

<https://www.vogella.com/tutorials/JavaServerFaces/article.html>

JavaServer Faces with Eclipse. This article describes how to develop JavaServer Faces web applications with Eclipse WTP JSF tooling. It demonstrates managed beans, validators, external resource bundles and the JSF navigation concept. This tutorial was developed with Java 1.6, JavaServerFaces 1.2, the Apache MyFaces JSF implementation, Tomcat 6.0 and Eclipse 3.6.

1. JavaServer Faces - JSF

1.1. What is JSF

JavaServer Faces (JSF) is a UI component based Java Web application framework. JSF is serverbased, e.g. the JSF UI components and their state are represented on the server with a defined life cycle of the UI components. JSF is part of the Java EE standard.

A JSF application run in a standard web container, for example [Tomcat](#) or [Jetty](#).

This articles provides an introduction to JSF using only standard JSF features. For the usage of special Apache Trinidad features please see [Apache Myfaces Trinidad with Eclipse - Tutorial](#).

1.2. A JSF application

A JSF application consists of web pages with JSF UI components. A JSF application requires also some configuration files ("faces-config.xml" and `web.xml`).

The faces-config.xml defines:

- Managed Bean - the data elements of the JSF application (managed beans and backing beans) represent a Java class which will be created dynamically during runtime of the JSF application. It can be defined for which scope the bean is valid (Session, Request, Application or none)
- the navigation between web pages
- data validators - Used to check the validity of UI input
- data converters -Used to translate between UI and model

Managed beans are simple Java objects (POJO's) which are declared in "faces-config.xml" and can be used in an JSF application. For example you can define a Java object "Person". Once you define the object in faces-config.xml you can use the attributes of Person in your JSF UI components, e.g. by binding the value "firstName" of this object to an JSF input field.

JSF uses the Unified Expression Language (EL) to bind UI components to object attributes or methods.

1.3. Value and Method Binding

In JSF you can access the values of a managed bean via value binding. For value binding the universal Expression Language (EL) is used (to access bean and / or methods). In JSF you do not need to specify the get() or set() method but just the variable name. Method binding can be used to bind a JSF component, e.g. a button to an method of a Java class.

Expression Language statements either start with "\${" or with "#{", and end with "}". JSP EL expressions are using the \${...} syntax. These EL expressions are immediately evaluated. JSF EL expressions are of the type #{...}. These are only evaluated when needed (and otherwise stored as strings).

1.4. Prerequisites to use JSF

To use JSF you need:

- JSF Implementation (in the form of the JSF jars)
- The JSTL tags library
- A Java runtime environment
- A web-container to use JSF in (for example Tomcat)

1.5. JSF Main features

JSF has the following main features:

- JSF is based on the Model-View-Controller concept
- JSF has a stateful UI component model, e.g. each component is aware of its data
- JSF separates the functionality of a component from the display of the component. The renderer is responsible of displaying the component for a certain client. This renderer can get exchanged. The standard renderer for JSF components is the HTML renderer.
- JSF support listeners on UI components
- JSF support data validation, data binding and data conversion between the UI and the model

1.6. JSP and JSF

In this tutorial the JSF application will be build based on JavaServer Pages (JSP's). JSTL tags are used to include JSF UI components into the JSP. This is standard in JSF 1.2. The JSF 2.0 version is using Facelets.

2. JSF configuration files

2.1. Overview

JSF is based on the following configuration files:

- web.xml - General web application configuration file
- faces-config.xml - Contains the configuration of the JSF application.

2.2. web.xml

JSF requires the central configuration list `web.xml` in the directory WEB-INF of the application. This is similar to other web-applications which are based on servlets.

You must specify in `web.xml` that a "FacesServlet" is responsible for handling JSF applications. "FacesServlet" is the central controller for the JSF application.

"FacesServlet" receives all requests for the JSF application and initializes the JSF components before the JSP is displayed.

2.3. faces-config.xml

"faces-config.xml" allows to configure the application, managed beans, convertors, validators, and navigation.

3. Installation

3.1. Eclipse

For JSP development you need the Eclipse WTP and an installed Tomcat. See [Installation of Eclipse WTP and Tomcat](#).

3.2. JSF library

A JSF library is required. We will later use Eclipse to download and install the Apache MyFaces JSF implementation during project creation.

3.3. JSLT library

Download the JSLT library from <https://jstl.java.net/download.html>.

4. Your first JSF project

Our first JSF example will be a celsius to fahrenheit convertor.

4.1. Create JSF Project

Create a new Dynamic Web Project "de.vogella.jsf.first". Under "Configuration" select "JavaServer Faces v1.2".

New Dynamic Web Project

Dynamic Web Project
Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name:

Project contents
☒ Use default
Directory:

