtechnologie geschlossen werden.

Das grundlegende Ziel muss lauten, das Web noch stärker an die oft sehr unterschiedlichen Bedürfnisse der Benutzer anzupassen.

<F:AJAX ...

- **`event:`**
  Name des Ereignisses, das die Ajax-Anfrage auslöst.
  Mögliche Werte sind ***valueChange*** für ***Eingabekomponenten***,
  action für ***Steuerkomponenten*** und alle anderen HTML-Ereignisse - allerdings ohne das Präfix on. Der Defaultwert dieses Attributs wird von der Komponente bestimmt.
- **`execute:`**
  Eine durch Leerzeichen separierte Liste der IDs jener Komponenten, die beim Bearbeiten der Ajax-Anfrage durch JSF im Lebenszyklus ausgeführt werden sollen. Kann auch die Konstanten ***@this*** (das Element selbst), ***@form*** (das Formular des Elements), ***@all*** (alle Elemente) und ***@none*** (kein Element) beinhalten. Der Defaultwert ist @this.
- **`render:`**
  Eine durch Leerzeichen separierte Liste der IDs jener Komponenten, die beim Bearbeiten der Ajax-Anfrage durch JSF im Lebenszyklus gerendert werden sollen. Kann auch die Konstanten @***this*** , @***form*** , @***all*** und @***none*** beinhalten. Der Defaultwert ist @none.
- **`listener:`**
  In diesem Attribut kann eine ***Methode*** einer ***Managed-Bean*** über eine Method-Expression als Listener registriert werden. Die Methode wird während der Ajax-Anfrage in der Invoke-Application-Phase ausgeführt.

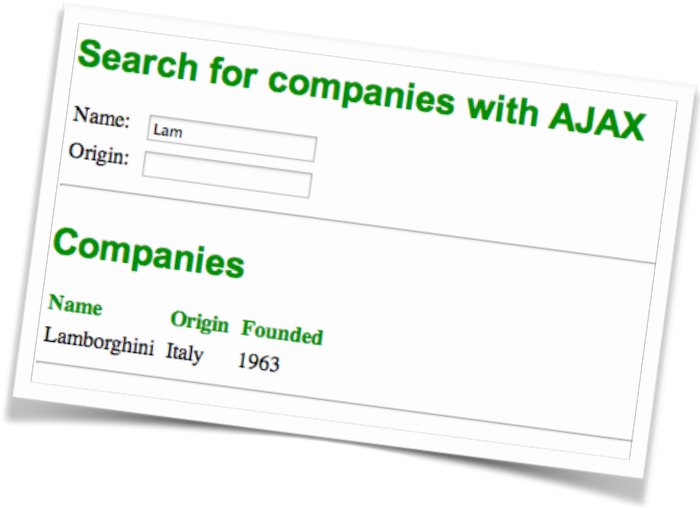
3

<F:AJAX ...

- **`onerror:`**
  Erlaubt das Registrieren einer clientseitigen JavaScript-Callback-Funktion für Fehler, die beim Bearbeiten der Ajax-Anfrage auftreten.
- **`disabled:`**
  Das Ajax-Verhalten wird "abgeschaltet", wenn dieses Attribut auf true gesetzt ist.

4

EXAMPLE 11



5

WITHOUT AXAJ

```
<h:form>
    <h:outputText value="Search for companies" styleClass="header" />
    <br />
    <h:panelGrid columns="2">
        <h:outputText value="Name: " />
        <h:inputText value="#{companyCatalogBean.name}" />
        <h:outputText value="Origin: " />
        <h:inputText value="#{companyCatalogBean.origin}" />
    </h:panelGrid>
    <h:commandButton value="Search" action="#{companyCatalogBean.searchCompanies}" />
</h:form>
```

6

WITH AJAX

```
<h:form>
    <h:outputText value="Search for companies with AJAX" styleClass="header" />
    <br />
    <h:panelGrid columns="2">
        <h:outputText value="Name: " />
        <h:inputText value="#{companyCatalogBean.name}">
            <f:ajax event="keyup" render="searchResults messages" execute="@form"
            listener="#{companyCatalogBean.searchCompaniesByTiping}" />
        </h:inputText>
        <h:outputText value="Origin: " />
        <h:inputText value="#{companyCatalogBean.origin}">
            <f:ajax event="keyup" render="searchResults messages" execute="@form"
            listener="#{companyCatalogBean.searchCompaniesByTiping}" />
        </h:inputText>
    </h:panelGrid>
</h:form>
```

7

RESULT TABLE

```
<h:outputText value="Companies" styleClass="header" />
<h:panelGroup id="searchResults">
    <h:dataTable value="#{companyCatalogBean.foundCompanies}" var="company"
    rendered="#{not empty companyCatalogBean.foundCompanies}">
        <h:column>
            <f:facet name="header">
                <h:outputText value="Name" />
            </f:facet>
            <h:outputText value="#{company.name}" />
        </h:column>
        <h:column>
            <f:facet name="header">
                <h:outputText value="Origin" />
            </f:facet>
            <h:outputText value="#{company.origin}" />
        </h:column>
        <h:column>
            <f:facet name="header">
                <h:outputText value="Founded" />
            </f:facet>
            <h:outputText value="#{company.founded}" />
        </h:column>
    </h:dataTable>
</h:panelGroup>
```

8

MANAGED BEAN

```
public String searchCompanies() {
    foundCompanies.clear();
    message = null;
    try {
        foundCompanies = companyCatalog.searchCompany(name, origin);
        if (foundCompanies.isEmpty()) {
            message = "No matching company found";
        } else {
            return null;
        }
    } catch (Exception e) {
        message = "Search criteria are missing";
    }
    return null;
}

public void searchCompaniesByTiping(AjaxBehaviorEvent ev) {
    foundCompanies.clear();
    message = null;
    if (name.length() > MIN_SEARCH_LENGTH || origin.length() > MIN_SEARCH_LENGTH) {
        searchCompanies();
    }
}
```

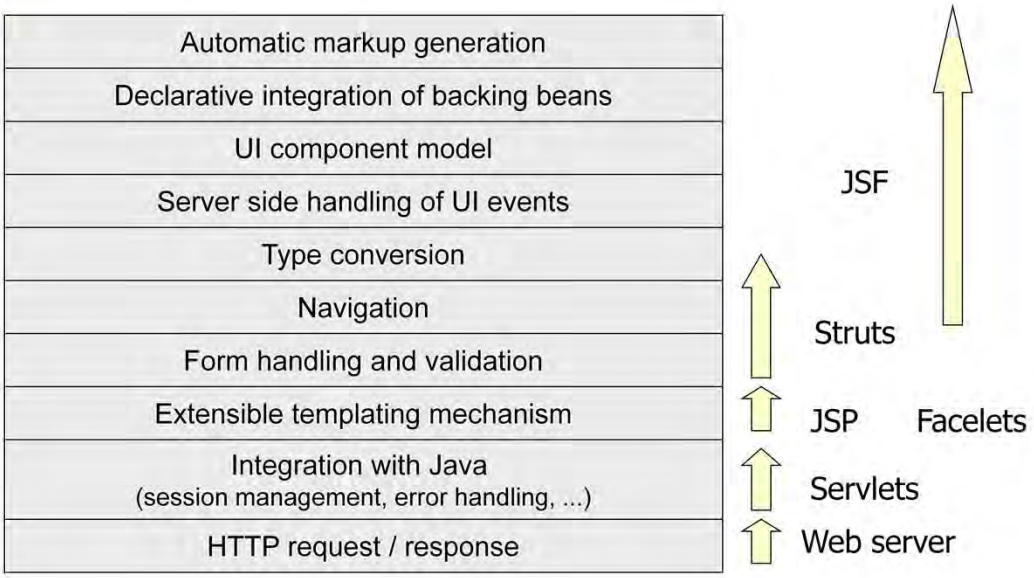
JavaServer Faces

Beatrice Amrhein

Contents

1 Introduction

Web Frameworks



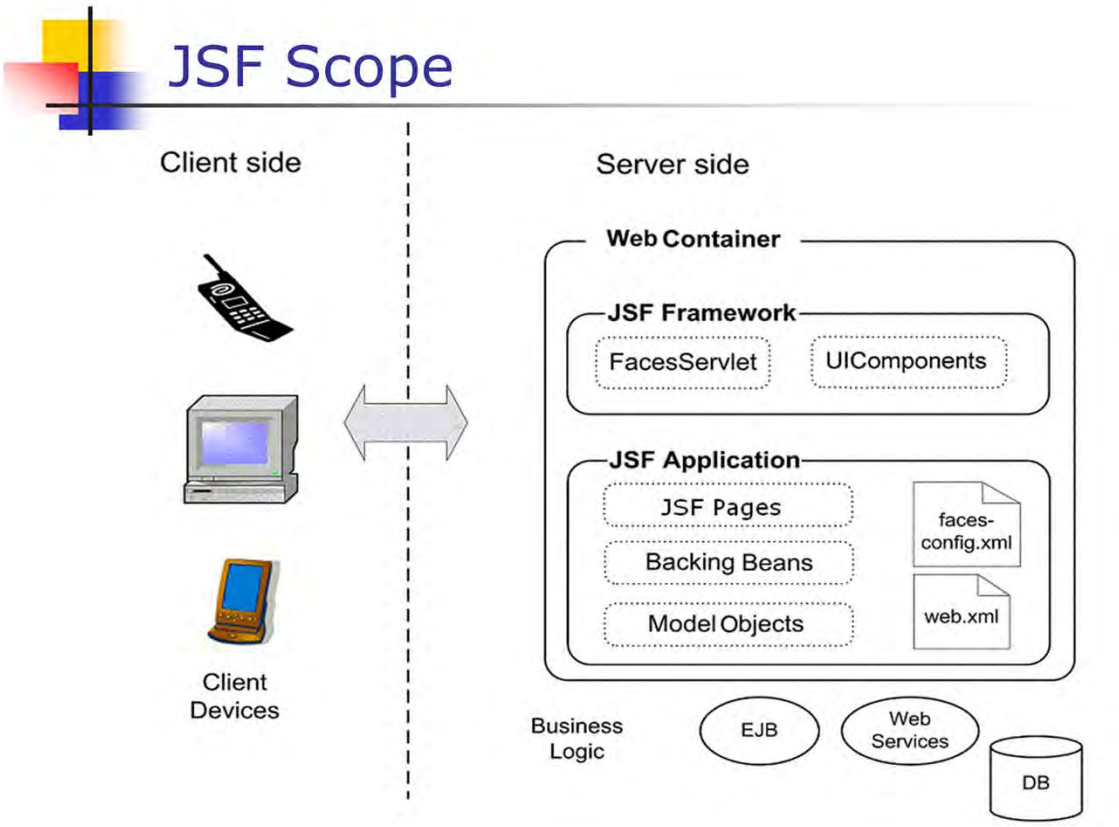
The biggest advantage of the JSF technology is its flexible, extensible component model, which includes:

• 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, RichFaces:www.jboss.org/richfaces, PrimeFaces: primefaces.org or ICEFaces: 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.



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>
```

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.

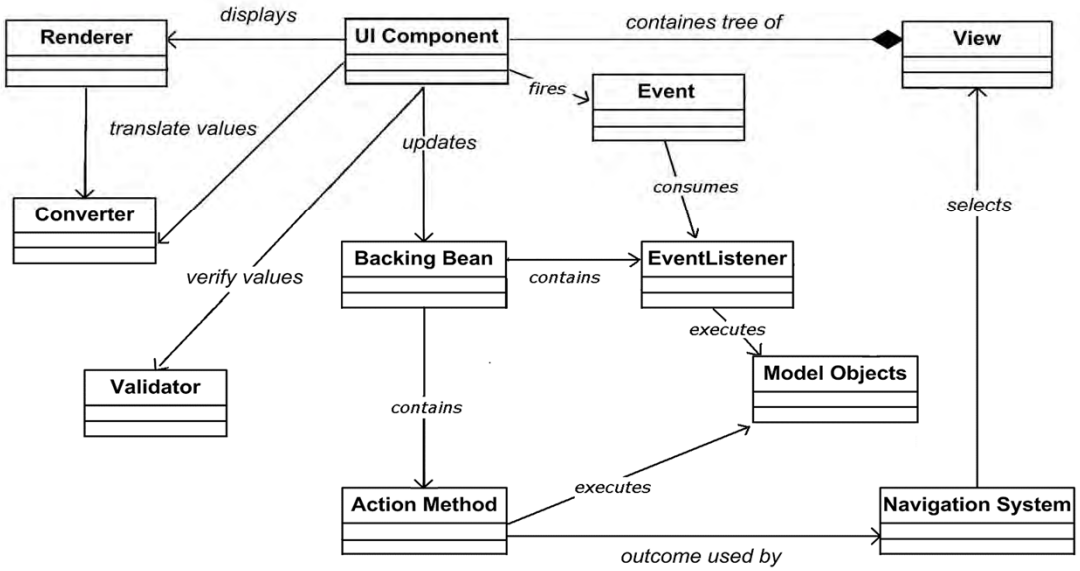
Translation to HTML


```
<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.

Confer chapter 2.2 of

http://jsfatwork.irian.at

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.

Confer chapter 2.5 of

http://jsfatwork.irian.at

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

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

home.xhtml (1)

```
<?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)


```
<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)


```
    <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.

web.xml

```
<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>
```

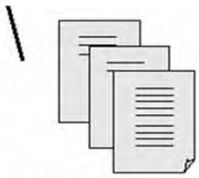
The welcome file list is optional and defines the possible entry points of your JSF application.

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.

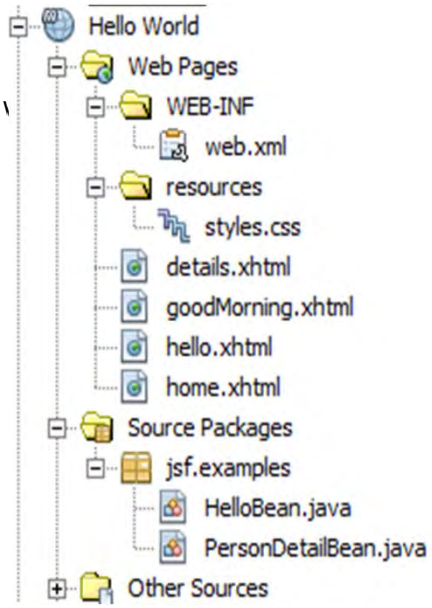


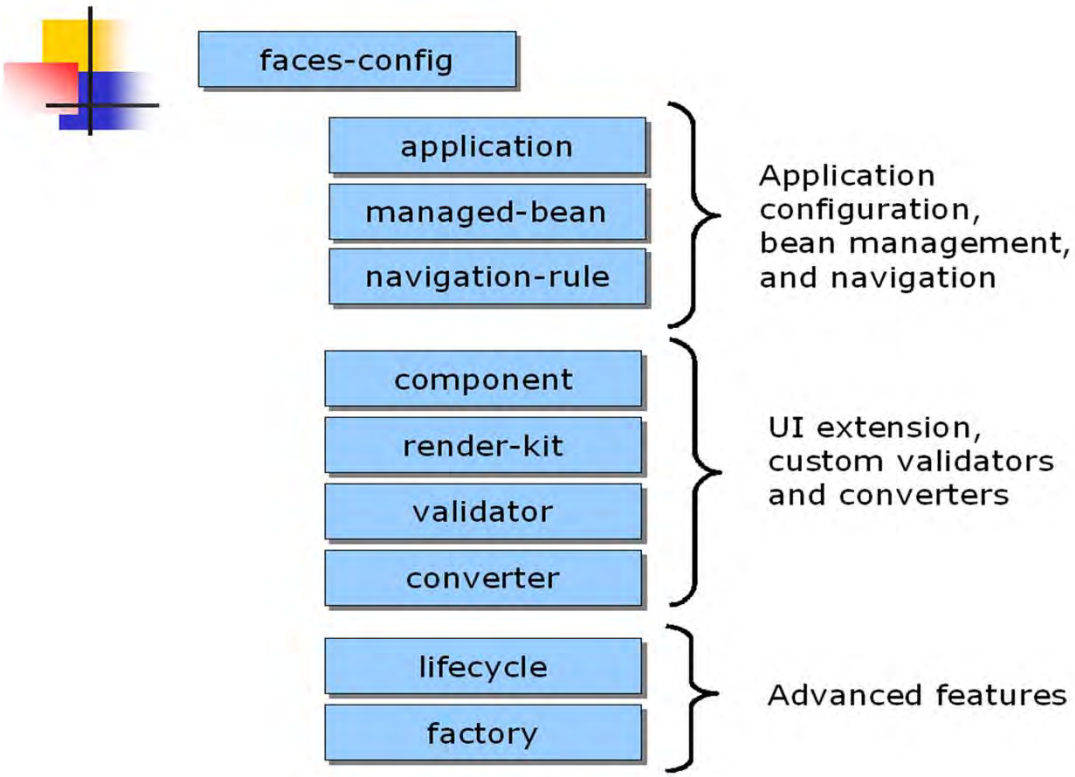
JSF Pages

\
\WEB-INF
web.xml
faces-config.xml
\lib
application jar files
\classes
application class files
resource files (java classpath)
\resources
images, css-files

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.

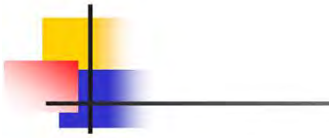
All top level elements (application, managed-bean, ..., factory) are optional and can be included once or more than once.

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


```
<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


```
<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!

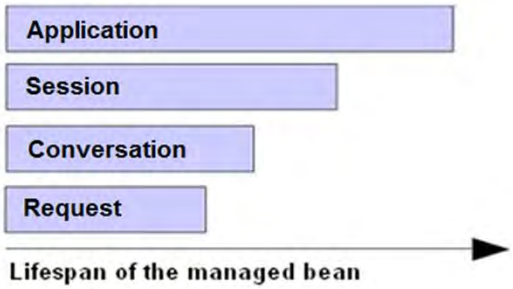
Home

```
<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 been 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 construtor 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 ad 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.

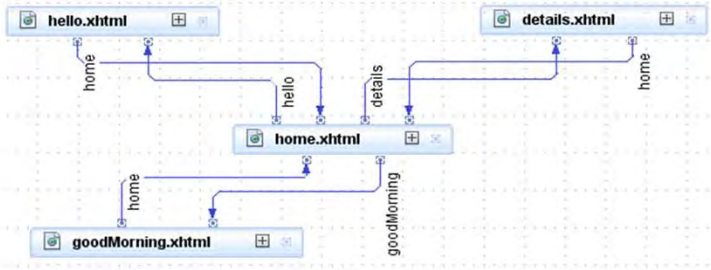
XML configuration

```
<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)



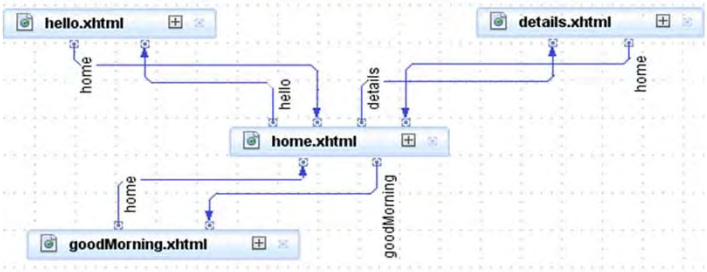

```
<navigation-rule>
   <from-view-id>/home.xhtml</from-view-id>
   <navigation-case>
         <from-action>#{helloBean.sayHello}
         </from-action>
         <from-outcome>hello</from-outcome>
         <to-view-id>/hello.xhtml</to-view-id>
   </navigation-case>
```

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

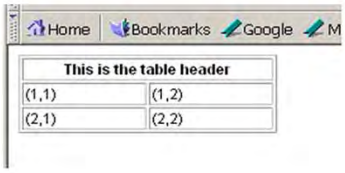
Shared Attributes

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

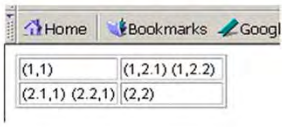
```
<h:panelGrid columns="2" border="1"
        headerClass="page-header" cellpadding="1" width="40%">
    <f:facet name="header">
        <h:outputText value="This is the table header"/>
    </f:facet>
    <h:outputText value="(1,1)"/>
    <h:outputText value="(1,2)"/>
    <h:outputText value="(2,1)"/>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



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>
```



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"/>
```

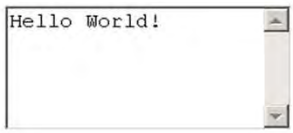
Your Name:

Your Age: 17

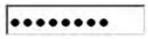
The HtmlInputText component is used for basic user input. It maps to a simple text field—the HTML <input> element with type “text”.
The size and disabled properties are passed through to the HTML input tag.
In this example, the component is associated via a value-binding expression with the „name“ property of helloBean.
HtmlInputText is the most common input control. It displays a single input text field.

Further Input Components

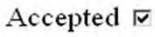
```
<h:inputTextarea value="#{helloBean.helloText}"  rows="5" />
```




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



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



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"/>
<h:outputLink/>

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

to the jsf page

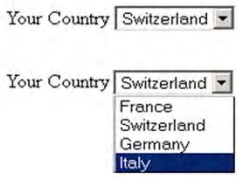
The command components represent an action initiated by the user. The two standard command components are CommandButton (for displaying a button) and CommandLink (for displaying a hyperlink).
As they perform the same operation, their usage is quite similar. They both trigger an action via a POST request 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>
```

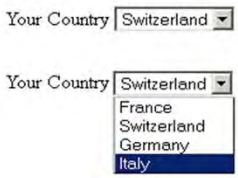


User interfaces often allow a user to select one or more items from a set of possible choices.
In JSF, a selection item represents a single choice, and has a value and a label. Components in the SelectMany and SelectOne families, like HtmlSelectManyCheckbox and HtmlSelectOneListbox, display lists of 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>
```



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

Name	Profession
Peter	Pilot
Alice	Cook
George	Philosopher
Julia	Actress

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

Name	Profession
Peter	Pilot
Alice	Cook
George	Philosopher
Julia	Actress

```
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;
    }
      ...
}
```

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
Peter	Pilot
Alice	Cook
George	Philosopher
Julia	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?

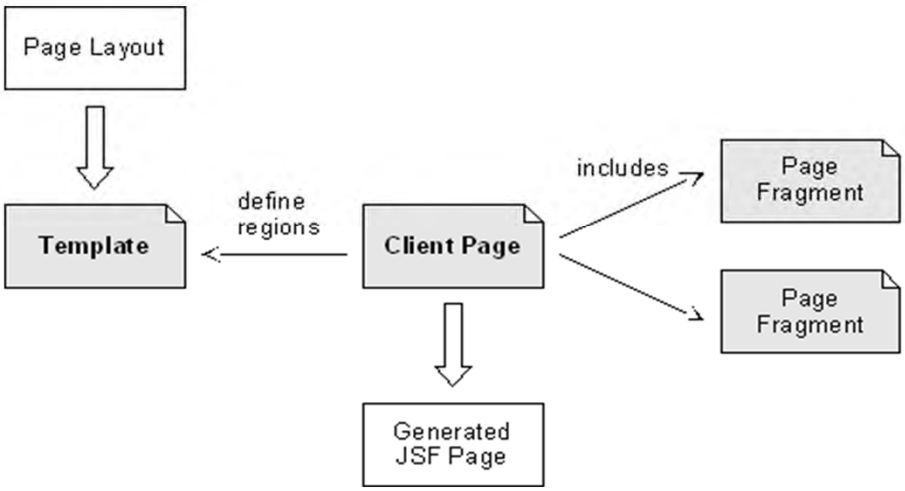
- Avoiding repetition in JSF pages
- Support for standard look and feel

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.

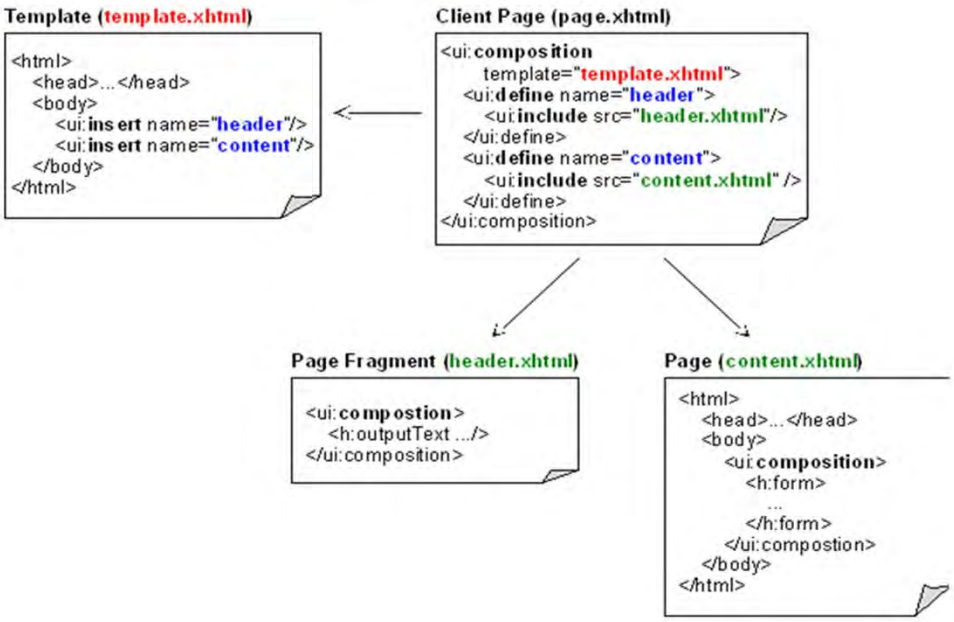
Basic Steps

- 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



Templating Example

Hello World

Your Name John

Details Say Hello

Person Details

Your Age 22

Your Country Switzerland

Name	Profession
Peter	Pilot
Alice	Cook
George	Philosopher
Julia	Actress

Your Best Friend: George

Home

Good morning John from Switzerland!

Home

The Template

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>

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

inline definition

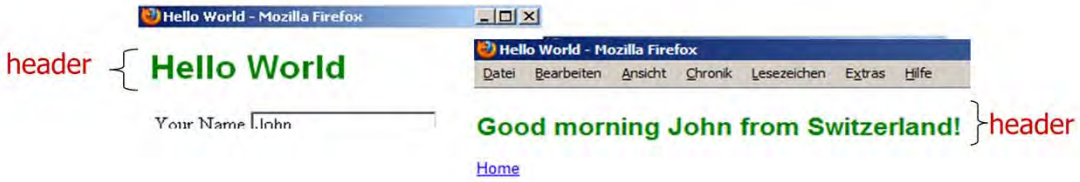
header { Hello John from Switzerland!

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>
```

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.

home_content.xthml (page)

```
<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."

Confer chapter 2.14 of

http://jsfatwork.irian.at

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>
```

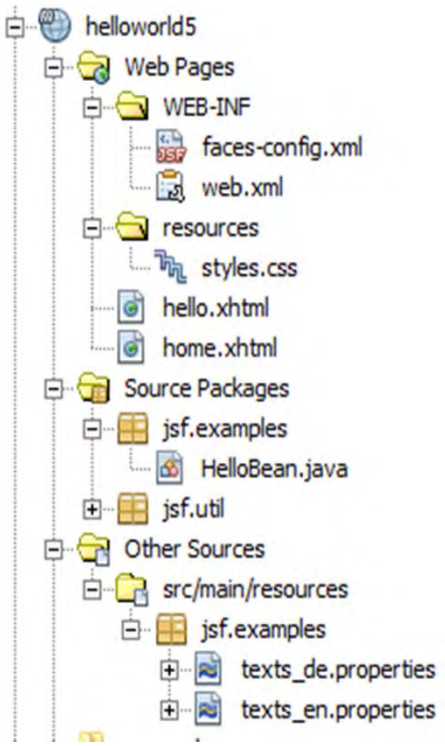
You have to tell your JSF application which language it should support. The supported locales are specified with the <locale-config> element in the Faces configuration file, inside the <application> element.
The element <default-locale> specifies the default locale which is used if the current locale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

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=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>
```

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 UIcomponent 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 ist 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 directroy 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.

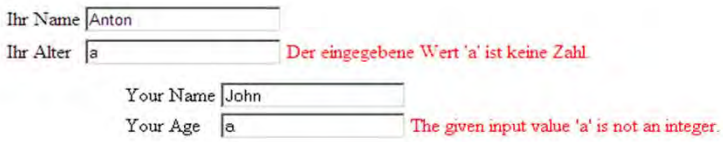
Using Localized Messages

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= . . .
```



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 summery text, but with an _detail extension.

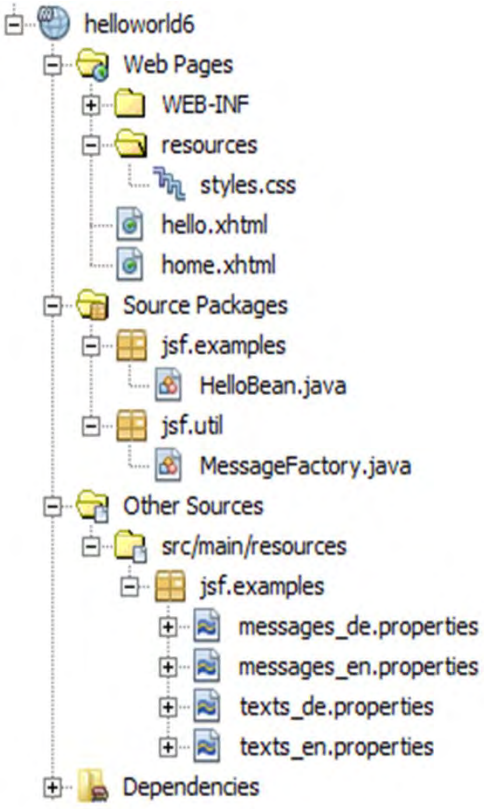
Directory Structure

HTML, CSS, ...

\WEB-INF
web.xml
faces-config.xml
\lib
application jar files
\classes
application class files
resource files
(e.g. messages_en.properties)

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

In out 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

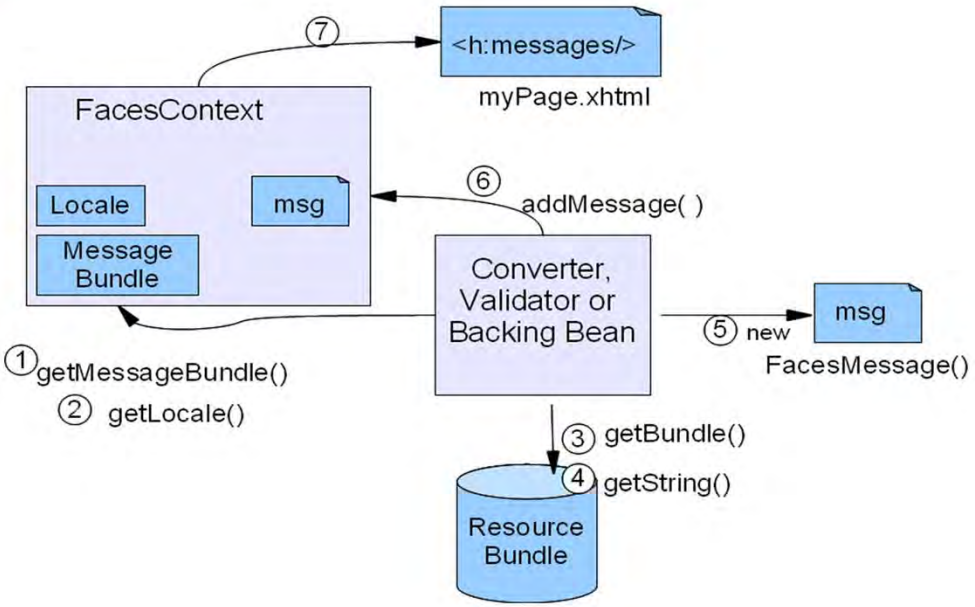
```
public String sayHello() {
   if (age < 1 || age > 120) {

      // 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 optain 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 mesage (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

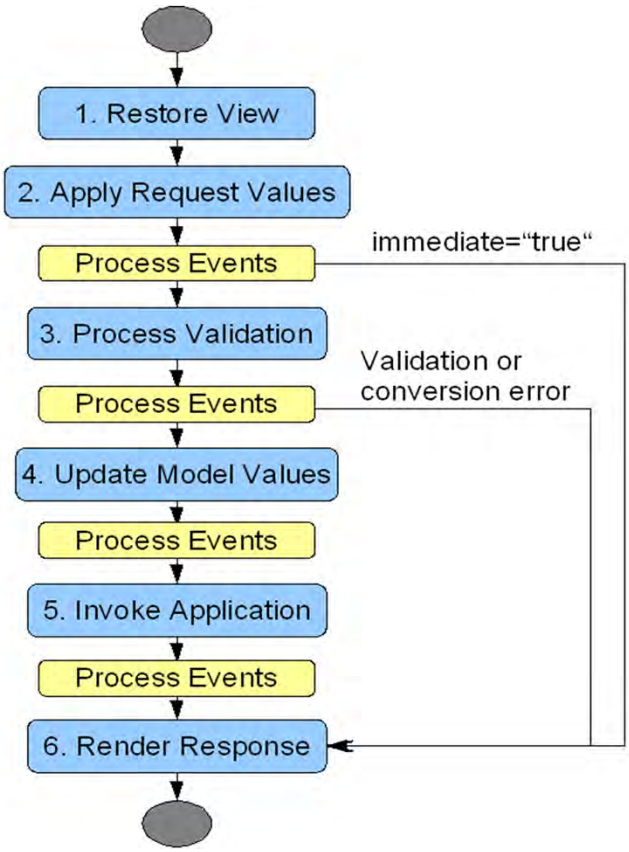
```
// display only global messages
<h:messages globalOnly="true">
```