Target runtime

Dynamic web module version

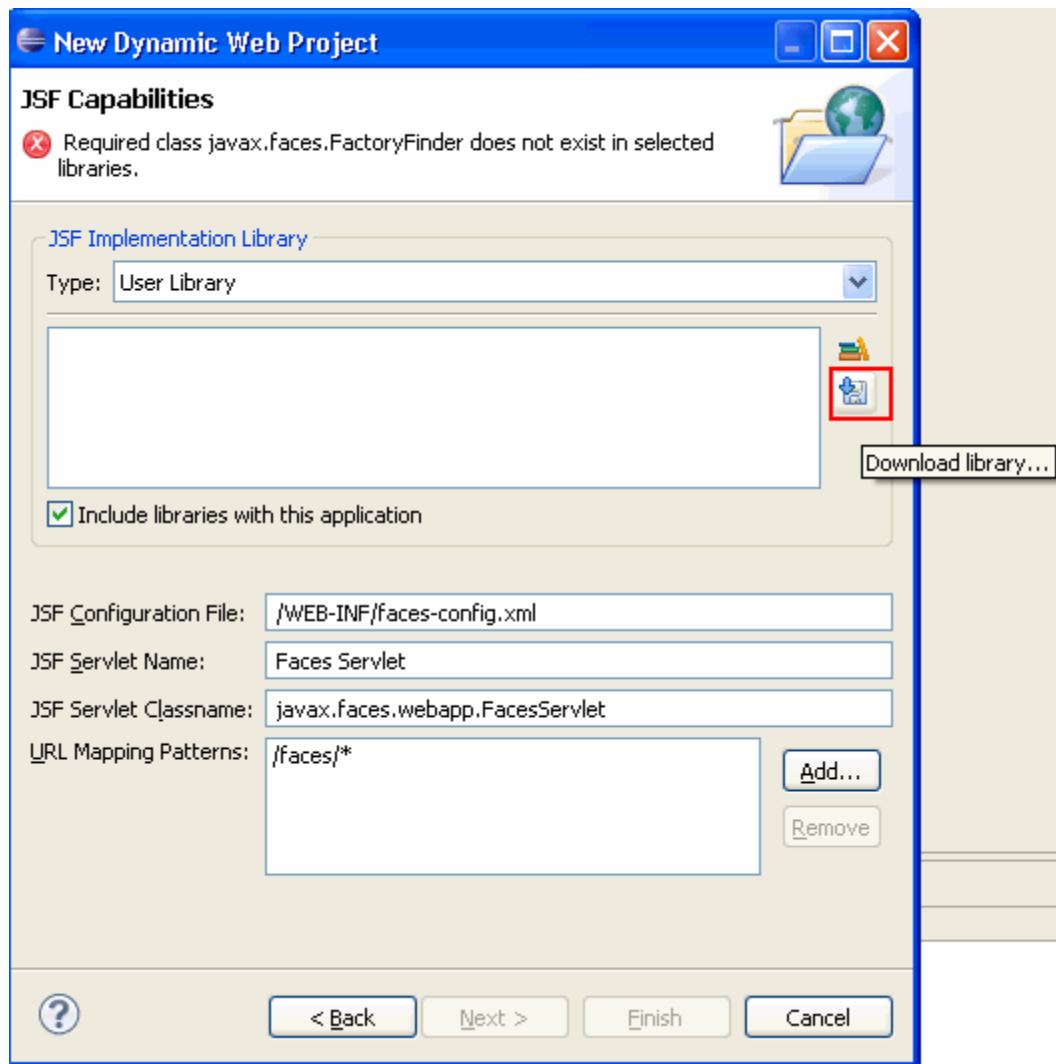
Configuration

Configures a Dynamic Web application to use JSF v1.2

EAR membership
☐ Add project to an EAR
EAR project name:

Working sets
☐ Add project to working sets
Working sets:

Press next until you see the following screen.




The first time you create a JSF project you need to install / download a JSF implementation. Press the `Download library...` button and select the Apache Library and install it.

Download Library

Download Library

Select a library to download from the specified provider.



Library ▲	Download Provider
JSF 1.2 (Apache MyFaces)	Apache Software Foundation
JSF 1.2 (Sun RI)	Sun Microsystems


Library name:

JSF 1.2 (Apache MyFaces)

Download destination:

C:\Documents and Settings\d034797\Desktop\Documents\16_EclipseProjects\20_Webpag

Browse...

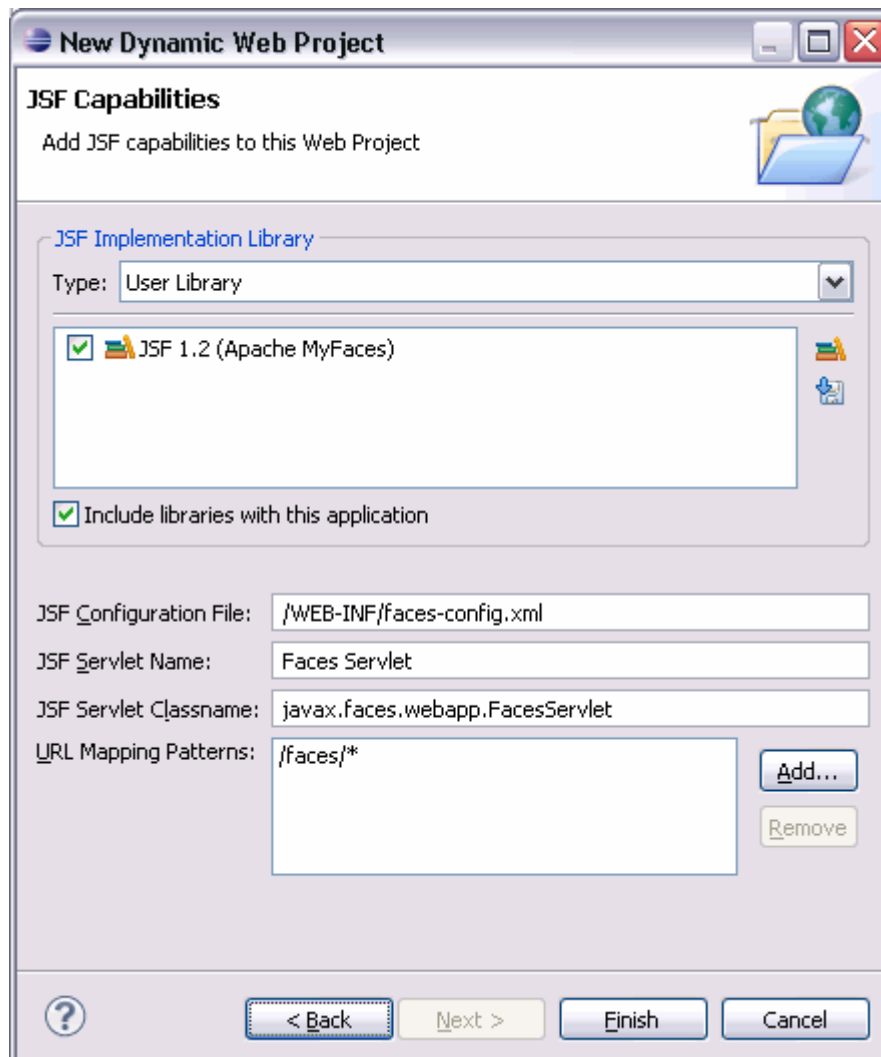


< Back

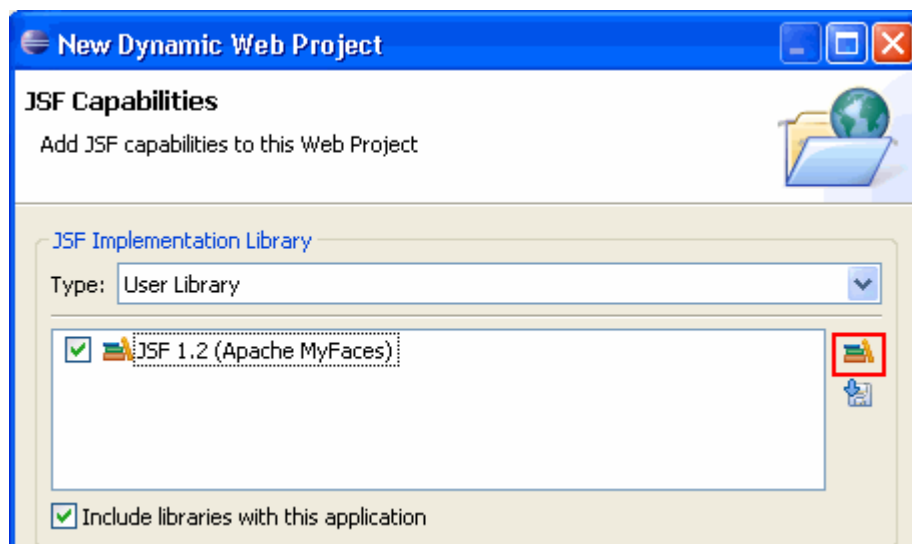
Next >

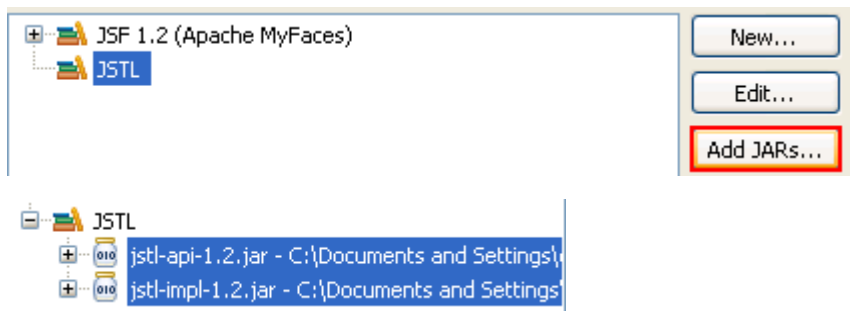
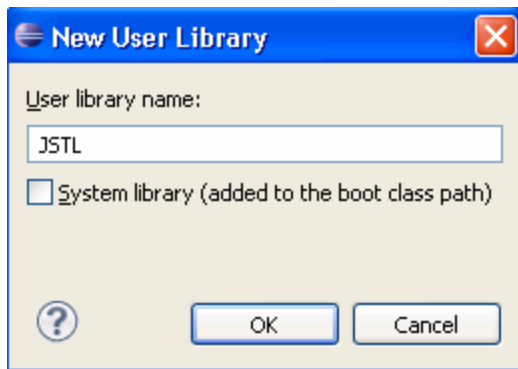
Finish

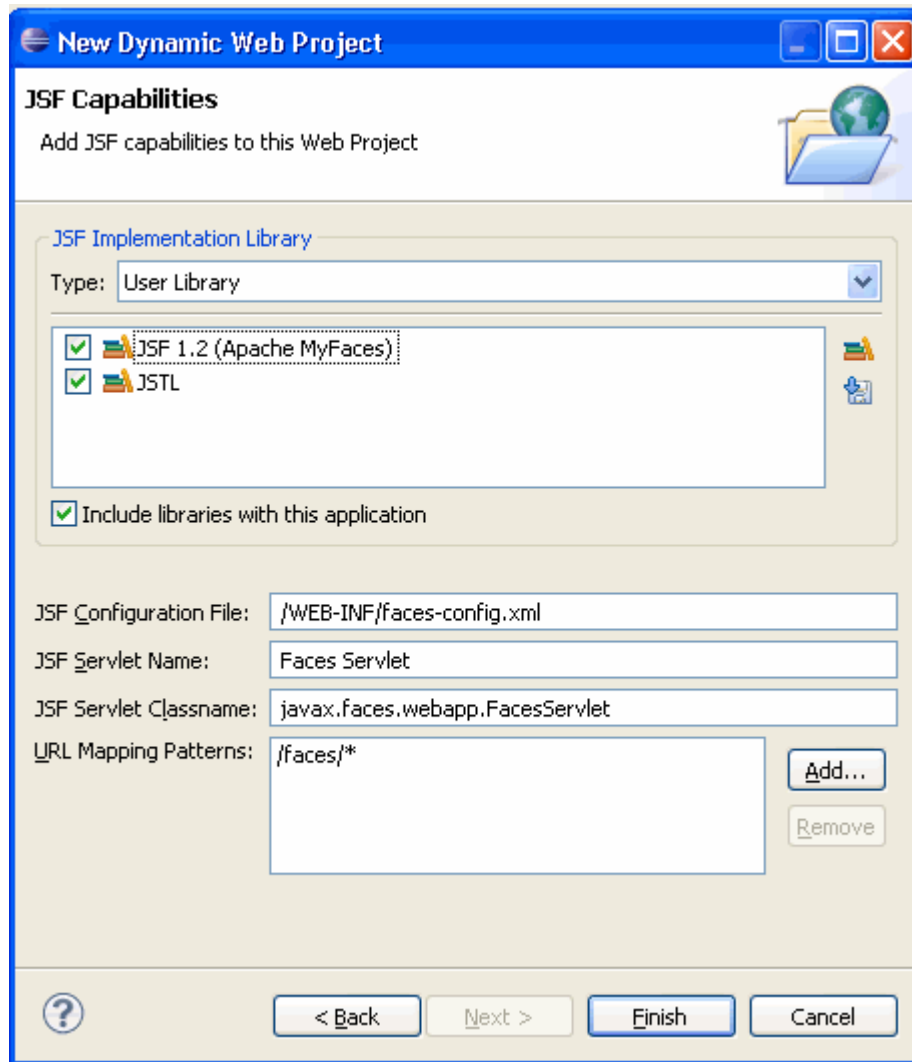
Cancel



Press Manage libraries and create a library for JSTL.







Click **Finish**. Your project has been created.

4.2. Review the generated project

Review the `web.xml` file. It has an entry for the Faces Servlet and for the servlet mapping. Also the file "faces-config.xml" has been created.

To add the JSF settings to an existing dynamic web project, right-click on your project, select **Project Properties** **Project Facets** and add then JSF facet to your project.