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

Confer chapter 2.6 of

http://jsfatwork.irian.at



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 [0] 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 19.04.2006

```
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

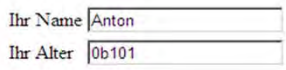
```
@Override
public Object getAsObject(FacesContext ctxt, UIComponent c,
                          String value) throws ConverterException {
  int radix = 10;
  if (value.startsWith("0x")) {
    radix = 16;  value = value.substring(2);
  }
  if (value.startsWith("0")) {
    radix = 8;  value = value.substring(1);
  }
  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" />
```



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
- myfaces.apache.org/tomahawk
- icesoft.org
- 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), for example, to find out about different JSF products and libraries.

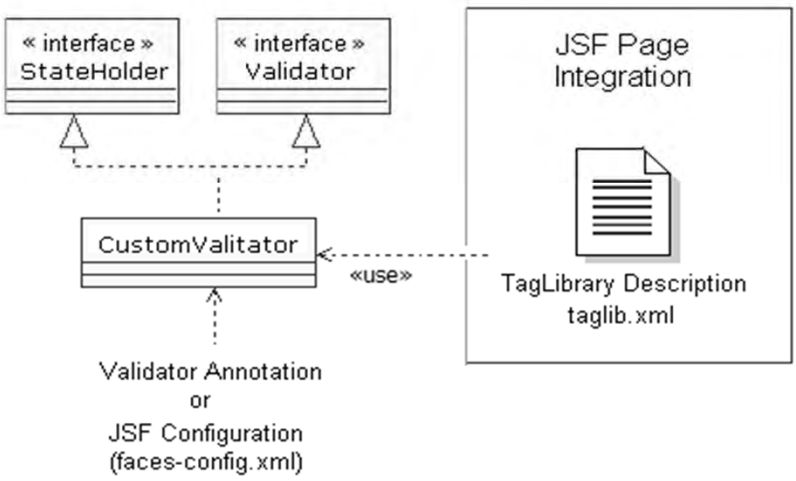
Custom Validator or Converter Tag

- 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.

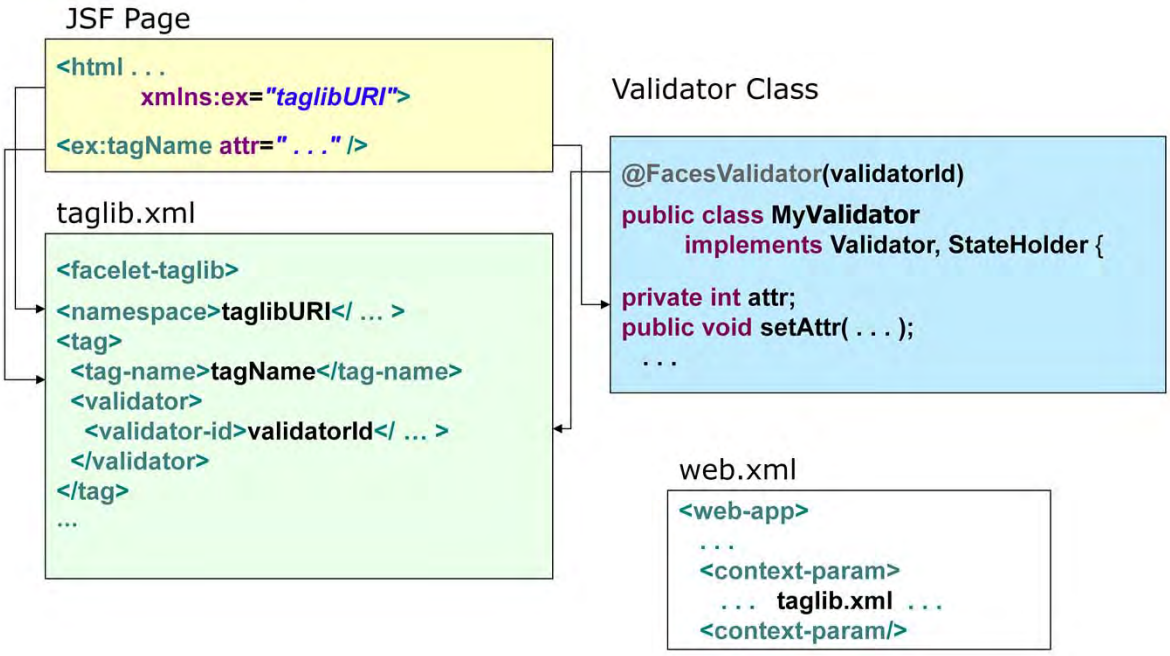
The method isTransient() returns true, if all properties are transient (nothing has to be saved). Therefore, in this example the default value for transientValue is false.

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.

Confer chapter 5.1 of

http://jsfatwork.irian.at

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, developer 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 auther can assign an actionMethod, which is invoked as soon as the user clicks on the icon.

Composite Implementation

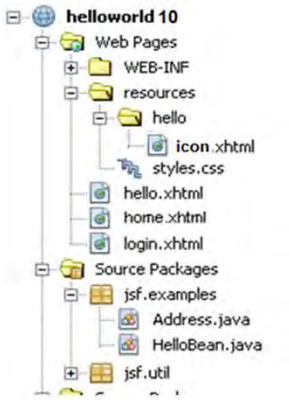
```
<composite:implementation>
    <h:commandLink action="#{cc.attrs.actionMethod}"
                   immediate="#{not cc.attrs.doValidation}">
        <h:graphicImage url="#{cc.attrs.image}" />
    </h:commandLink>
</composite:implementation>

</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**.

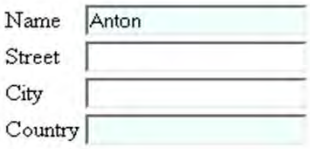
Usage of the Component

```
<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



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

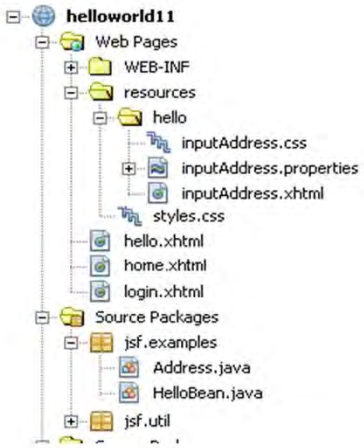
```
<h:panelGrid columns="3">
   <h:outputLabel for="name" value="#{cc.resourceBundleMap.name}"/>
   <h:inputText id="name" value="#{cc.attrs.value.name}"
         required="#{cc.attrs.nameRequired}"
         requiredMessage="#{cc.resourceBundleMap.requiredMessage}"
         styleClass="required#{cc.attrs.nameRequired}"/>
   <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.properies file in the components directory.

Namespace and Tag Name

The helloWorld directory structure



The name of the tag:

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

The namespace:

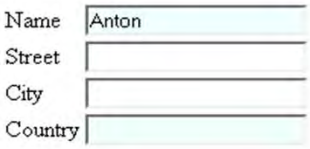
```
xmlns:hello = "http://xmlns.jcp.org/jsf/composite/hello"
```

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>
```



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

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.

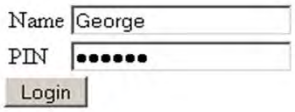
The Implementation: Functionality

```
<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"/>
```

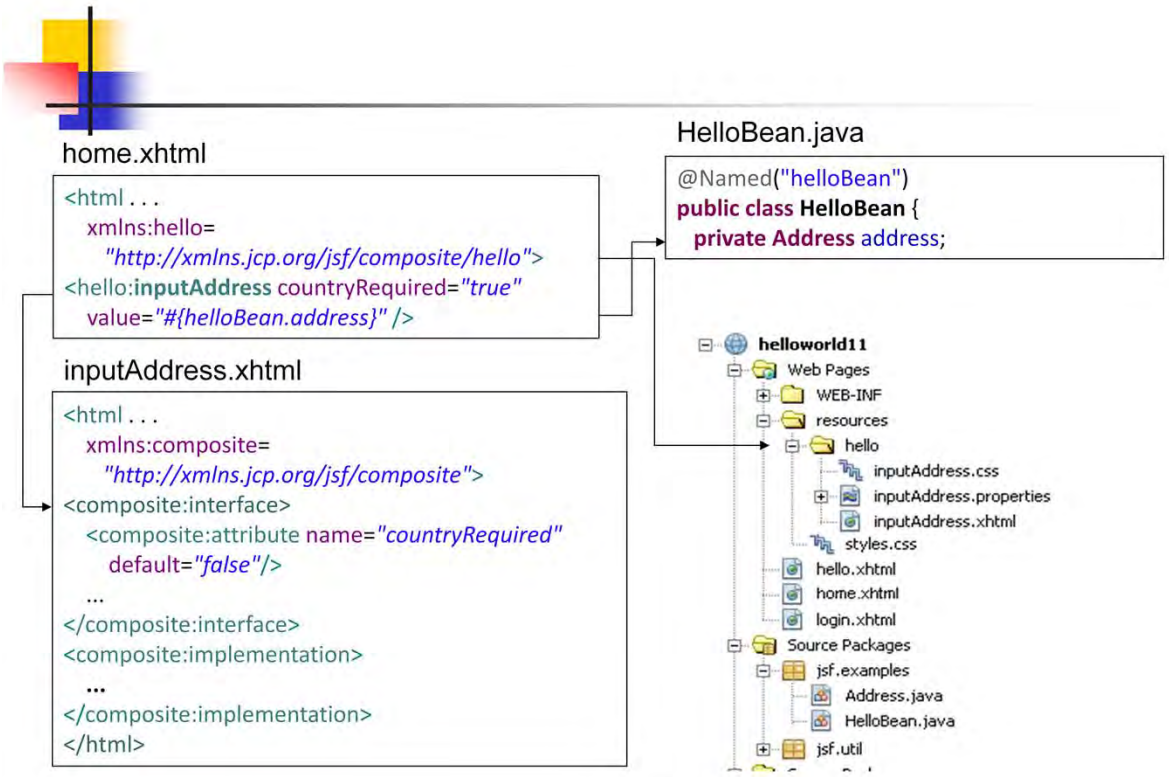
The inputText and inputSecret component each have an id, which is the target of the corresponding editableValueHolder.
This id can be used by the page author to register a validator or a special converter.

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>
```



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

Confer chapter 6 of

http://jsfatwork.irian.at

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

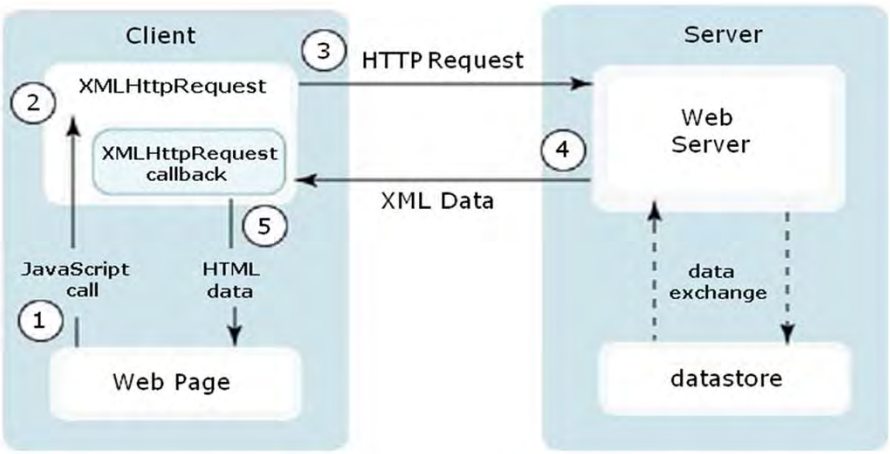
AJAX uses

- 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.

f:ajax

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.

f:event

Install an event listener

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

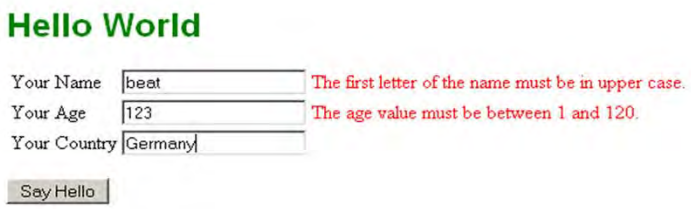
Example

```
<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.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 beat 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.

hello.xhtml

Immediate validation and error message processing

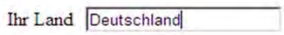
```
<h:inputText id="age" label="#{texts.yourAge}"
    value="#{helloBean.age}" required="true"
    validatorMessage="#{texts.invalidAge}">
        <f:validateLongRange minimum="1" maximum="120" />
        <f:ajax event="keyup" render="ageMessage" />
</h:inputText>
<h:message id="ageMessage" for="age"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

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

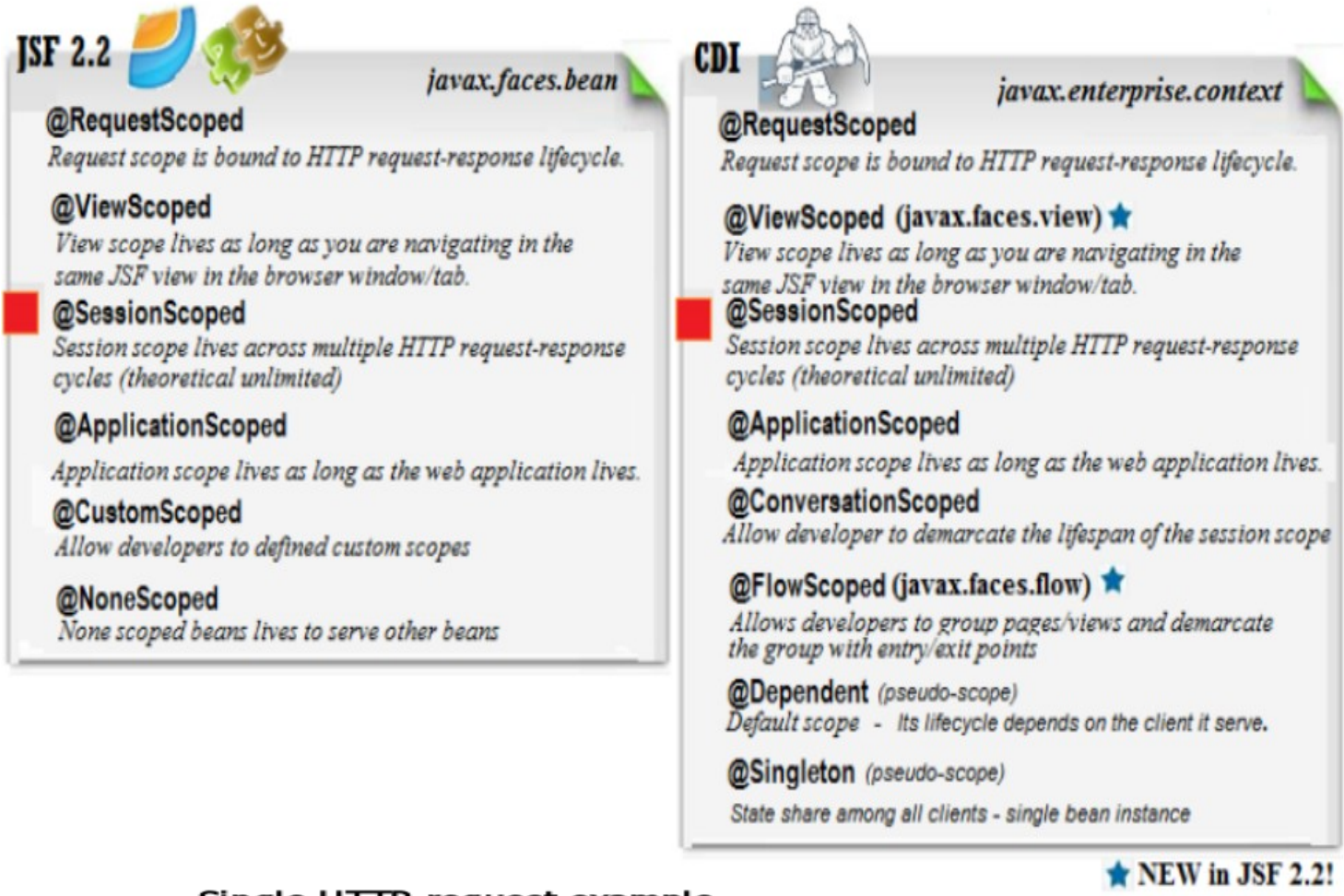
Input completion

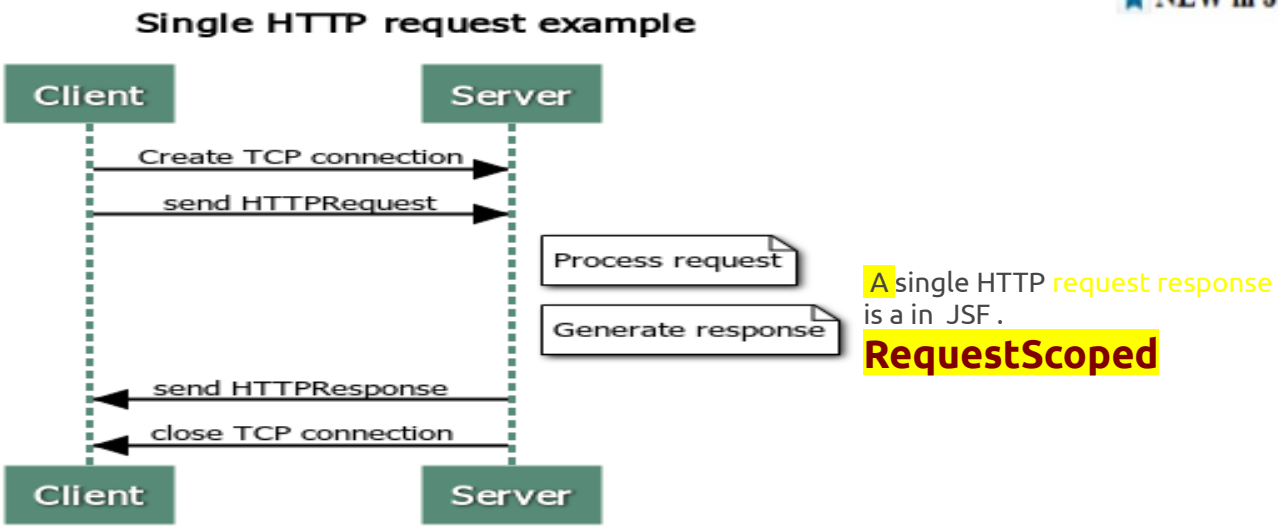
```
<h:inputText id="country" label="#{texts.yourCountry}"
    value="#{helloBean.country}"
    required="true"
    requiredMessage="#{texts.enterYourCountry}">
        <f:event type="preValidate"
                listener="#{countryCompleter.complete}" />
        <f:ajax event="keyup" render="country"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



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.





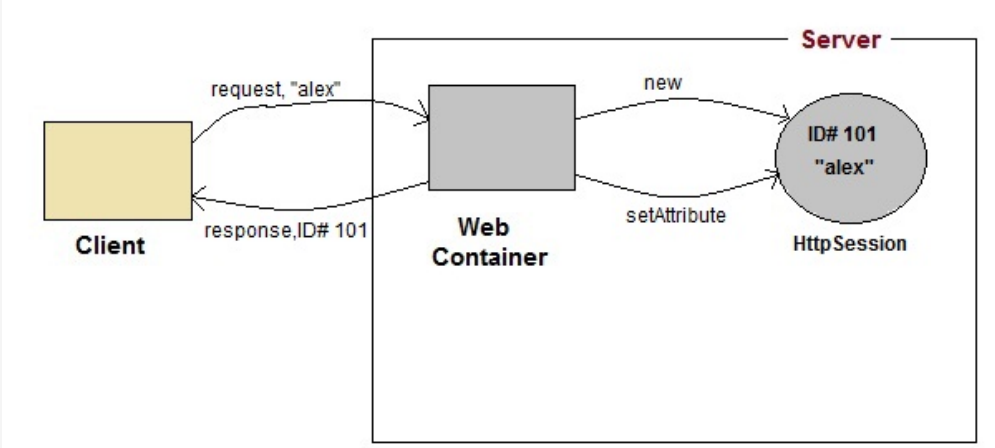
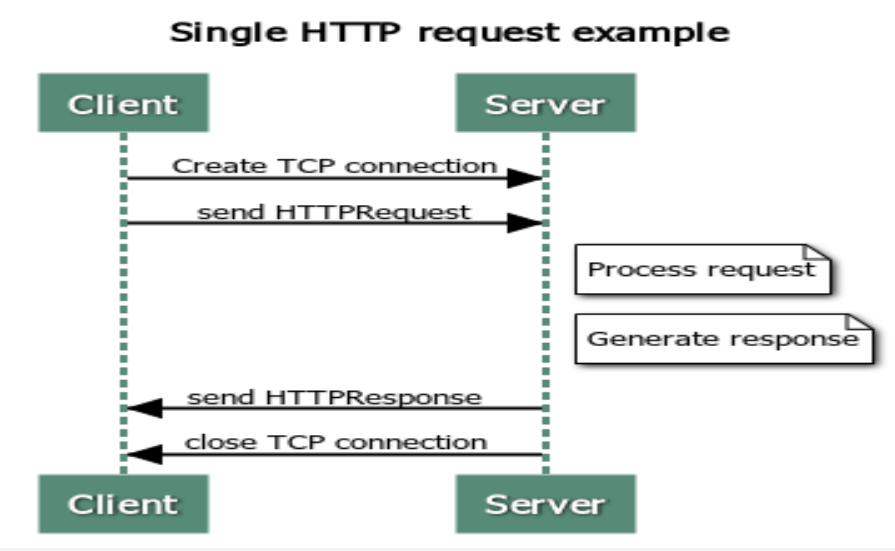
A single HTTP request response is a in JSF .
RequestScoped

https://www.javacodegeeks.com/2015/11/jsf-scopes-tutorial-jsfcdi-session-scope.html

In this tutorial you will learn in detail about the distinct bean scopes provided by CDI and how to use them.

Introduction

When a managed bean is initialized by CDI the bean will be initialized in a very specific scope. The scope in which the bean is initialized will determine its lifecycle. CDI provides the following bean scopes:

Scope	Description
ApplicationScoped	When a bean scope is defined as **ApplicationScoped** this means that only one instance of the bean will exist in the entire application. After bean initialization, every time a client requests an instance of this bean the container will always provide the same bean instance.
SessionScoped	**SessionScoped** beans are to be used inside a web application context. When a bean is configured as **SessionScoped** this means that an instance of this bean will exist per HTTP session. 
RequestScoped	**RequestScoped** beans are to be used inside a web application context. When a bean is configured as **RequestScoped** this means that an instance of this bean will exist per HTTP request. 

Scope	Description
ConversationScoped	**ConversationScoped** beans are to be used inside a web application context. As the name states, this kind of beans are used to represent a conversation between the client and the server. They may be used to keep state between multiple client AJAX requests or even between distinct page requests, if the client provides the conversation identifier to the server between page requests.

Update: Java EE 7 introduced a new set of CDI bean scopes, namely the TransactionScoped, FlowScoped and ViewScoped. More info on **Java EE 7 CDI bean scopes**.

CDI also provides two additional pseudo-scopes:

Pseudo-scope	Description
Singleton	As the name states, a **Singleton** bean represents a singleton so it will only exist one instance of a bean configured as **Singleton**.
Dependent	**Dependent** scope beans will be assigned the same scope as the bean they are being injected into, ie. if a **Dependent** scoped bean is injected into a **SessionScoped** bean, the injected bean will also be configured as **SessionScoped** and will exist until the current HTTP session exists.

Proxies

When a CDI managed bean is injected into another bean, the CDI container will not pass a reference to the injected bean itself but will instead pass a reference to a proxy. This proxy will handle all the calls made by the client to the injected bean transparently.

Consider a **SessionScoped** bean: If distinct HTTP sessions request a **SessionScoped** bean instance they will all be provided with the same proxy instance. When they make calls to the injected bean through the proxy, the proxy will know how to fetch the correct bean instance and then provide it to each HTTP session respectively.

Note: The **Singleton** pseudo-scope is an exception to the proxy mechanism. When a client request a Singleton bean it will be assigned a reference to the bean itself.

Serializable scopes

Managed CDI beans that are able to live for longer periods of time, such as **SessionScoped** and **ConversationScoped**, must implement the Serializable interface. This is because the container may need to free resources for an arbitrary reason and it may decide to persist a bean of this type into physical storage (or other persistent media type). When the persisted bean is once again needed for correct application execution, the container will fetch the bean and restore its state.

The Singleton pseudo-scope

When **Singleton** scoped beans are injected into client beans, the client beans will get a direct reference to the injected bean. This means that if the client bean is **Serializable** (ex: **SessionScoped** or **ConversationScoped**) it must also take care of the correct injected **Singleton** bean serialization. There are ways to assure that the injected bean remains a true singleton:

- Implement **writeReplace()** and **readResolve()** in the Singleton bean as specified by the standard Java serialization
- Make the reference to the **Singleton** bean transient. When the client bean is de-serialized by the container, the container will inject again the reference to the **Singleton** scoped bean.

The Dependent pseudo-scope

Dependent scope is the default CDI bean scope, ie. if a bean does not declare a specific scope it will be injected as a **Dependent**scoped bean. This means that it will have the same scope as the bean where it's being injected. **Dependent** scoped bean instances are never shared between clients. Every client will always fetch a new instance of a **Dependent** scoped bean.

Where is the View scope?

Update: Java EE 7 introduced ViewScoped beans. More info on **Java EE 7 CDI bean scopes** and **Java EE CDI ViewScoped example**.

If you are familiar with JSF bean scopes you may be wondering where does the **ViewScoped** bean type fits into this model. CDI does not offer any View scope but provides **ConversationScoped** beans instead.

With **ConversationScoped** beans we achieve the same functionality one usually needs from a View scoped JSF bean (maintain bean state between AJAX requests) and even more: it is possible to maintain the same conversation - or state - between distinct page requests.

Table of Contents

http://adfpractice-fedor.blogspot.ch/2012/02/understanding-immediate-attribute.html

Understanding the JSF Immediate Attribute

I think, we should be aware of the following key points:

Components with Immediate attribute set to true.

- Processing of any components with Immediate attribute is moved up to the Apply Request Values phase of the lifecycle. But there is a big difference in processing of editableValueHolder (inputText) and actionSource (commandButton) components.
- For EditableValueHolder components (like inputText) with Immediate attribute validation and valueChangeEvent delivering are done in Apply Request Values phase instead of usual Process Validation phase. This is the only change. The lifecycle is not stopped, it is going on, there is no any lifecycle phase skipping! Restore View->Apply Request Values->Process Validations->Update Model->Invoke App->Render Response
- For ActionSource components (like commandButton) with Immediate attribute, action event is delivered to Apply Request Values phase instead of usual Invoke Application phase. But!!! After that the lifecycle is jumping to the end, to Render Response phase. Validation and Model Update phases are skipped. Restore View->Apply Request Values->Render Response

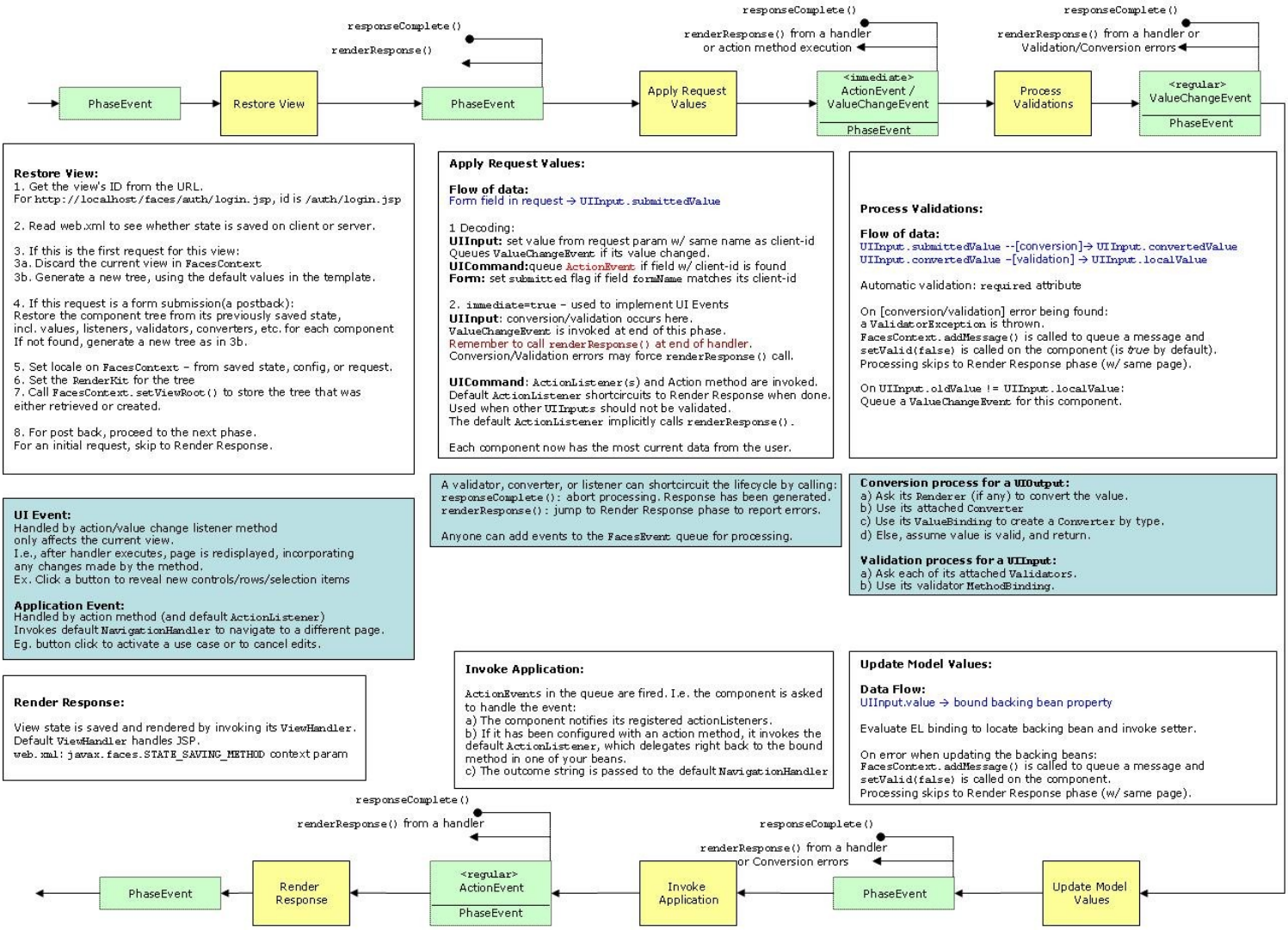
When to use immediate attribute set to true

Performance Tip:

There are some cases where setting the `immediate` attribute to `true` can lead to better performance:

- When you create a navigation train, and have a `commandNavigationItem` component in a `navigationPane` component, you should set the `immediate` attribute to `true` to avoid processing the data from the current page (train stop) while navigating to the next page. For more information, see Section 18.8.1, "How to Create the Train Model."
- If an input component value has to be validated before any other values, the `immediate` attribute should be set to `true`. Any errors will be detected earlier in the lifecycle and additional processing will be avoided.

How JSF Request works



Note: All JSF requests get forwarded to FacesServlet as specified in the web.xml.

FacesServlet

FacesServlet creates an object called FacesContext, which contains information necessary for request processing.
To be more precise, FacesContext contains the ServletContext, ServletRequest, and ServletResponse objects that are passed to the service method of FacesServlet by the Web container.
During processing, FacesContext is the object that is modified.

All the processing is done by Lifecycle. The FacesServlet servlet hands over control to the Lifecycle object.

NOTE : You can specify a context parameter saveStateInClient with a value of true to force JSF to save state in the client as opposed to saving it in the server.
If you choose to do so, you must add the following context-param element before the servlet element in your deployment descriptor.

```
<context-param>
 <param-name>saveStateInClient</param-name><param-value>false</param-value>
</context-param>
```

FacesServlet web.xml

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

Example: Cancel Button with facesMessage

```
public String cancel() {
 newUser = new User();

 FacesContext.getCurrentInstance().addMessage(null,
   new FacesMessage("Cancelled new user"));

 return "/home.xhtml";
}
```

http://www.cs-repository.info/2014/12/jsf-notes.html

The JSF 2 Expression Language

JSF 2.2 Version

Originals of slides and source code for examples: http://www.coreservlets.com/JSF-Tutorial/jsf2/
Also see the PrimeFaces tutorial – http://www.coreservlets.com/JSF-Tutorial/primefaces/
and customized JSF2 and PrimeFaces training courses – http://courses.coreservlets.com/jsf-training.html

PRIMEFACES

JSF

For live training on JSF 2, PrimeFaces, or other Java EE topics, email hall@coreservlets.com
Marty is also available for consulting and development support

Taught by the author of *Core Servlets and JSP*, this tutorial, and JSF 2.2 version of *Core JSF*. Available at public venues, or customized versions can be held on-site at your organization.

- Courses developed and taught by Marty Hall
  - JSF 2, PrimeFaces, Ajax, jQuery, Spring MVC, JSP, Android, general Java, Java 8 lambdas/streams, GWT, custom topic mix
  - Courses available in any location worldwide. Maryland/DC area companies can also choose afternoon/evening courses.
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Hadoop, Spring, Hibernate/JPA, RESTful Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- **Motivating use of the expression language**
  - Comparing to the JSF 1.*x* and JSP 2.0 ELs
- **Simplified testing of EL capabilities**
- **Accessing bean properties**
  - Direct
  - Nested
- **Submitting bean properties**
  - Expressions in output values
  - Expressions in submission values
  - Expressions for action controllers
- **Accessing collection elements**
- **Using implicit objects and operators**
- **Conditionally rendering output**
- **Passing arguments to methods**

5

Overview

6

The Expression Language

- **JSP scripting not supported in facelets**
  - So, you need a way to indirectly invoke Java
- **Quick examples**
  - #{employee.firstName}
    - Call getFirstName on bean named employee. Output it.
  - <h:inputText value="#{employee.firstName}"/>
    - When form displayed, call getFirstName, and if non-empty, fill it in as initial value of textfield.
    - When form submitted, validate value and if it is OK, pass value to the setFirstName method
  - #{employee.addresses[0].zip}
    - Call getAddresses on bean named employee (which should return an array or list), then take first entry, then call getZip on that, then output it

7

Advantages of the Expression Language (Very Important)

- **Shorthand notation for bean properties**
  - To reference the result of the getCompanyName method of a managed bean named company, you use #{company.companyName}.
  - To reference the firstName property of the president property of a managed bean named company, you use #{company.president.firstName}.
- **Simple access to collection elements**
  - To reference an element of an array, List, or Map, you use #{someBean.someProperty[indexOrKey]}.
    - E.g., #{person.friends[2]}

8

Advantages of the EL (Moderately Important)

- **A small but useful set of simple operators**
  - To manipulate objects within EL expressions, you can use any of several arithmetic, relational, logical, or empty-testing operators.
- **Conditional output**
  - To choose among output options:
    - #{test ? option1 : option2}
    - <h:someElement … rendered="#{test}"/>
    - <ui:fragment rendered="…">…</ui:fragment>
      - We will give very brief examples in this tutorial section. The later section on looping with ui:repeat will give more details.

9

Advantages of the EL (Less Important)

- **Predefined variables (implicit objects)**
  - To access request params, cookies, HTTP headers, and other standard types of request data, you can use one of several predefined implicit objects.
- **Passing arguments**
  - Version 2.1 of the EL lets you pass arbitrary arguments to methods. Works only in Java EE 6 or other servers that support EL 2.1. *Not part of JSF 2 itself.*
    - E.g, works in Tomcat 7 but not Tomcat 6, even though JSF 2 works in both.
- **Empty values instead of error messages**
  - In most cases, missing values or NullPointerExceptions result in empty strings, not thrown exceptions.

10

JSF vs. JSP ELs

Feature	JSF 2.0 EL	JSF 1.*x* EL (with JSP)	JSP 2.0 EL
Format	#{blah} (immediate output values could be accessed with ${blah})	#{blah}	${blah}
Where used	Anywhere in facelets page Eg: #{customer.firstName}	Only in attributes of JSF tags. Eg: <h:outputText value= "#{customer.firstName}"/>	Anywhere in page Eg: ${customer.firstName}
Represents	Output data, later location for submitted data. Eg: <h:inputText value= "#{customer.firstName}"/>	Output data, later location for submitted data. Eg: <h:inputText value= "#{customer.firstName}"/>	Output data. Eg ${customer.firstName}
Where it looks for beans	Request, session, application (etc.) scopes and managed bean defs.	Request, session, application (etc.) scopes and managed bean defs.	Request, session, application scopes.
Declaration type	None needed for simplest usage. xlmns declaration for h:, ui:, f: tags.	@taglib	None needed
Environments	Java EE 6 servers or servlet 2.5 servers with JSF 2.0 JARs.	Java EE 5 servers or servlet 2.4 servers with JSF 1.*x* JARs.	Servlet 2.4+ servers

11

Simplified Testing of EL Capabilities

12

Simplifying Testing of the EL

- **JSP**
  - Checks existing scopes (request, session, etc.). If not found, gives up.
- **JSF**
  - Checks existing scopes (request, session, etc.). If not found, looks for managed bean definition of that name (either from @ManagedBean or from faces-config.xml).
- **Implication for testing and experimenting**
  - You can create a simple bean and a simple standalone page to test it. No form, no action controller, no results page. Great for experimenting with EL features.
    - See next page for an example

13

Simplifying Testing of the EL: Example

Bean