4.3. Domain Model

Create a package "de.vogella.jsf.first.model" and the class "TemperatureConvertor".

```

package de.vogella.jsf.first.model;

public class TemperatureConvertor {
    private double celsius;
    private double fahrenheit;
    private boolean initial= true;

    public double getCelsius() {
        return celsius;
    }
    public void setCelsius(double celsius) {
        this.celsius = celsius;
    }
    public double getFahrenheit() {
        return fahrenheit;
    }

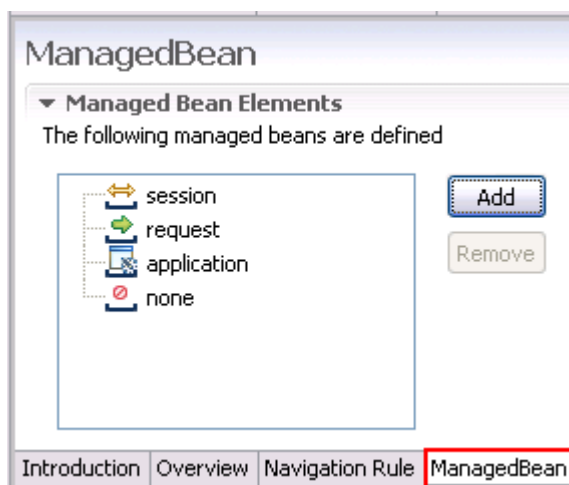
    public boolean getInitial(){
        return initial;
    }

    public String reset (){
        initial = true;
        fahrenheit =0;
        celsius = 0;
        return "reset";
    }
    public String celsiusToFahrenheit(){
        initial = false;
        fahrenheit = (celsius *9 / 5) +32;
        return "calculated";
    }
}

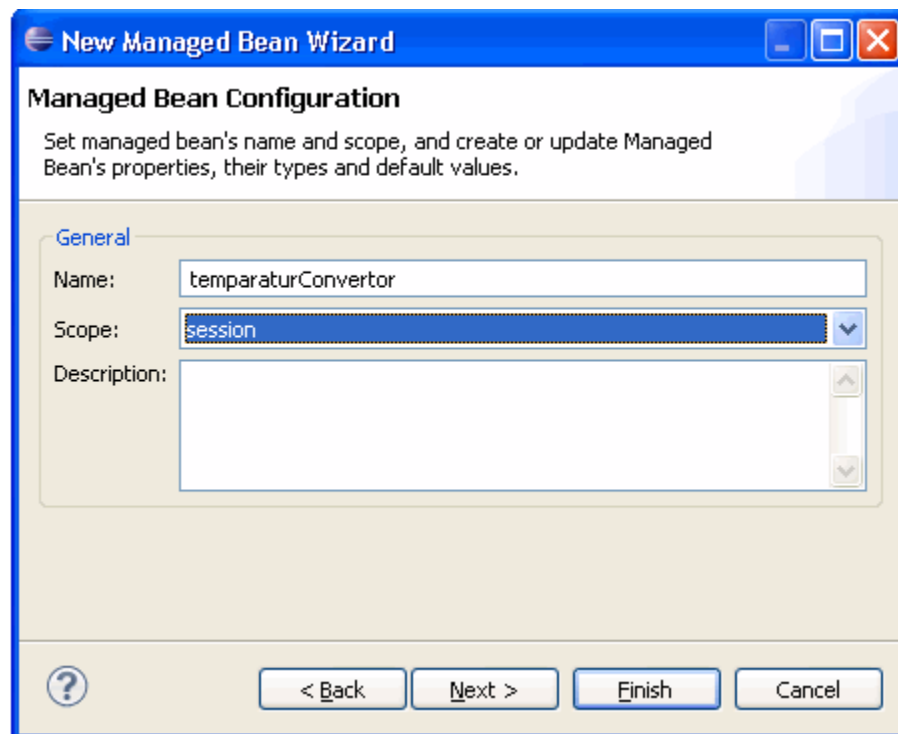
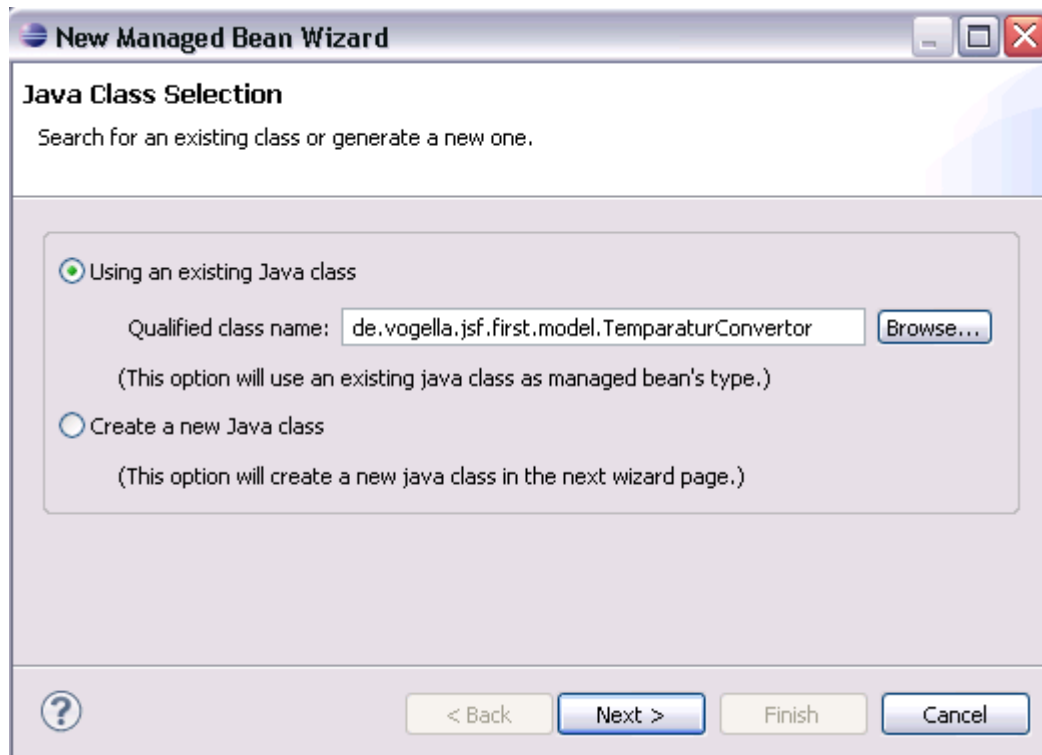
```

4.4. Define managed bean

Double-click on faces-config.xml in the WEB-INF directory and select the tab "ManagedBeans".