```
@ManagedBean
public class SimpleBean {
  private String[] colors =
    { "red", "orange", "yellow" };

  public String getMessage() {
    return("Hello, World");
  }

  public String[] getColors() {
    return(colors);
  }
}
```

Standalone Test Page

```
<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
...
<ul>
  <li>Message: #{simpleBean.message}</li>
  <li>First color: #{simpleBean.colors[0]}</li>
</ul>
...
```

14

Outputting Simple Bean Properties

Outputting Simple Bean Properties

- **Format**
  - #{varName.propertyName}
  - <h:outputText value="#{varName.propertyName}" …/>
    - For new JSF 2 code, top version is usually used unless you need some other attribute of h:outputText (e.g. “id”, “rendered”, or “escape”)
- **Interpretation**
  - First, find varName
    - Search for “varName” in all defined scopes, from most specific to most general (request, session, application, in that order for standard Web app scopes). Then look in managed bean defs and instantiate if found.
  - Call getPropertyName and output the result
    - This must be a normal zero-arg accessor method. If boolean, name of method could be *is*PropertyName

Bean Properties Example: Java Code

```
@ManagedBean
@ApplicationScoped
public class TestBean1 {
  private Date creationTime = new Date();
  private String greeting = "Hello";

  public Date getCreationTime() {
    return(creationTime);
  }

  public String getGreeting() {
    return(greeting);
  }

  public double getRandomNumber() {
    return(Math.random());
  }
}
```

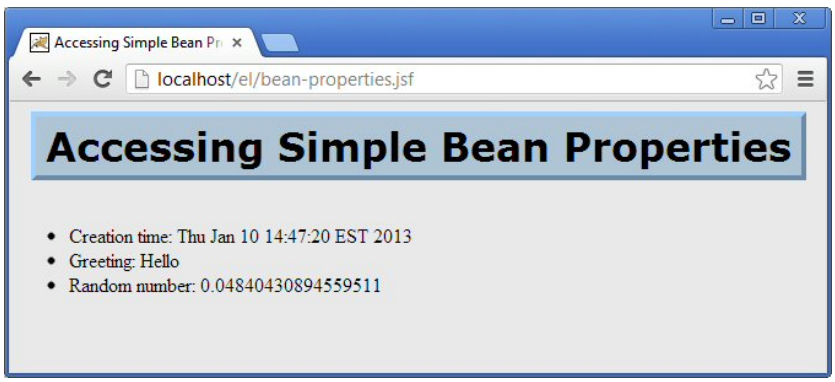
17

Bean Properties Example: Facelets Code

```
<!DOCTYPE …>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head><title>Accessing Simple Bean Properties</title>
<link href="./css/styles.css"
      rel="stylesheet" type="text/css"/>
</h:head>
<h:body>
…
<ul>
  <li>Creation time: #{testBean1.creationTime}</li>
  <li>Greeting: #{testBean1.greeting}</li>
  <li>Random number: #{testBean1.randomNumber}</li>
</ul>
</h:body></html>
```

18

Bean Properties Example: Result



19

Accessing Nested Bean Properties

20

Nested Bean Properties

- **Format**
  - #{var.prop1.prop2.prop3}
  - <h:outputText value="#{var.prop1.prop2.prop3}" …/>
    - Again, use this form only if you need some extra attribute of h:outputText such as “id”, “rendered”, or “escape”
- **Interpretation**
  - First, find var
    - Same as before. Look in existing scopes (narrowest to widest). Use if found. If not found, look in managed bean defs and instantiate.
  - Call getProp1 on bean
  - Call getProp2 on result of getProp1
  - Call getProp3 on result of getProp2
    - And then output the result

21

Nested Properties Example: Name

```
public class Name {
  private String firstName, lastName;

  public Name(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public String getFirstName() {
    return(firstName);
  }

  public void setFirstName(String newFirstName) {
    firstName = newFirstName;
  }
  ...
}
```

22

Nested Properties Example: Company

```
public class Company {
  private String companyName, business;

  public Company(String companyName, String business) {
    this.companyName = companyName;
    this.business = business;
  }

  public String getCompanyName() { return(companyName); }

  public void setCompanyName(String newCompanyName) {
    companyName = newCompanyName;
  }
  ...
}
```

23

Nested Properties Example: Employee

```
public class Employee {
  private Name name;
  private Company company;

  public Employee(Name name, Company company) {
    this.name = name;
    this.company = company;
  }

  public Name getName() { return(name); }

  public Company getCompany() { return(company); }

  ...
}
```

24

Nested Properties Example: Employee1

```
@ManagedBean
public class Employee1 extends Employee {
  public Employee1() {
    super(new Name("Marty", "Hall"),
          new Company("coreservlets.com",
                      "Customized Java EE and Ajax Training"));
  }
}
```

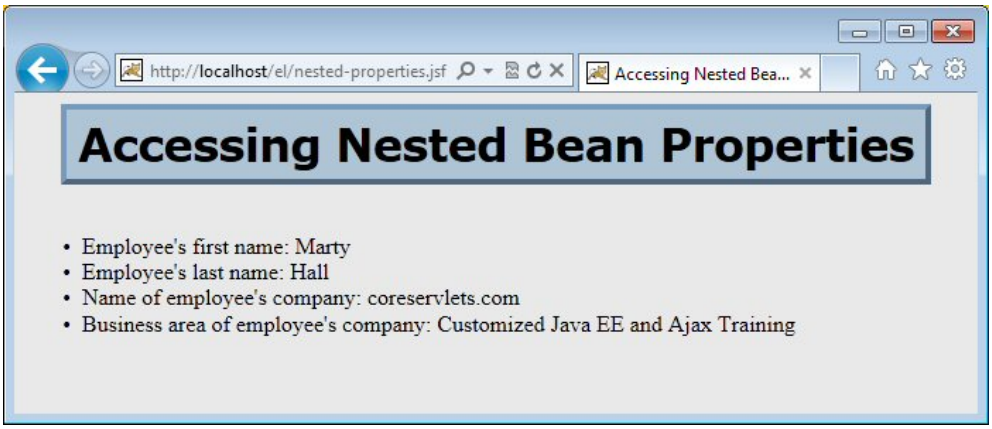
25

Nested Properties Example: Facelets Code

```
…
<ul>
  <li>Employee's first name:
      #{employee1.name.firstName}</li>
  <li>Employee's last name:
      #{employee1.name.lastName}</li>
  <li>Name of employee's company:
      #{employee1.company.companyName}</li>
  <li>Business area of employee's company:
      #{employee1.company.business}</li>
</ul>
…
```

26

Nested Properties Example: Result



27

Submitting Bean Properties

28

Three Uses of #{...}

- **Designating output value**
  - #{employee.address} or <h:outputText value="#{employee.address}"/>
    - Anytime accessed, means to output getAddress
  - <h:inputText value="#{employee.address}"/>
    - When form initially displayed, means to prepopulate field. Call getAddress and put value in field if non-empty.
- **Designating submitted value**
  - <h:inputText value="#{employee.address}"/>
    - When form submitted, designates where value stored. Pass textfield value to setAddress.
- **Designating method call after submission**
  - <h:commandButton value="Button Label" action="#{employee.processEmployee}"/>
    - When form submitted, designates action handler. This is exact method name, not a shorthand for it.

29

Understanding Getter vs. Setter Method Correspondence

- **Example**
  - <h:inputText value="#{myBean.a.b.c.d}"/>
- **When displaying form**
  - Find or instantiate myBean. Call getA. Call getB on result. Call getC on that result. Call getD on that result. If non-empty use as initial value of textfield.
- **When submitting form**
  - Find myBean (instantiate new version if in request scope). Call getA. Call getB on result. Call getC on that result. Then pass submitted value to the setD method of that result.
    - Point: only *final* one becomes setter on submission.
    - This assumes value passes validation. Discussed later.

30

Submitting Properties Example: Employee

```
public class Employee {
  private Name name;
  private Company company;

  …

  public String processEmployee() {
    if (Math.random() < 0.5) {
      return("accepted");
    } else {
      return("rejected");
    }
  }
}
```

31

Submitting Properties Example: Facelets Code for Form

```
…
<h:form>
<h:panelGrid columns="2" styleClass="formTable">
Your first name:
<h:inputText value="#{employee1.name.firstName}"/>
Your last name:
<h:inputText value="#{employee1.name.lastName}"/>
Name of your company:
<h:inputText value="#{employee1.company.companyName}"/>
Business area of your company:
<h:inputText value="#{employee1.company.business}"
             size="38"/>
</h:panelGrid>
<h:commandButton value="Process"
                 action="#{employee1.processEmployee}"/>
</h:form>
```

h:panelGrid is a shortcut for making an HTML table. More details in the lecture on validating form data.

32

Submitting Properties Example: Input Page Initial Result

33

Submitting Properties Example: accepted.xhtml

```
…
<table border="5" align="center">
  <tr><th class="title">Employee Accepted</th></tr>
</table>
<p/>
<ul>
  <li>Employee's first name:
      #{employee1.name.firstName}</li>
  <li>Employee's last name:
      #{employee1.name.lastName}</li>
  <li>Name of employee's company:
      #{employee1.company.companyName}</li>
  <li>Business area of employee's company:
      #{employee1.company.business}</li>
</ul>
…
```

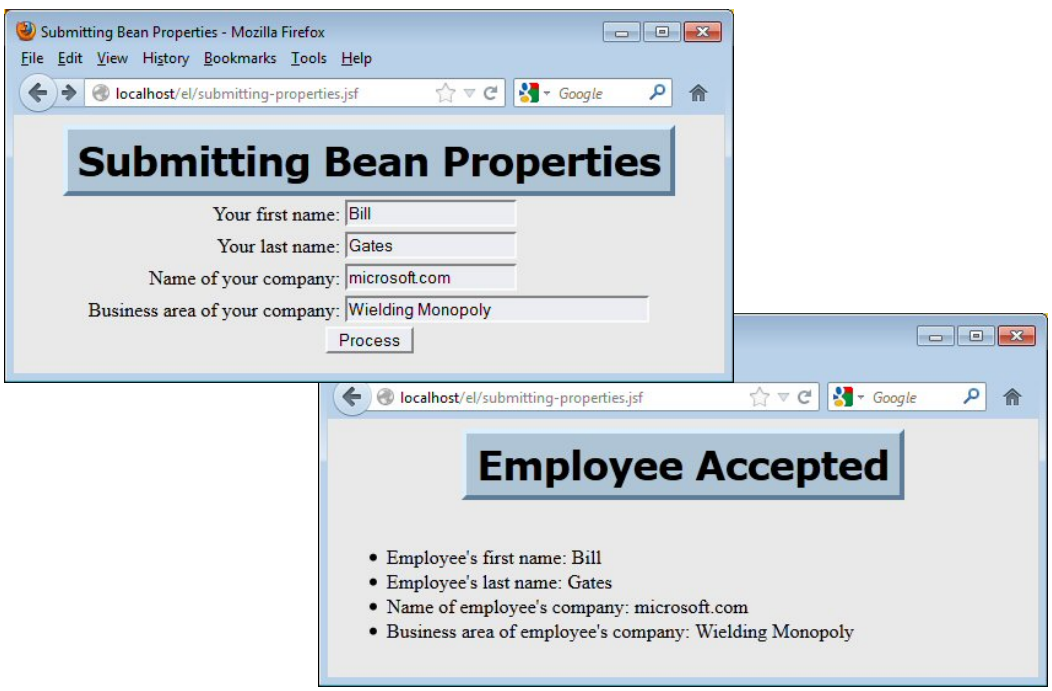
34

Submitting Properties Example: rejected.xhtml

```
…
<table border="5" align="center">
  <tr><th class="title">Employee Rejected</th></tr>
</table>
<p/>
<ul>
  <li>Employee's first name:
      #{employee1.name.firstName}</li>
  <li>Employee's last name:
      #{employee1.name.lastName}</li>
  <li>Name of employee's company:
      #{employee1.company.companyName}</li>
  <li>Business area of employee's company:
      #{employee1.company.business}</li>
</ul>
…
```

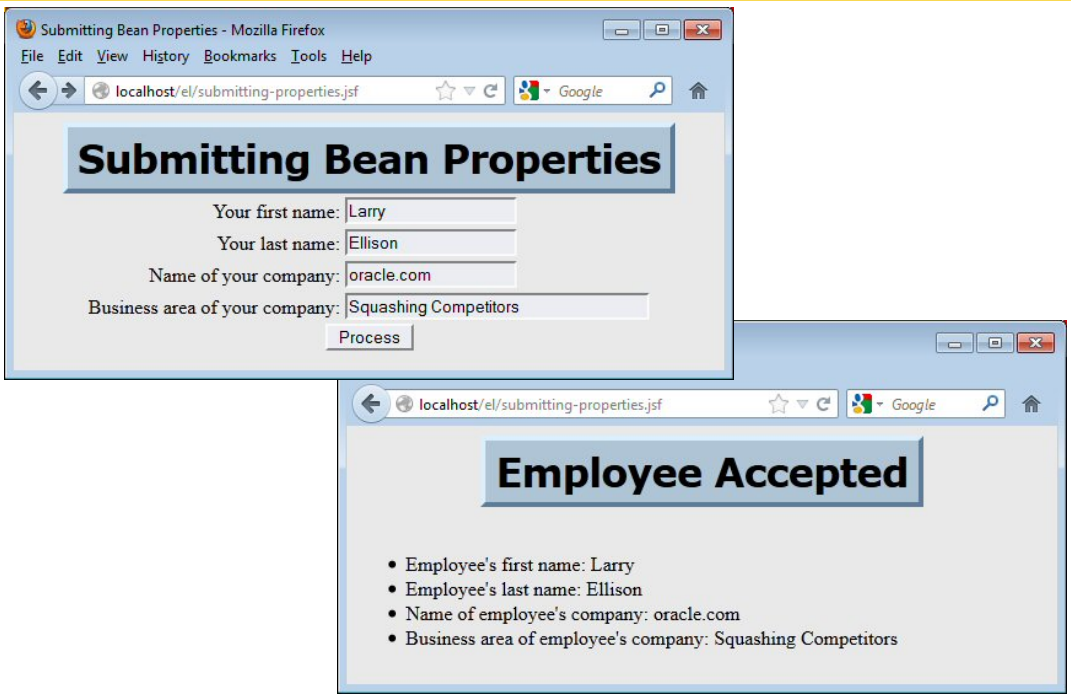
35

Submitting Properties Example: Results



36

Submitting Properties Example: Results (Continued)



Accessing Collections

Equivalence of Dot and Array Notations

- **Equivalent forms**
  - #{name.property}
    - Only legal if “property” would be legal Java variable name
  - #{name["property"]}
- **Reasons for using bracket notation**
  - To access arrays, lists, and other collections
    - See upcoming slides
  - To calculate the property name at request time.
    - #{name1[name2]} (no quotes around name2)
  - To use names that are illegal as Java variable names
    - #{foo["bar-baz"]}
    - #{foo["bar.baz"]}

39

Using the [ ] Form

- **Works for**
  - Array. Equivalent to
    - theArray[index] (getting and setting)
  - List. Equivalent to
    - theList.get(index) or theList.set(index, *submittedVal*)
  - Map. Equivalent to
    - theMap.get(key) or theMap.put(key, *submittedVal*)
- **Equivalent forms (for Maps)**
  - #{stateCapitals["maryland"]}
  - #{stateCapitals.maryland}
  - But you can’t use this for lists (numbers are not legal Java variables names, so #{listVar.2} is illegal). And not all hash table keys are legal variable names. So, use brackets.

40

Collections Example: Purchases

```
@ManagedBean
public class Purchases {
  private String[] cheapItems =
    { "Gum", "Yo-yo", "Pencil" };
  private List<String> mediumItems =
    new ArrayList<>();
  private Map<String,String> valuableItems =
    new HashMap<>();
  private boolean isEverythingOK = true;

  public Purchases() {
    mediumItems.add("iPod");
    mediumItems.add("GameBoy");
    mediumItems.add("Cell Phone");
    valuableItems.put("low", "Lamborghini");
    valuableItems.put("medium", "Yacht");
    valuableItems.put("high", "JSF Training Course");
  }
```


Collections Example: Purchases (Continued)

```
  public String[] getCheapItems() {
    return(cheapItems);
  }

  public List<String> getMediumItems() {
    return(mediumItems);
  }

  public Map<String,String> getValuableItems() {
    return(valuableItems);
  }
```

Collections Example: Purchases (Continued)

```
  public String purchaseItems() {
    isEverythingOK = Utils.doBusinessLogic(this);
    isEverythingOK = Utils.doDataAccessLogic(this);
    if (isEverythingOK) {
      return("purchase-success");
    } else {
      return("purchase-failure");
    }
  }
}
```

43

Collections Example: Utils

```
public class Utils {
  public static boolean doBusinessLogic
                          (PurchaseBean bean) {
    // Business logic omitted
    return(Math.random() > 0.1);
  }

  public static boolean doDataAccessLogic
                          (PurchaseBean bean) {
    // Database access omitted
    return(Math.random() > 0.1);
  }
}
```

44

Collections Example: using-collections.xhtml

```
...
<h:form>
<ul>
<li><b>Cheap Items</b>
<ol>
<li>
  <h:inputText value="#{purchases.cheapItems[0]}"/>
</li>
<li>
  <h:inputText value="#{purchases.cheapItems[1]}"/>
</li>
<li>
  <h:inputText value="#{purchases.cheapItems[2]}"/>
</li>
</ol></li>
```

This example uses explicit indices. See the tutorial section on looping to see how to redo this example with ui:repeat and a variable for the index.

45

Collections Example: using-collections.xhtml (Continued)

```
<li><b>Medium Items</b>
<ol>
<li>
  <h:inputText value="#{purchases.mediumItems[0]}"/>
</li>
<li>
  <h:inputText value="#{purchases.mediumItems[1]}"/>
</li>
<li>
  <h:inputText value="#{purchases.mediumItems[2]}"/>
</li>
</ol></li>
```

46

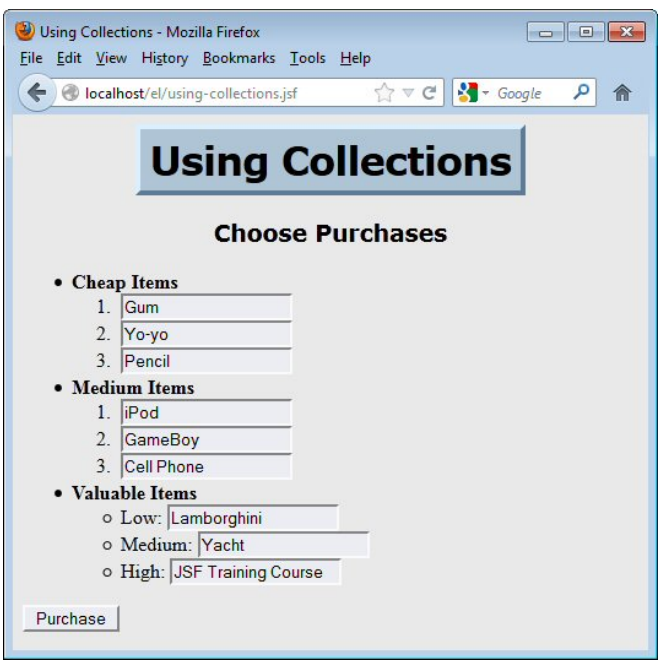
Collections Example: using-collections.xhtml (Continued)

```
<li><b>Valuable Items</b>
<ul>
<li>Low:
  <h:inputText value='#{purchases.valuableItems["low"]}'/>
</li>
<li>Medium:
  <h:inputText
   value='#{purchases.valuableItems["medium"]}'/>
</li>
<li>High:
  <h:inputText
   value='#{purchases.valuableItems["high"]}'/>
</li>
</ul></li>
</ul>
<h:commandButton value="Purchase"
                 action="#{purchases.purchaseItems}"/>
```

Since I use double quotes around the Map key, I use single quotes here.

47

Collections Example: Input Page Initial Result



48

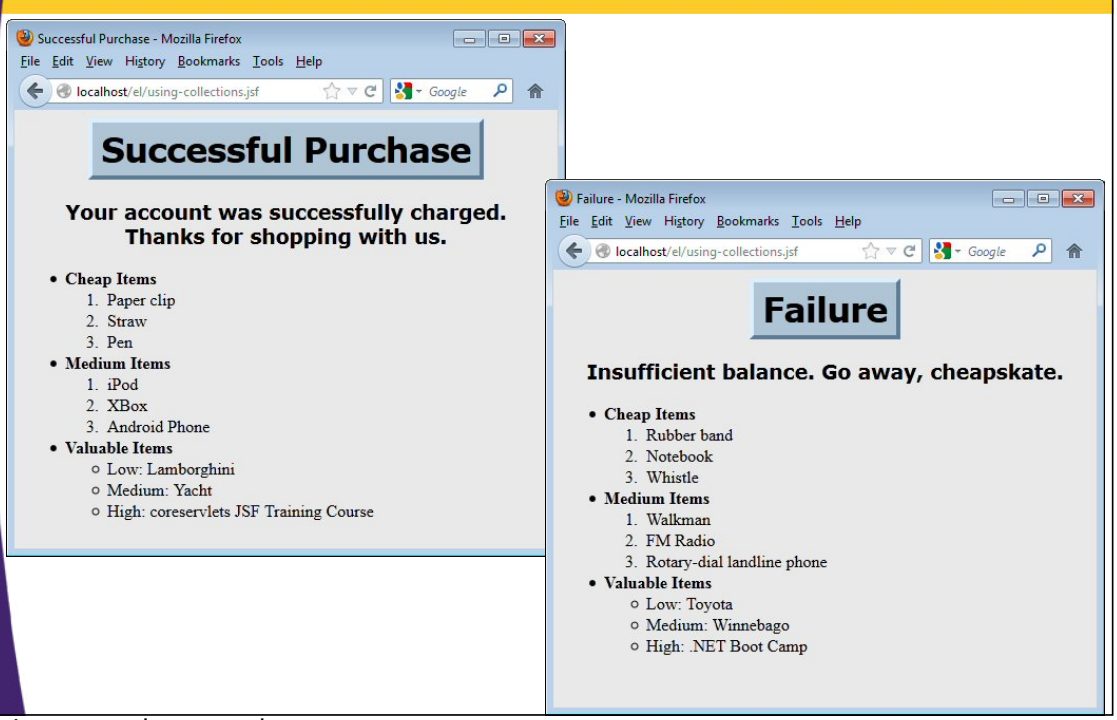
Submitting Properties Example: purchase-success.xhtml

purchase-failure.xhtml is very similar.

```
…
<li><b>Cheap Items</b>
<ol>
<li>#{purchases.cheapItems[0]}</li>
<li>#{purchases.cheapItems[1]}</li>
<li>#{purchases.cheapItems[2]}</li>
</ol></li>
<li><b>Medium Items</b>
<ol>
<li>#{purchases.mediumItems[0]}</li>
<li>#{purchases.mediumItems[1]}</li>
<li>#{purchases.mediumItems[2]}</li>
</ol></li>
<li><b>Valuable Items</b>
<ul>
<li>Low: #{purchases.valuableItems["low"]}</li>
<li>Medium: #{purchases.valuableItems["medium"]}</li>
<li>High: #{purchases.valuableItems["high"]}</li>
</ul></li>
</ul>
```

49

Submitting Properties Example: Results



50

Implicit Objects and Operators

JSF EL Has Almost the Same Predefined Variables as JSP 2

- **Predefined variables**
  - facesContext. The FacesContext object.
    - E.g. #{facesContext.externalContext.remoteUser}
  - param. Request params.
    - E.g. #{param.custID}
  - header. Request headers.
    - E.g. #{header.Accept} or #{header["Accept"]}
    - #{header["Accept-Encoding"]}
  - cookie. Cookie object (not cookie value).
    - E.g. #{cookie.userCookie.value} or #{cookie["userCookie"].value}
  - request, session
    - #{request.contextPath}, #{request.queryString}, #{session.id}
      - #{request.contextPath} useful for making relative URLs. See templating section.
  - initParam. Context initialization param.
- **Problem**
  - Using implicit objects works poorly with MVC model. You usually want to use these values in the Java code, not in the facelets pages.

Example: Implicit Objects (Facelets Code)

```
…
<ul>
  <li><b>Context path:</b>
      #{request.contextPath}</li>
  <li><b>Value of JSESSIONID cookie:</b>
      #{cookie.JSESSIONID.value}</li>
  <li><b>The "test" request parameter:</b>
      #{param.test}</li>
  <li><b>User-Agent request header:</b>
      #{header["User-Agent"]}</li>
</ul>
…
```

53

Example: Implicit Objects (Result)

54

Expression Language Operators

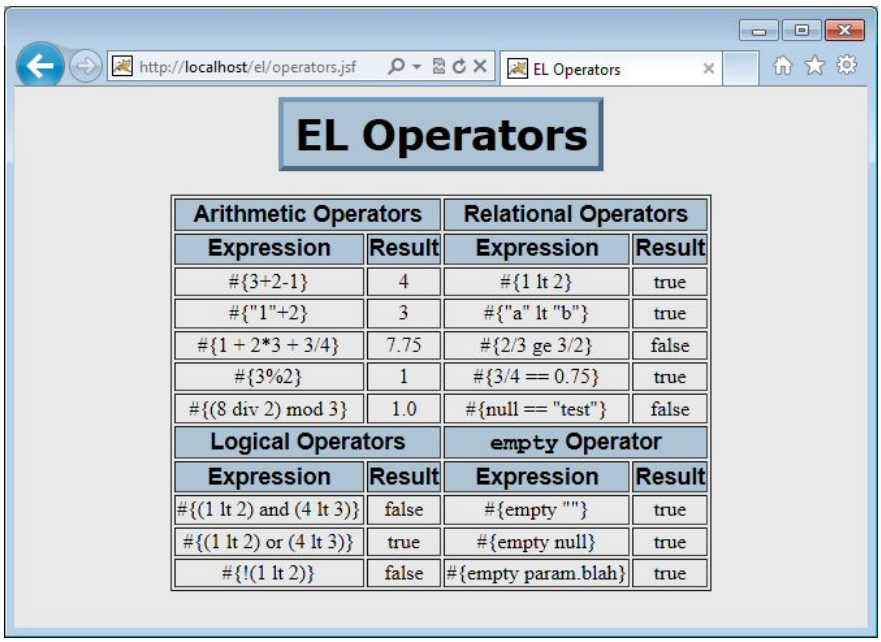
- **Arithmetic**
  - + - * / div % mod
- **Relational**
  - == *or* eq, != *or* ne, < *or* lt, > *or* gt, <= *or* le, >= *or* ge
    - Note: in many contexts in XML, using the operators that contain "<" is illegal. So, you usually use lt instead of <, le instead of <=, etc.
- **Logical**
  - && and || or ! Not
- **Empty**
  - empty
    - True for null, empty string, empty array, empty list, empty map. False otherwise.
- **Note**
  - Use operators sparingly to preserve MVC model

Example: Operators

```
…
<table border="1" align="center">
  …
 <tr align="center">
    <td>\#{3+2-1}</td><td>#{3+2-1}</td>
    <td>\#{1 lt 2}</td><td>#{1 lt 2}</td></tr>
  <tr align="center">
    <td>\#{"1"+2}</td><td>#{"1"+2}</td>
    <td>\#{"a" lt "b"}</td><td>#{"a" lt "b"}</td></tr>
  <tr align="center">
    <td>\#{1 + 2*3 + 3/4}</td><td>#{1 + 2*3 + 3/4}</td>
    <td>\#{2/3 ge 3/2}</td><td>#{2/3 ge 3/2}</td></tr>
  <tr align="center">
    <td>\#{3%2}</td><td>#{3%2}</td>
    <td>\#{3/4 == 0.75}</td><td>#{3/4 == 0.75}</td></tr>
  …
</table>
…
```

\#{blah} is taken literally. The backslash prevents EL evaluation.

Example: Operators (Result)



Arithmetic Operators		Relational Operators	
Expression	Result	Expression	Result
#{3+2-1}	4	#{1 lt 2}	true
#{"1"+2}	3	#{"a" lt "b"}	true
#{1 + 2*3 + 3/4}	7.75	#{2/3 ge 3/2}	false
#{3%2}	1	#{3/4 == 0.75}	true
#{(8 div 2) mod 3}	1.0	#{null == "test"}	false
Logical Operators		**`empty` Operator**	
Expression	Result	Expression	Result
#{(1 lt 2) and (4 lt 3)}	false	#{empty ""}	true
#{(1 lt 2) or (4 lt 3)}	true	#{empty null}	true
#{!(1 lt 2)}	false	#{empty param.blah}	true

Conditional Output

Conditional Text in JSF

- **Alternatives**
  - #{someCondition ? simpleVal1 : simpleVal2}
  - <h:outputText value="#{someValue}"
    rendered="#{someCondition}"/>
    - Or, in general, use h:blah and the “rendered” attribute
  - <ui:fragment rendered="#{someCondition}">
    <someHTML>…</someHTML>
    </ui:fragment>
- **Note**
  - More detailed examples shown in tutorial section on looping in facelets pages

59

Conditional Text with #{ condition ? val1 : val2 }

- **Idea**
  - The EL directly supports limited conditional output via the ternary operator (test ? thenResult : elseResult). Supply a boolean for the test, put conditional content after the “?” and/or the “:”. Values can be literal strings or EL expressions, but they cannot contain HTML tags.
    - Note: you are not permitted to omit the “else” part!
- **Examples**
  - <td class="#{customer.balance < 0 ? 'red': 'black'}">
  - #{ !status.last ? ',' : ''}
- **When used**
  - When you are outputting simple text (no HTML).

If you want to output HTML, you could use the ternary operator within h:outputText and supply escape="false". But in that case, one of the other two upcoming alternatives is probably simpler.

60

Conditional Text with h:outputText and "rendered"

- **Idea**
  - Pass a boolean to the "rendered" attribute, put conditional content in "value" attribute. The value can be a literal string or an EL expression, but the literal string cannot contain HTML tags.
- **Examples**
  - <h:outputText rendered="#{!status.last}" value=","/>
  - <h:outputText rendered="#{status.index > 5}"
    value="#{user.someWarning}"
    escape="false"/>

    The assumption here is that the getSomeWarning method outputs a string containing HTML tags. If so, the escape="false" is needed to prevent JSF from turning the < into < and so forth.
- **When used**
  - When you are outputting simple text (no HTML) or when the HTML comes from a bean.

61

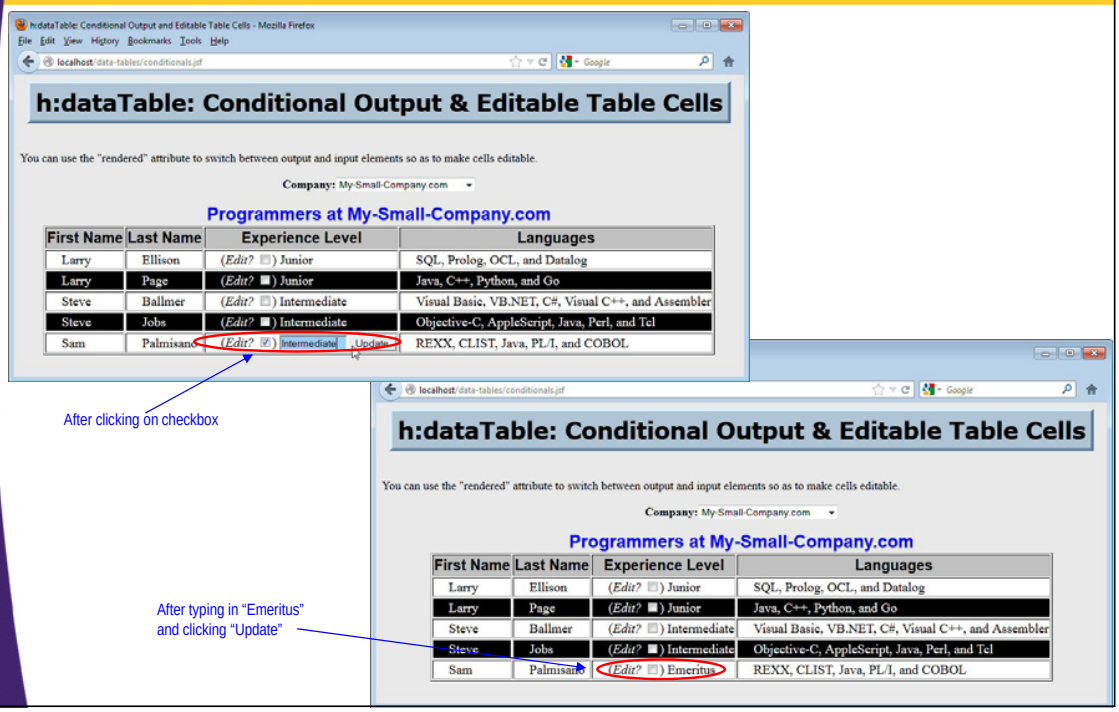
More on "rendered" Attribute

- **Almost all h:*blah* elements use "rendered"**
  - So, you can insert almost any JSF element conditionally.
- **Example**
  - Insert either textfield followed by button *or* simple value (full example in tutorial section on h:dataTable)