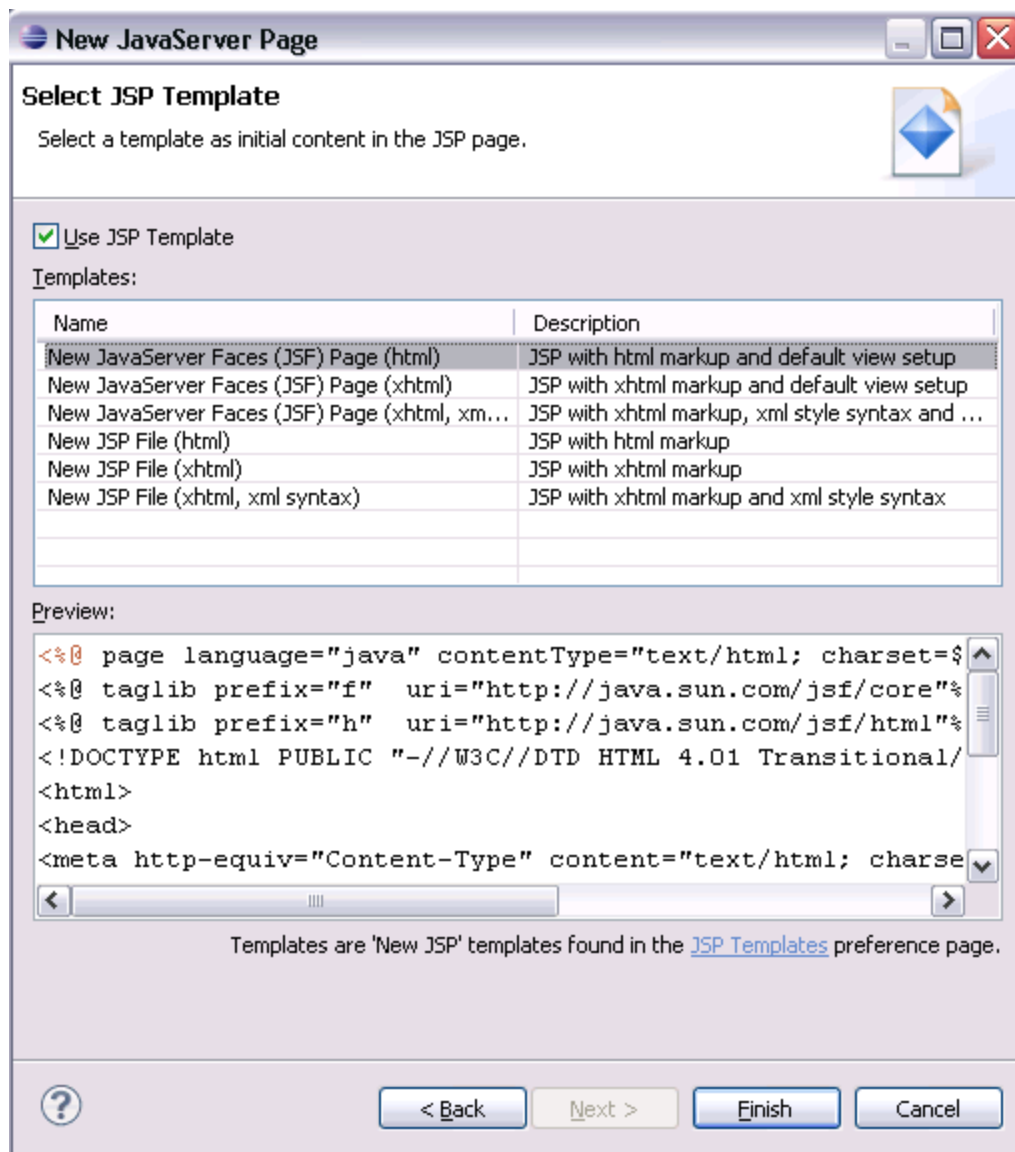


Press add and maintain your class.



The result should look like the following:

Select your project, right-click on it, select New → JSP. Create the JSP page "Convertor.jsp". Use the "New JavaServer Faces (JSF) Page (html)" template.



Change the code to the following.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Celsius to Fahrenheit Convertor</title>
</head>
<body>
<f:view>
    <h:form>
        <h:panelGrid columns="2">
```

```

        <h:outputLabel value="Celsius"></h:outputLabel>
        <h:inputText
value="#{temperatureConvertor.celsius}"></h:inputText>
        </h:panelGrid>
        <h:commandButton action="#{temperatureConvertor.celsiusToFahrenheit}"
value="Calculate"></h:commandButton>
        <h:commandButton action="#{temperatureConvertor.reset}"
value="Reset"></h:commandButton>
        <h:messages layout="table"></h:messages>
    </h:form>

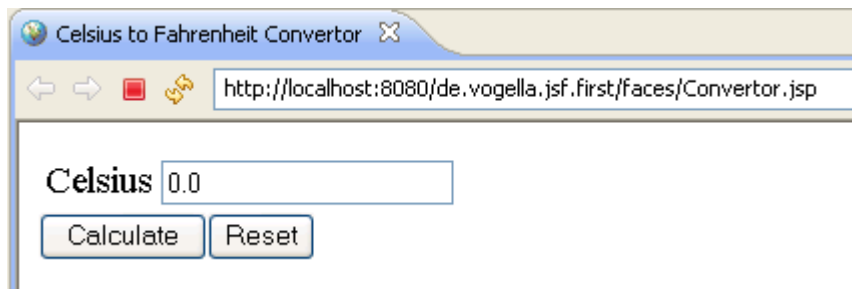
    <h:panelGroup rendered="#{temperatureConvertor.initial!=true}">
    <h3> Result </h3>
    <h:outputLabel value="Fahrenheit "></h:outputLabel>
    <h:outputLabel
value="#{temperatureConvertor.fahrenheit}"></h:outputLabel>
    </h:panelGroup>
</f:view>
</body>
</html>

```

All JSF tag must be always be enclosed in a <f:view> tag.

4.6. Run your webapplication

Select Convertor.jsp, right mouse-click- >run as → run on server.



Congratulations. You should be able to use your JSF application.

4.7. Layout via css

JFP applications can get styled via css files. To load a style sheet include the following in your JSP page in the header section. This is related to the mystyle.css file under your folder WebContent/css..

```

<LINK href="<%=request.getContextPath()%>/css/mystyle.css" rel="stylesheet"
type="text/css">

```

5. Your second JSF application

This second JSF application will add validation, resource bundles and navigation as additional functionality.

5.1. Create JSF Project

Create a new Dynamic Web Project "de.vogella.jsf.starter".

5.2. Domain model

Create a new package de.vogella.jsf.starter.model and the following class.

```
package de.vogella.jsf.starter.model;

public class User {
    private String name;
    private String password;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }

    public String login(){
        // Image here a database access to validate the users
        if (name.equalsIgnoreCase("tester") &&
password.equalsIgnoreCase("tester")){
            return "success";
        } else {
            return "failed";
        }
    }
}
```

Please note that we are hard-coding that only user tester with password tester can login.

Create the following class.

```
package de.vogella.jsf.starter.model;

import java.util.Random;
```



```

public class Card {
    private int left;
    private int right;
    private int result = 0;

    public Card() {
        Random random = new Random();
        int i = 0;
        int j = 0;
        do {
            i = random.nextInt(10);
        } while (i <= 4);

        do {
            j = random.nextInt(100);
        } while (j <= 20);

        left = i;
        right = j;
    }

    public int getLeft() {
        return left;
    }

    public void setLeft(int left) {
        this.left = left;
    }

    public int getRight() {
        return right;
    }

    public void setRight(int right) {
        this.right = right;
    }

    // Controller

    public String show() {
        result = left * right;
        return "success";
    }

    public String clear() {
        result = 0;
        return "clear";
    }

    public int getResult() {
        return result;
    }

    public void setResult(int result) {
        this.result = result;
    }
}

```

```
}
```

The class Card contains currently some controller code. The next chapter will demonstrate how to keep your model code clean and how to use controllers directly.

5.3. Register your managed beans

Double-click on faces-config.xml and select the tab "ManagedBeans". Register your User.java and Card.java as managed beans.

5.4. Validators

JSP allows to define validators which allows to check certain values which are placed in the UI. Create therefore the following class.