```
<h:inputText value="#{programmer.level}" size="12"
             rendered="#{programmer.levelEditable}"/>
<h:commandButton value="Update"
                 rendered="#{programmer.levelEditable}">
    <f:ajax render="@form" execute="@form"/>
</h:commandButton>
<h:outputText value="#{programmer.level}"
              rendered="#{!programmer.levelEditable}"/>
```

62

Example: Use of “rendered”



63

Conditional Text with ui:fragment

- **Idea**
  - Pass a boolean to the “rendered” attribute, put conditional content in body content. The value can be a literal string or an EL expression, and the literal string *can* contain HTML tags.
- **Example**
  - <ui:fragment rendered="#{!status.last}">
    ,
    </ui:fragment> Outputs a bold comma after every entry except the last
- **When used**
  - When you are outputting literal HTML.
    - Can always be used in lieu of h:outputText, but if no HTML, h:outputText is more succinct.
- **Note: define the ui namespace at top**
  - <html … xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

64

Passing Arguments to Methods

Big Idea

- **EL version 2.2 lets you call regular methods**
  - Rather than only zero-arg accessor methods
- **Syntax**
  - Basic syntax is straightforward
    - #{someBean.someMethod(arg1, arg2)}
  - The arguments can also be EL expressions
- **Cautions**
  - Use sparingly: put complexity in Java, not facelets
  - Works only in EL 2.2. Not part of JSF 2.0 itself.
    - Server must support servlets 3.0
      - All Java EE 6 servers automatically do
    - So, works in Glassfish 3, JBoss 6, and Tomcat 7. Fails in Tomcat 6, JBoss 5, and other servlet 2.5 engines.

Method Args: Java Code

```
@ManagedBean
@ApplicationScoped
public class TestBean2 {
  private final String HELLO_ENGLISH = "Hello!";
  private final String HELLO_SPANISH = "¡Hola!";

  public String greeting(boolean useSpanish) {
    if (useSpanish) {
      return(HELLO_SPANISH);
    } else {
      return(HELLO_ENGLISH);
    }
  }

  public String greeting() {
    return(greeting(false));
  }

  public double randomNumber(double range) {
    return(range * Math.random());
  }
```

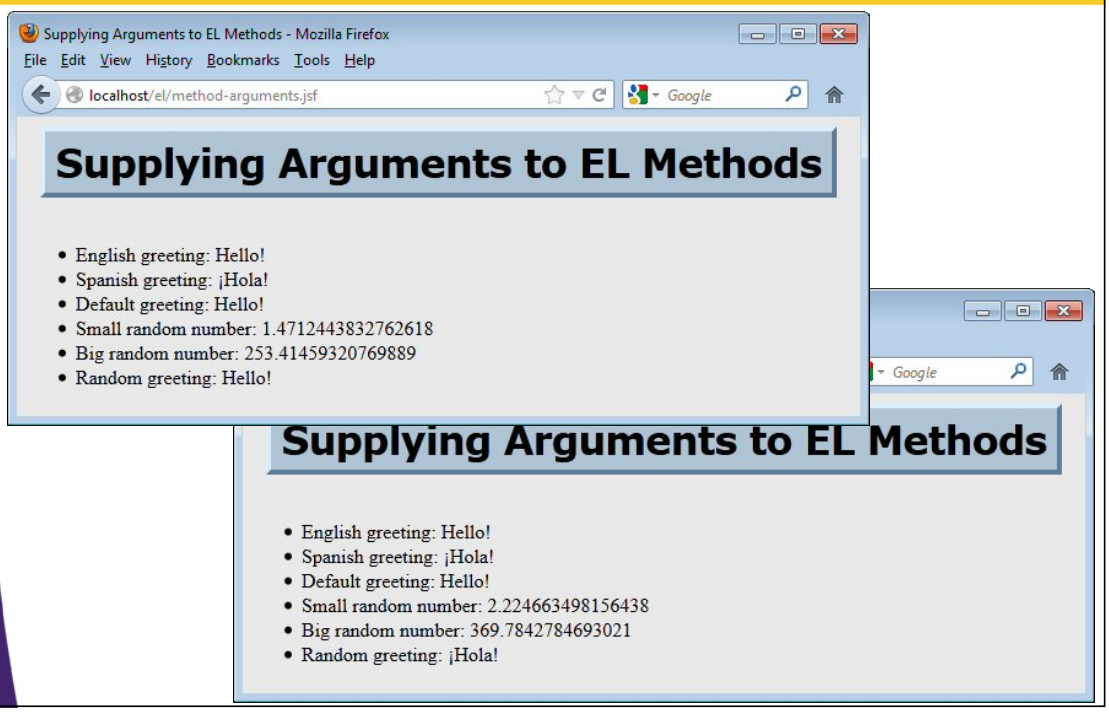
67

Method Args: Facelets Code

```
…
<ul>
  <li>English greeting: #{testBean2.greeting(false)}</li>
  <li>Spanish greeting: #{testBean2.greeting(true)}</li>
  <li>Default greeting: #{testBean2.greeting()}</li>
  <li>Small random number: #{testBean2.randomNumber(5)}</li>
  <li>Big random number: #{testBean2.randomNumber(500)}</li>
  <li>Random greeting:
      #{testBean2.greeting(testBean2.randomNumber(200) gt 100)}
  </li>
</ul>
…
```

68

Method Args: Results



Wrap-Up

Summary

- **Outputting bean properties**
  - #{customer.company.name}
  - <h:outputText value="#{customer.company.name}"/>
    - h:outputText needed only when using rendered, escape, etc.
- **Textfields and other input elements**
  - <h:inputText value="#{customer.firstName}"/>
    - When form displayed, calls getFirstName
    - When form submitted, passes value to setFirstName
- **Collections**
  - #{customer.addresses[0].zip}
    - Call getAddresses, index into array or list, call getZip
    - See also separate tutorial section on looping
- **Operators, conditional evaluation, args**
  - Use for display logic, not for things that could be in Java code

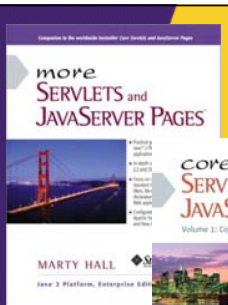
Questions?

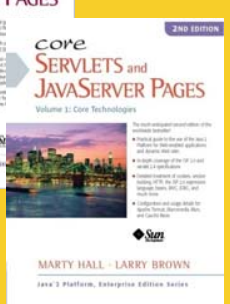
More info:
http://www.coreservlets.com/JSF-Tutorial/jsf2/ – JSF 2.2 tutorial
http://www.coreservlets.com/JSF-Tutorial/primefaces/ – PrimeFaces tutorial
http://courses.coreservlets.com/jsf-training.html – Customized JSF and PrimeFaces training courses
http://coreservlets.com/ – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training

h:dataTable – Building HTML Tables from Collections

JSF 2.2 Version

Originals of slides and source code for examples: http://www.coreservlets.com/JSF-Tutorial/jsf2/
Also see the PrimeFaces tutorial – http://www.coreservlets.com/JSF-Tutorial/primefaces/
and customized JSF2 and PrimeFaces training courses – http://courses.coreservlets.com/jsf-training.html





For live training on JSF 2, PrimeFaces, or other Java EE topics, email hall@coreservlets.com

Marty is also available for consulting and development support

Taught by the author of *Core Servlets and JSP*, this tutorial, and JSF 2.2 version of *Core JSF*. Available at public venues, or customized versions can be held on-site at your organization.

- Courses developed and taught by Marty Hall
  - JSF 2, PrimeFaces, Ajax, jQuery, Spring MVC, JSP, Android, general Java, Java 8 lambdas/streams, GWT, custom topic mix
  - Courses available in any location worldwide. Maryland/DC area companies can also choose afternoon/evening courses.
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Hadoop, Spring, Hibernate/JPA, RESTful Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- **Options for handling variable-length data**
  - Building HTML from a bean property
  - Using a builtin component like h:dataTable
  - Making your own composite component
  - Looping with ui:repeat
- **Using h:dataTable**
  - Basics: h:dataTable and h:Column
  - Headings
  - Style sheets
  - Ajax-enabled tables
  - Tables with conditional values

5

Overview

Motivation for h:dataTable

- **Goal**
  - You want results pages to be simple and HTML-oriented
    - Separation of concerns
    - Allows Web page developers to build GUI
- **Problem**
  - What if the action controller method produces data whose length can change? How do you generate output without resorting to JSP scripting and explicit Java looping?
- **Solutions**
  - There are a number of alternatives, *all* of which would work well in some circumstances. The issue is how much control the Web page author needs.
    - This section covers h:dataTable, but other alternatives are covered in other tutorial sections

7

JSF Constructs for Handling Variable-Length Data

Simplest for Page Author

Most Control for Page Author

- **Bean**
  - Have bean getter method spit out string or very simple HTML based on a collection
- **h:dataTable**
  - Use a builtin component that builds a table from a collection.
- **Your own composite component**
  - Make own component that builds some HTML construct (e.g., <ul> list) from a collection
- **ui:repeat**
  - Do explicit looping in results page

8

Which Option to Use

- **In general**
  - Use the simplest option that gives the Web page author enough control
- **h:dataTable**
  - The tutorial section on ui:repeat compares and contrasts the options with a brief example of each.
  - Bottom line of the comparison: h:dataTable is usually best when
    - You want to produce an HTML table from the collection
    - Each entry in the collection corresponds to a table row in a relatively consistent manner
- **Other options**
  - There are separate tutorial sections on ui:repeat and composite components

9

Example Notes

- **Data**
  - Normally, the data is produced in the business logic that is called by the action controller
    - E.g., you collect a bank customer ID and month in a form, and the button says
      <h:commandButton … action="#{user.findChanges}"/>
      where findChanges finds the bank account changes (deposits and withdrawals) in the month and puts then into an array or List
  - Here, we will hardcode the data for simplicity
    - I.e., make a managed bean with a method that returns a fixed collection

10

Basics

Simplest Syntax

- **Attributes**
  - var, value, border
- **Nested element**
  - h:column
- **Example**

```
<h:dataTable var="someVar" value="#{bean.someCollection}"
             border="…">
  <h:column>#{someVar.property1}</h:column>
  <h:column>#{someVar.property2} </h:column>
  …
</h:dataTable>
```

- **Legal types for "value" attribute**
  - Array, List, Collection, ResultSet, Result, DataModel
    - Never use ResultSet: breaks "business logic" separation discussed in managed beans section.
  - JSF 2.3 promises to also support Map and Iterable

12

Supporting Class: Programmer

```
public class Programmer {
  private String firstName, lastName, level;
  private double salary;
  private String[] languages;

  public Programmer(String firstName,
                    String lastName,
                    String level,
                    double salary,
                    String... languages) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.level = level;
    this.salary = salary;
    this.languages = languages;
  }

  // getFirstName, getLastName, getLevel, getSalary,
  // getLanguages
```

13

Programmer (Continued)

```
/** Formats the salary like "$24,567.89". */

 public String getFormattedSalary() {
   return(String.format("$%,.2f", salary));
 }

 /** Returns a String like "Java, C++, and Lisp".
  *  That is, it puts commas and the word "and" into list.
  */
 public String getLanguageList() {
   StringBuilder langList = new StringBuilder();
   for(int i=0; i<languages.length; i++) {
     if(i < (languages.length-1)) {
       langList.append(languages[i] + ", ");
     } else {
       langList.append("and " + languages[i]);
     }
   }
   return(langList.toString());
 }
```

14

Supporting Class: Company

```
public class Company {
  private String companyName;
  private Programmer[] programmers;

  public Company(String companyName,
                 Programmer... programmers) {
    this.companyName = companyName;
    this.programmers = programmers;
  }

  public String getCompanyName() {
    return (companyName);
  }

  public Programmer[] getProgrammers() {
    return (programmers);
  }
}
```

15

Managed Bean: Company1

This data never changes, so you might as well make it application scoped.

```
@ManagedBean
@ApplicationScoped
public class Company1 extends Company {
  public Company1() {
    super("My-Small-Company.com",
        new Programmer("Larry", "Ellison", "Junior", 34762.52,
                       "SQL", "Prolog", "OCL", "Datalog"),
        new Programmer("Larry", "Page", "Junior", 43941.86,
                       "Java", "C++", "Python", "Go"),
        new Programmer("Steve", "Ballmer", "Intermediate", 83678.29,
                       "Visual Basic", "VB.NET", "C#", "Visual C++",
                       "Assembler"),
        new Programmer("Sam", "Palmisano", "Intermediate", 96550.03,
                       "REXX", "CLIST", "Java", "PL/I", "COBOL"),
        new Programmer("Steve", "Jobs", "Intermediate", 103488.80,
                       "Objective-C", "AppleScript", "Java", "Perl",
                       "Tcl"));
  }
}
```

16

Facelets Page: Top

```
<!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:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head><title>h:dataTable Basics</title>
<link href="./css/styles.css"
      rel="stylesheet" type="text/css"/>
</h:head>
<h:body>
```

This first simple example doesn't use any f: tags, but most real h:dataTable examples use f:facet. So, plan ahead and add this namespace from the beginning.

You probably are already using this namespace for h:form, h:outputText, etc. But even if not, you need it for h:dataTable and h:column.

17

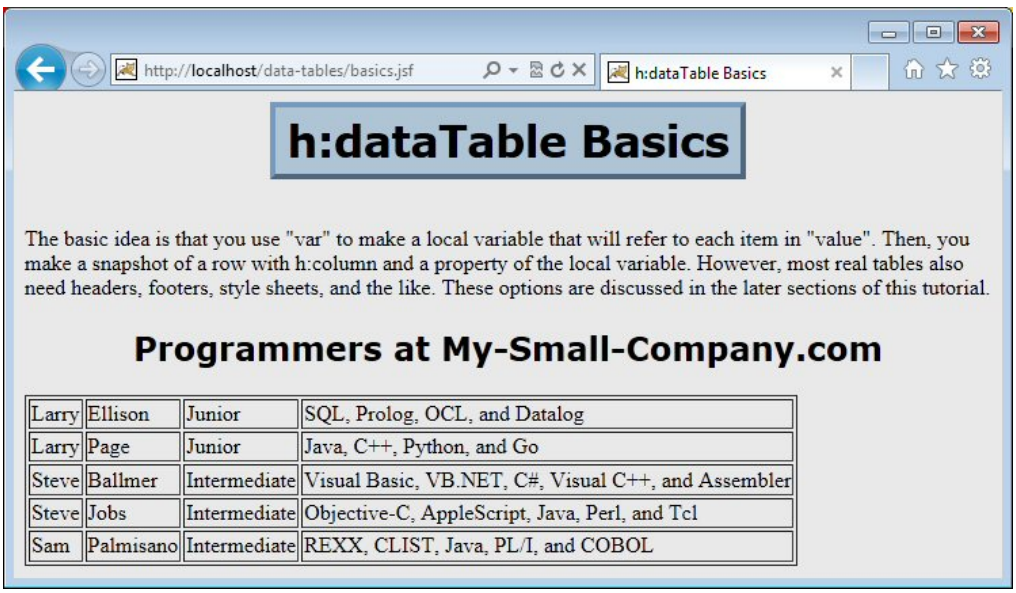
Facelets Page: Main Code

```
<h2>Programmers at #{company1.companyName}</h2>

<h:dataTable var="programmer"
             value="#{company1.programmers}"
             border="1">
  <h:column>#{programmer.firstName}</h:column>
  <h:column>#{programmer.lastName}</h:column>
  <h:column>#{programmer.level}</h:column>
  <h:column>#{programmer.languageList}</h:column>
</h:dataTable>
```

The border attribute of h:dataTable just becomes the border attribute of table.

18

Results



19

Headers and Captions

Overview

- **Problem**
  - Regular content gives a snapshot of a row. So, most content inside h:column is repeated for every row.
- **Solution**
  - Mark headings with f:facet. Shown for first row only.