```
package de.vogella.jsf.starter.validator;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class LoginValidator implements Validator {
    public void validate(FacesContext context, UIComponent component,
        Object value) throws ValidatorException {
        String user = (String) value;
        if (!user.equalsIgnoreCase("tester")) {
            FacesMessage message = new FacesMessage();
            message.setDetail("User " + user + " does not exists");
            message.setSummary("Login Incorrect");
            message.setSeverity(FacesMessage.SEVERITY_ERROR);
            throw new ValidatorException(message);
        }
    }
}
```

Select your faces-config.xml and select the tab Component. Select Validators and press Add.

Component

▸ Components

▸ Converters

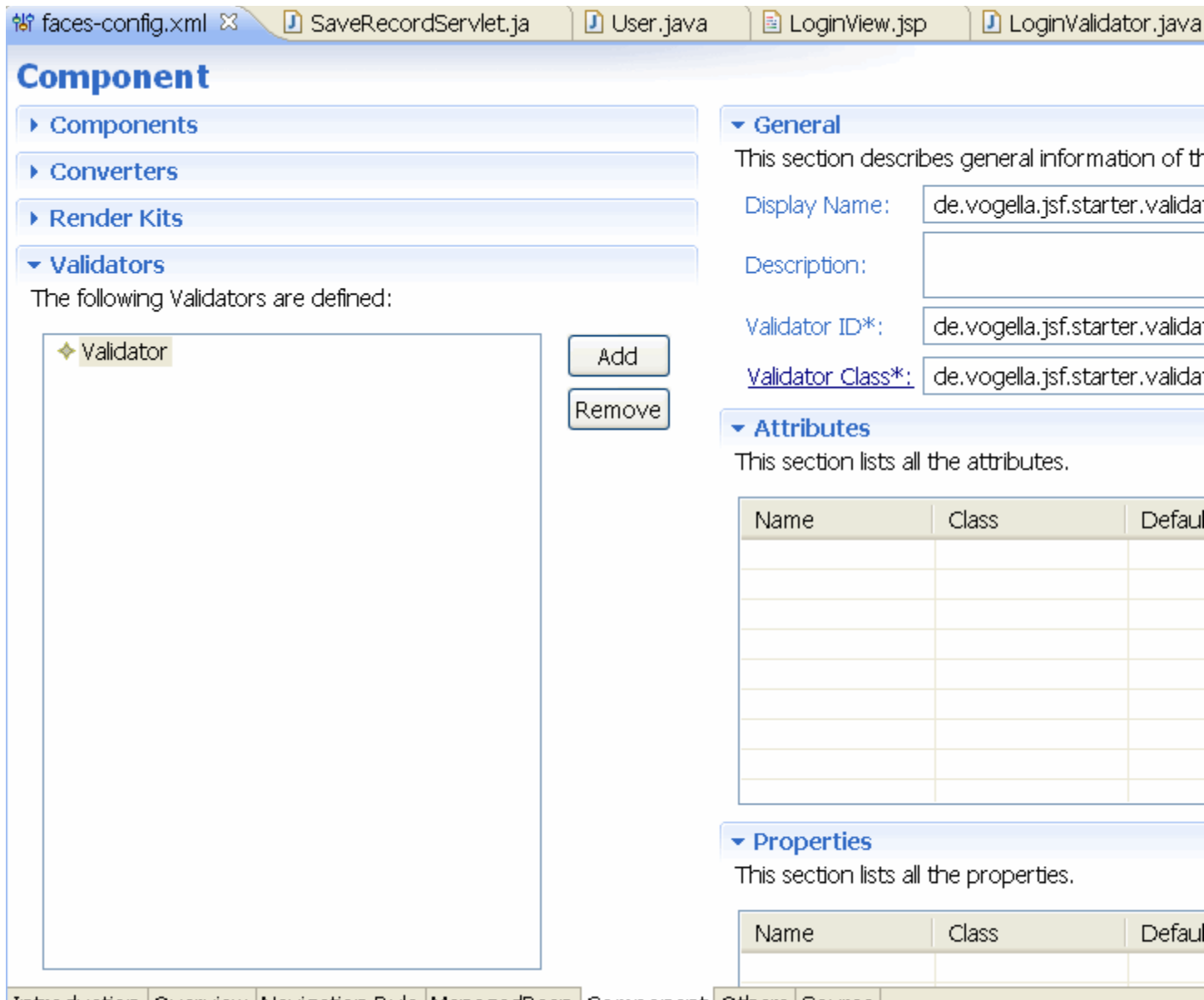
▸ Render Kits

▼ Validators

The following Validators are defined:

Add

Remove



5.5. Resource bundle for messages

With JSP it is easy to use resource bundles for the static text in your JSP application. Create the following file "messages.properties" in your source folder under the package "de.vogella.jsf.starter".

```
user=User
password=Password
login=Login
hello=Moin
left=Left Side
right=Right Side
result= Result
show= Show Result
```

next= New Test
reset= Reset

5.6. JavaServer Page with JSF components

Create a new JSP page "LoginView.jsp" and change the code to the following:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Login</title>
</head>
<body>
<f:view>
    <f:loadBundle basename="de.vogella.jsf.starter.messages" var="msg" />
    <h:form>
        <h:panelGrid columns="2">
            <h:outputLabel value="#{msg.user}"></h:outputLabel>
            <h:inputText value="#{user.name}">
                <f:validator
validatorId="de.vogella.jsf.starter.validator.LoginValidator" />
                </h:inputText>
            <h:outputLabel value="#{msg.password}"></h:outputLabel>
            <h:inputSecret value="#{user.password}">
                </h:inputSecret>
            </h:panelGrid>
            <h:commandButton action="#{user.login}"
value="#{msg.login}"></h:commandButton>
            <h:messages layout="table"></h:messages>
        </h:form>
    </f:view>
</body>
</html>
```

Lets explain a few fields.

Table 1. Fields

Element	Description
<pre><%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%> <%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%></pre>	Makes the core and html tags available in the page

Table 1. Fields

Element	Description
<code><f:view></code>	Indicates that the following will use JSF components.
<code><f:loadBundle basename="de.vogella.jsf.starter.messages" var="msg"/></code>	load the resource / message bundle which is then available in the application under the name msg
<code><h:form></code>	Starts a form
<code><h:outputLabel value="#{msg.user}"></h:outputLabel></code>	Defines a label which used the text user define in the resource bundle
<code><h:inputText tabindex="1" value="#{user.name}"></h:inputText></code>	Define a input field which used the managed bean user and maps to field name
<code><h:inputSecret tabindex="2" value="#{user.password}"></code>	Masked input files, mapped to the managed bean user and field password
<code><h:commandButton action="{user.login}" value="{msg.login}"></h:commandButton></code>	The button is mapped to the method user.login

Create another JSP "Trainer.jsp" with the following code.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<f:view>
    <f:loadBundle basename="de.vogella.jsf.starter.messages" var="msg" />
    <h:form>
        <h:panelGrid columns="3">
            <h:outputLabel value="#{msg.left}"></h:outputLabel>
```

```

        <h:inputText id="left" value="#{card.left}"></h:inputText>
        <h:message for="left"></h:message>

        <h:outputLabel value="#{msg.right}"></h:outputLabel>
        <h:inputText id="right" value="#{card.right}">
        </h:inputText>
        <h:message for="right"></h:message>

    </h:panelGrid>
    <h:commandButton action="#{card.show}"
value="#{msg.show}"></h:commandButton>
    <h:commandButton action="#{card.clear}" value="#{msg.reset}"
        immediate="true"></h:commandButton>
    <h:messages layout="table"></h:messages>
</h:form>

<h:panelGrid rendered="#{card.result!=0}" columns="3">
    <h:outputLabel value="#{msg.result}"></h:outputLabel>
    <h:inputText id="result" value="#{card.result}">
    </h:inputText>
    <h:message for="result"></h:message>
</h:panelGrid>

</f:view>
</body>
</html>

```

Create another JSP FailedLogin.jsp with the following code.

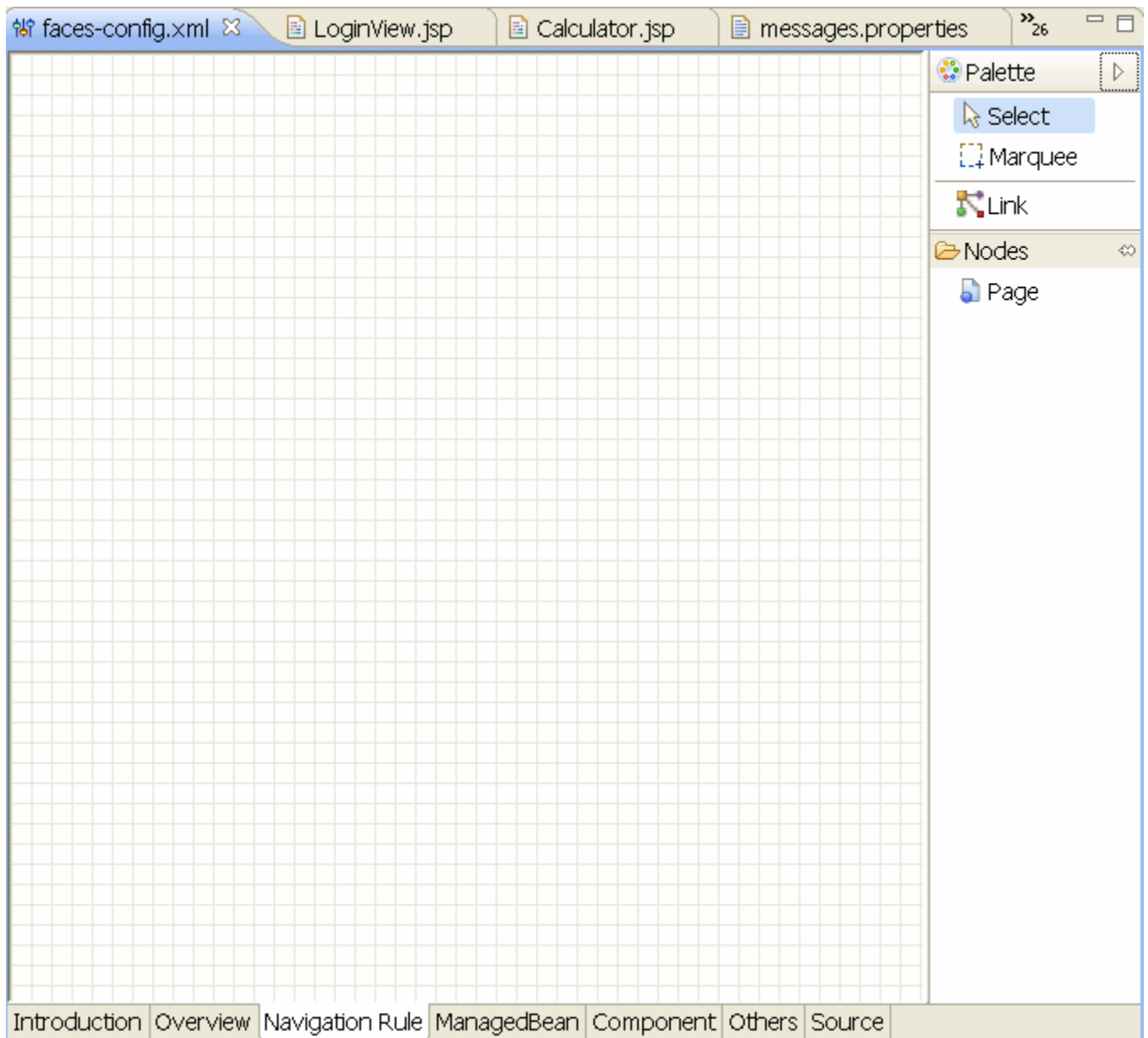
```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<f:view>
    <h1>Failed Login.</h1>
</f:view>
</body>
</html>