```
<h:dataTable var="someVar" value="#{someCollection}">
  <h:column>
    <f:facet name="header">First Heading</f:facet>
    #{someVar.property1}
  </h:column>
  …
</h:dataTable>
```

This is a reserved name, not something arbitrary.
So, it will fail if you use "heading", "Header", or "headers".

- **Other f:facet options**
  - name="footer", name="caption" (outside h:caption)

21

Other Options for h:dataTable

- **Summary**

  <h:dataTable var="…" value="…" *otherAttributes*>…</h:dataTable>
- **Most important other attributes**
  - border, bgcolor, cellpadding, cellspacing, width
    onclick, ondoubleclick, onmouseover, etc.
    - Same as for the normal <table> tag.
  - styleClass, captionClass, headerClass, footerClass
    - CSS style names for main table, caption, header, and footer. Discussed in upcoming section.
  - rowClasses, columnClasses
    - List of CSS style names for each row and column. Will use each name in turn and then repeat. Discussed in upcoming section.
  - first, rows
    - First entry in collection to use, total number of rows to show.
  - id, rendered
    - Same as for all h:elements. Use id for Ajax. Use rendered if you will omit the table in certain situations (e.g., if no rows).

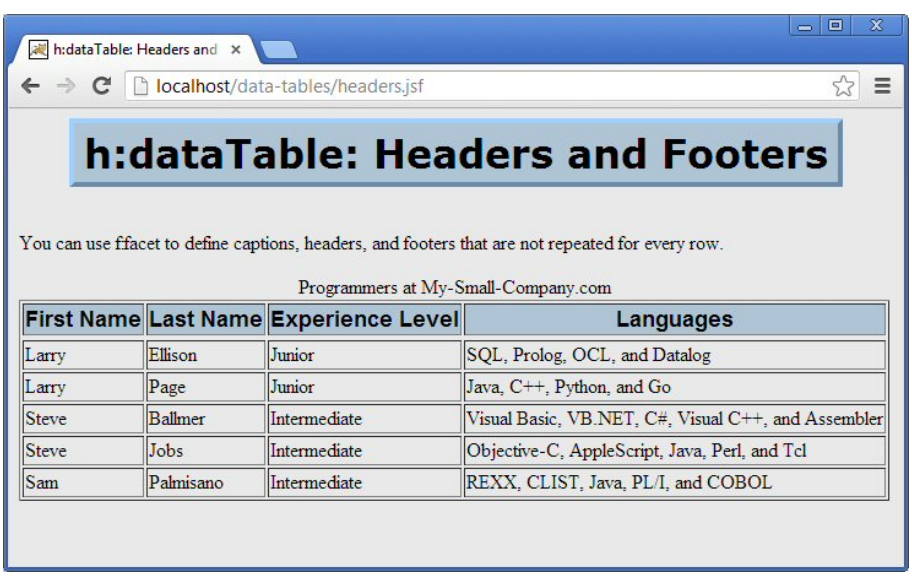
22

Facelets Page: Main Code

```
<h:dataTable var="programmer"
             value="#{company1.programmers}" border="1">
  <f:facet name="caption">Programmers at
                         #{company1.companyName}</f:facet>
  <h:column>
    <f:facet name="header">First Name</f:facet>
    #{programmer.firstName}
  </h:column>
  <h:column>
    <f:facet name="header">Last Name</f:facet>
    #{programmer.lastName}
  </h:column>
  <h:column>
    <f:facet name="header">Experience Level</f:facet>
    #{programmer.level}
  </h:column>
  <h:column>
    <f:facet name="header">Languages</f:facet>
    #{programmer.languageList}
  </h:column>
</h:dataTable>
```

Note: same managed bean (Company1) and supporting class (Programmer) as previous example.

Results



Style Sheets

Overview

- **Problem**
  - Using a general style sheet for the entire table does not always work
    - Might be multiple tables in the page that want different styles
    - Different rows might want different styles
- **Solution**
  - Explicitly supply CSS style names.
- **Summary**
  - styleClass, captionClass, headerClass, footerClass
    - CSS name that will apply to table as a whole, caption (if there is one), first row (if it is a header), and last row (if it is a footer)
  - rowClasses
    - Comma-separated list of names. Apply first name to first non-header row or column, then next name, etc. *When you get to end of list, repeat.* For instance, two names will apply to odd/even rows.
  - columnClasses
    - Comma-separated list of names. Apply until you run out, then stop. *Do not repeat as with rowClasses.* Thus, you normally supply exactly as many entries as you have columns.

Example: Summary

```
<h:dataTable var="someVar" value="#{someValue}"
        styleClass="tableStyle"
        captionClass="captionStyle"
        headerClass="headerStyle"
        footerClass="footerStyle"
        rowClasses="row1,row2"
        columnClasses="col1,col2,col3">
    ...
</h:dataTable>
```

Overall table style. I.e., <table class="tableStyle" ...>

Style for caption (if any).

Style for first table row, if it is a heading (i.e., there is at least one f:facet with name="header").

Style for last table row, if it is a footer.

Apply row1 to first, third, fifth… non-heading rows.
Apply row2 to second, fourth, sixth… non-heading rows.

Apply col1 to first column, col2 to second column, col3 to third column. If there are more than three columns, then no custom class is given to the later columns. Unlike rowClasses, these do not repeat.

Example: Facelets Part 1

```
<h:dataTable var="programmer"
             value="#{company1.programmers}"
             border="1"
             styleClass="mainTable"
             captionClass="caption1"
             headerClass="heading"
             rowClasses="even,odd">

   (Same body as last example)

</h:dataTable>
```

Fewer entries for rowClasses than rows in the table. Will repeat once the end of the list of styles is reached.

Despite the simplicity and popularity of rowClasses, sometimes the same effect can be accomplished by using a single style sheet for the table as a whole, and applying the positional CSS selectors nth-child(even), nth-child(odd), or nth-child(xn+y). These selectors are now supported consistently by all major browsers. See the CSS lecture earlier in this JSF tutorial series.

Example: Facelets Part 2

```
<h:dataTable var="programmer"
             value="#{myCompany.programmers}"
             border="1"
             styleClass="mainTable"
             captionClass="caption2"
             headerClass="heading"
             columnClasses="even,odd,even,odd">

   (Same body as last example)

</h:dataTable>
```

Same number of entries for columnClasses as there are columns in the table. Unlike with rowClasses, will *not* repeat once the end of the list of styles is reached.

29

Example: Style Sheet Part 1

```
.mainTable {
    margin-left: auto;
    margin-right: auto;
}
.caption1 {
   font-family: sans-serif;
   font-weight: bold;
   font-size: 24px;
   color: blue;
}
.caption2 {
   font-family: sans-serif;
   font-weight: bold;
   font-size: 24px;
   color: red;
}
.heading {
    font-family: sans-serif;
    font-weight: bold;
    font-size: 20px;
    color: black;
    background-color: silver;
    text-align: center;  }
```

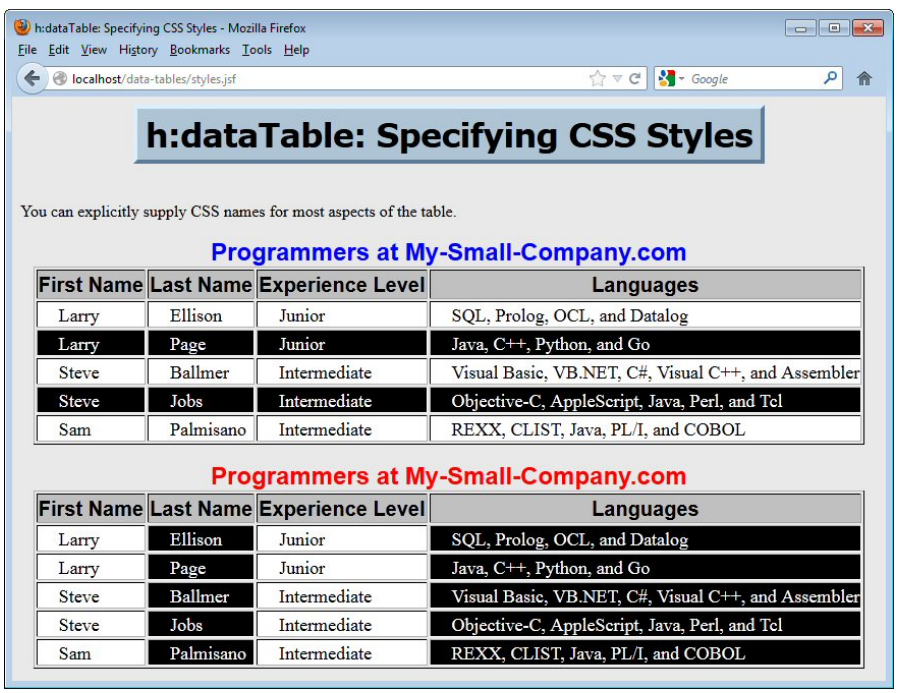
30

Example: Style Sheet Part 2

```
.even {
    font-family: Times New Roman, serif;
    font-size: 18px;
    color: black;
    background-color: white;
    text-indent: 20px;
}
.odd {
    font-family: Times New Roman, serif;
    font-size: 18px;
    color: white;
    background-color: black;
    text-indent: 20px;
}
```

31

Results



32

Ajax-Enabled Tables

Overview

- **Problem**
  - Wasteful to reload entire page when switching among various tables
    - Especially if there is a lot of non-table output on the page
- **Solution**
  - Use f:ajax to rebuild the table without reloading the entire page.
- **Notes**
  - Only basic code shown here.
  - Full details on f:ajax are given in separate tutorial section.

Example Overview

- **CompanyInfo class**
  - Has list of available companies
  - Maps company names to companies
- **Company class**
  - Stores list of programmers working for the company
    - Shown earlier in this tutorial
- **Programmer class**
  - Stores name, skill level, and programming languages
    - Shown earlier in this tutorial
- **Desired behavior**
  - User chooses a company name and sees table of programmers that work for that company.
  - Table should be updated without reloading entire page

35

CompanyInfo

```
@ManagedBean
@ApplicationScoped
public class CompanyInfo {
  private String companyName;
  private Company[] companies = {
    new Company1(), new Company2(), new Company3() };
  private List<String> companyChoices;
  private Map<String, Company> companyMap;

  public CompanyInfo() {
    companyChoices = new ArrayList<>();
    companyMap = new HashMap<>();
    companyName = companies[0].getCompanyName();
    for(Company c: companies) {
      companyChoices.add(c.getCompanyName());
      companyMap.put(c.getCompanyName(), c);
    }
  }
```

This information does not change based on user input, so it is application scoped.

Once the user chooses a company name, it will be associated with the corresponding Company.

36

CompanyInfo (Continued)

Simple accessor methods.

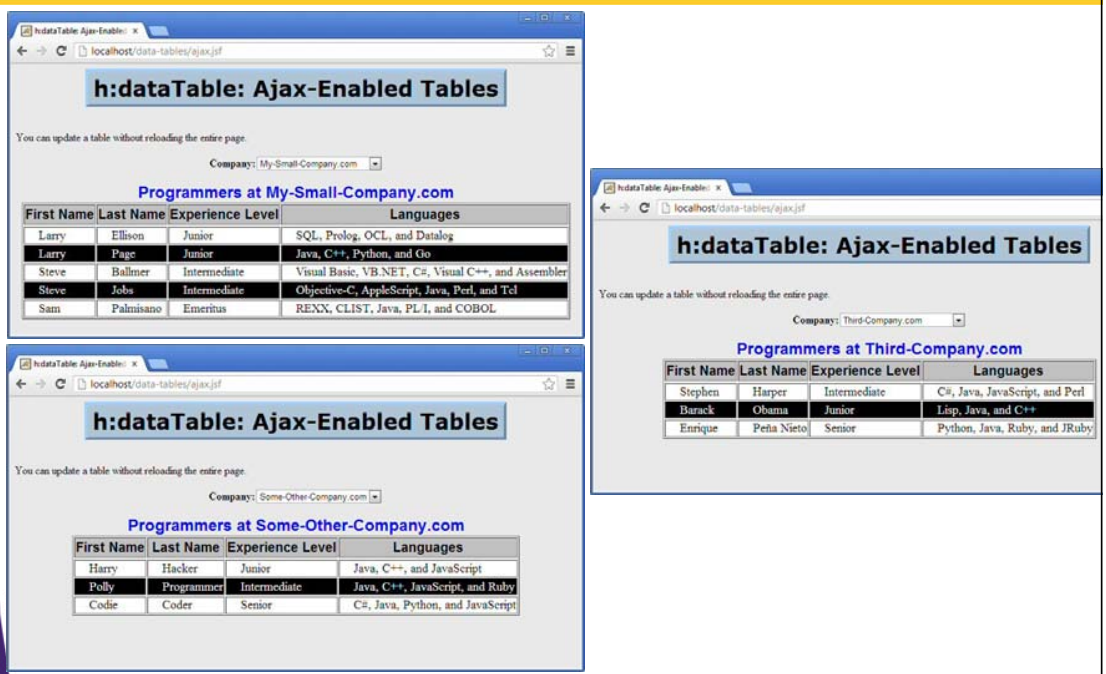
```
  public String getCompanyName() {
    return(companyName);
  }

  public void setCompanyName(String companyName) {
    this.companyName = companyName;
  }

  public List<SelectItem> getCompanyChoices() {
    return (companyChoices);
  }

  public Company getCompany() {
    return(companyMap.get(companyName));
  }
}
```


Facelets Code

```
<div align="center"><b>Company:</b>
<h:selectOneMenu value="#{companyInfo.companyName}">
  <f:selectItems value="#{companyInfo.companyChoices}"/>
  <f:ajax render="programmerTable"/>
</h:selectOneMenu>
</div>
<p/>
<h:dataTable var="programmer"
             value="#{companyInfo.company.programmers}"
             border="1"
             id="programmerTable"
             styleClass="mainTable"
             captionClass="caption1"
             headerClass="heading"
             rowClasses="even,odd">
   (Table body almost identical to previous two examples)
</h:dataTable>
```

When the combobox value changes, fire off a behind-the-scenes JavaScript request to the server. Update the model data for the combobox (i.e., call setCompanyName). Then, return the data needed to rebuild the element with the id "programmerTable". Replace value for that element only, without reloading entire page.

Results



39

Editable Table Cells

Overview

- **Problem**
  - You would like to let end user edit some table cells. However, you only want certain cells editable, and want a nearby "Update" button when it is.
- **Solution**
  - When user clicks "Edit" checkbox, change the output from simple text to a textfield followed by an update button. Use Ajax to prevent reloading entire page.
- **Main point of example**
  - h:dataTable cell values can use normal JSF constructs
    - Conditional text
    - h:inputText, h:commandButton and any other elements. Not limited to simple #{blah} output

41

Java Classes

- **CompanyInfo**
  - Shown earlier
- **Company**
  - Shown earlier
- **Programmer**
  - Partially shown earlier.
    - Added a boolean property to say whether the skill level is editable.
    - Added code to setLevel that resets the boolean property to false.

42

Programmer Class (New Parts)

```
public class Programmer {
  …
  private boolean isLevelEditable;
  …
  public void setLevel(String level) {
    isLevelEditable = false;
    this.level = level;
  }
  …
  public boolean isLevelEditable() {
    return(isLevelEditable);
  }

  public void setLevelEditable(boolean isLevelEditable) {
    this.isLevelEditable = isLevelEditable;
  }
}
```


Facelets Code

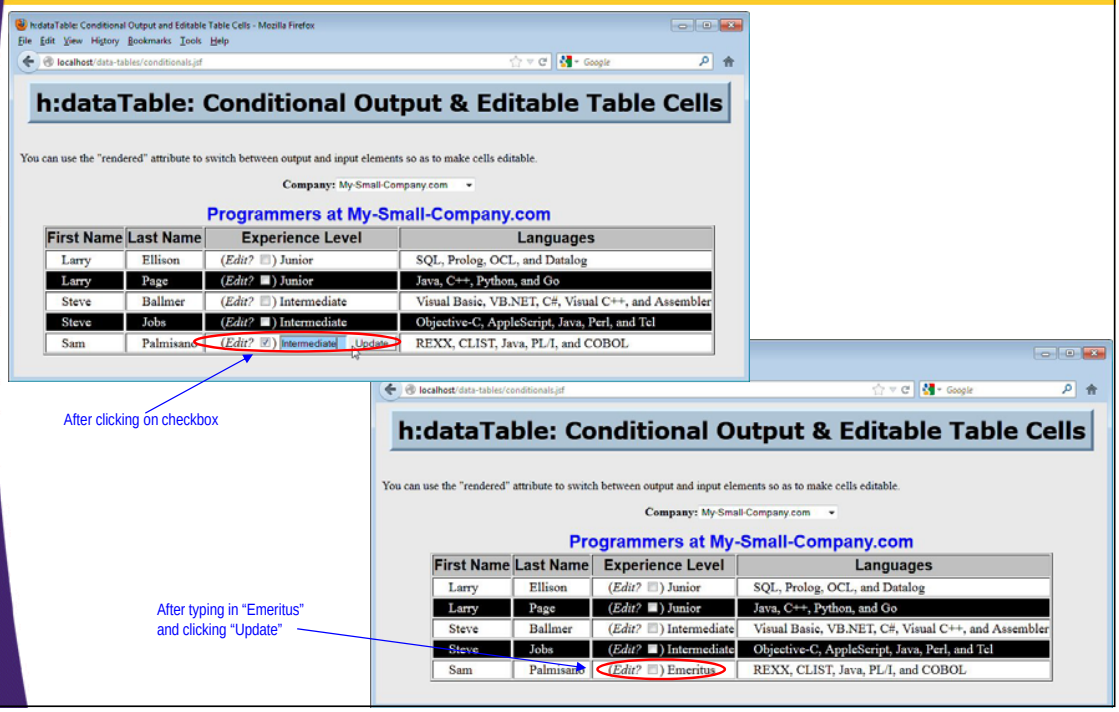
```
<h:selectOneMenu …>…</h:selectOneMenu>
…
<h:dataTable …>
  … (Most columns same as previous examples)
 <h:column>
    <f:facet name="header">Experience Level</f:facet>
    (<i>Edit?
    <h:selectBooleanCheckbox value="#{programmer.levelEditable}">
      <f:ajax render="@form"/>
    </h:selectBooleanCheckbox></i>)
    <h:inputText value="#{programmer.level}" size="12"
                 rendered="#{programmer.levelEditable}"/>
    <h:commandButton value="Update" rendered="#{programmer.levelEditable}">
      <f:ajax render="@form" execute="@form"/>
    </h:commandButton>
    <h:outputText value="#{programmer.level}"
                  rendered="#{!programmer.levelEditable}"/>
  </h:column>
 … (Most columns same as previous examples)
</h:dataTable>
```

Core h:selectOneMenu and h:dataTable code is about the same as the previous example.

If the checkbox is checked, these two elements (input field and Ajax-enabled submit button) are shown.

If the checkbox is not checked, this one element (giving simple output) is shown instead.

Results



Wrap-Up

Syntax Summary

```
<h:dataTable var="row"                Variable name to refer to each entry in collection
             value="#{someBean.someData}"   Collection (List, array, or a few other types)
             border="1"               It takes effort to use CSS to give table borders, so "border" is supplied as shortcut
             styleClass="css-table-style"   CSS style for main table (<table class="...">)
             headerClass="css-heading-style"   CSS for 1st row, if there is a header defined below
             rowClasses="evenRow,oddRow">   CSS styles for remaining rows, applied alternately
  <h:column>
    <f:facet name="header">Col 1 Title</f:facet>   Text for first col in first row
    #{row.someProperty}   Value for first col in all subsequent rows
  </h:column>
  <h:column>
    <f:facet name="header">Col 2 Title</f:facet>   Text for second col in first row
    #{row.someOtherProperty}   Value for second col in all subsequent rows.
                               You can also use other JSF constructs here.
  </h:column>
  …
</h:dataTable>
```


Questions?

More info:
http://www.coreservlets.com/JSF-Tutorial/jsf2/ – JSF 2.2 tutorial
http://www.coreservlets.com/JSF-Tutorial/primefaces/ – PrimeFaces tutorial
http://courses.coreservlets.com/jsf-training.html – Customized JSF and PrimeFaces training courses
http://coreservlets.com/ – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training