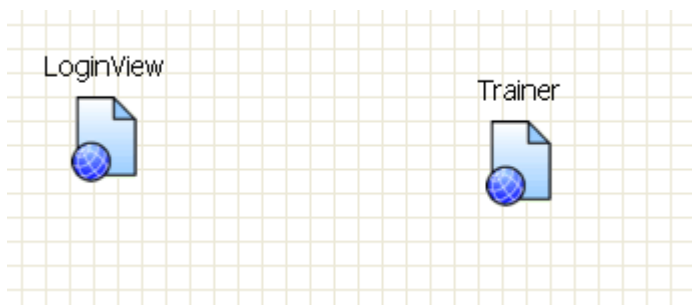
```

5.7. Navigation Rule

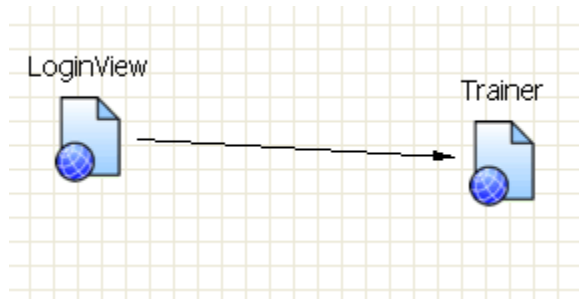
Select your faces-config.xml and select the tab "Navigation Rule". Make the palette available if necessary.



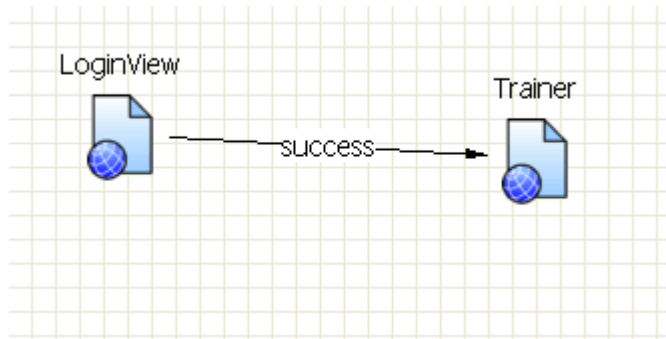
Select Page and click in the workarea. Add LoginView and Trainer to the workspace.



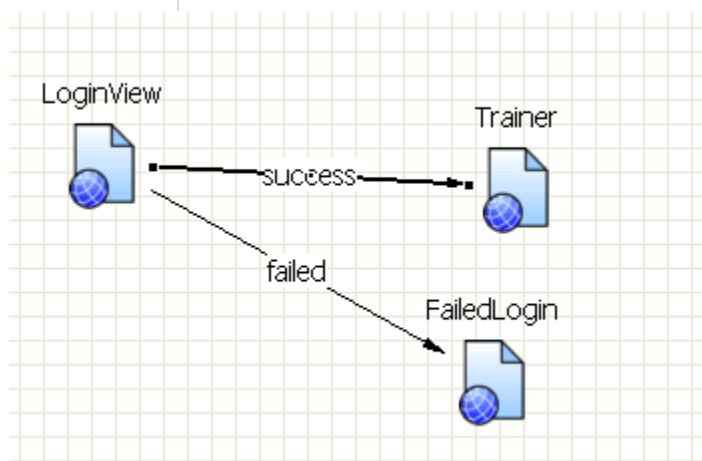
Click on Link, then on LoginView and then on Trainer. You should have now an arrow which indicates a navigation rule.



Click in the Palette on Select. Select then the arrow and the properties view. Input "success" in the From – Outcome



The user bean return the String success. In the navigation rule you now defined that if we receive "success" then we should be going to the next page.



Add a navigation rule so that in the case the user does not use the right user / password you send him to the failure page.

5.8. Run your webapplication

To run your webapplication, select LoginView.jsp, right mouse-click- >run as → run on server.

Remember that we are hard-coding that only user "tester" with password "tester" can login. Try another user this should not work.

You should be able to login with the right user and move to the next page.

Left Side

Right Side

Result

6. JSF application with a controller

We will now extend the example from the previous chapter to a math trainer. The system will propose two number and the user must multiply both values and input the result. This JSF application will use a controller which handles the JSF logic. This will allow you to create a domain model without application logic.

In general it is considered as a good design practice to keep the model independently from the application logic.

This example will also demonstrate the usage of dependency injection in JSF.

6.1. Create JSF Project

Create a new Dynamic Web Project "de.vogella.jsf.card".

6.2. Domain model

Create a new package `de.vogella.jsf.card.model` and the following class.

```
package de.vogella.jsf.card.model;

import java.util.Random;

public class Card {
    private int left;
    private int right;

    public Card() {
        Random random = new Random();
        int i = 0;
        int j = 0;
        do {
            i = random.nextInt(10);
        } while (i <= 4);

        do {
            j = random.nextInt(100);
        } while (j <= 20);

        left = i;
        right = j;
    }

    public int getLeft() {
        return left;
    }
    public void setLeft(int left) {
        this.left = left;
    }
    public int getRight() {
        return right;
    }
    public void setRight(int right) {
        this.right = right;
    }
}
```

6.3. Controller

Create the following class `CardController`.

```
package de.vogella.jsf.card.controller;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIPanel;
import javax.faces.context.FacesContext;

import de.vogella.jsf.card.model.Card;

public class CardController {
```

```

private Card card;
private UIPanel resultPanel;
private int result;

public CardController() {
}

public String checkResult() {
    FacesContext context = FacesContext.getCurrentInstance();
    resultPanel.setRendered(true);

    if (checkOperation()) {
        context.addMessage(null, new FacesMessage(
            FacesMessage.SEVERITY_INFO, "Correct", null));
    } else {
        context.addMessage(null, new FacesMessage(
            FacesMessage.SEVERITY_INFO, "Incorrect", null));
    }
    return null;
}

private boolean checkOperation() {
    return (card.getLeft() * card.getRight() == result);
}

public UIPanel getResultPanel() {
    return resultPanel;
}

public void setResultPanel(UIPanel resultPanel) {
    this.resultPanel = resultPanel;
}

public int getResult() {
    return result;
}

public void setResult(int result) {
    this.result = result;
}

public String next() {
    FacesContext context = FacesContext.getCurrentInstance();
    if (checkOperation()){
        resultPanel.setRendered(false);
        card = new Card();
        return null;
    } else {
        context.addMessage(null, new FacesMessage(
            FacesMessage.SEVERITY_INFO, "Incorrect", null));
    }
    return null;
}

public Card getCard() {
    return card;
}

```

```

    }

    public void setCard(Card card) {
        this.card = card;
    }
}

```

This class has a field resultPanel. This field will later get connected to a UIComponent (panel) from the JSP.

6.4. Register your managed beans- Dependency injection

Double-click on faces-config.xml and select the tab "ManagedBeans". Register the classes "CardController" and "Card" as managed beans. The scope of card will be set to none as it will be inserted into the ControllerCard via dependency injection. In the initialization tab maintain the data as displayed in the screenshot. The value #{card} refers to the managed bean "card".

ManagedBean

▼ **Managed Bean Elements**
The following managed beans are defined

- session
 - cardController
- request
- application
- none
 - card

Buttons: Add, Remove

▼ **Managed Bean**
This section describes general configuration of

Managed Bean name*: cardController

Managed Bean class*: de.vogella.jsf.card...

Managed Bean scope*: session

▼ **Initialization**
You can initialize the managed bean's properties with

Managed Bean class type: ☒ General class ☐

Name	Class	Value
card	de.vogella.j...	#{card}

The generated XML code should look like the following (you see this if you select the tab "Source").

```

<managed-bean>
    <managed-bean-name>cardController</managed-bean-name>

```

```

        <managed-bean-
class>de.vogella.jsf.card.controller.CardController</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
            <property-name>card</property-name>
            <property-class>de.vogella.jsf.card.model.Card</property-class>
            <value>#{card}</value>
        </managed-property>
    </managed-bean>
    <managed-bean>
        <managed-bean-name>card</managed-bean-name>
        <managed-bean-class>de.vogella.jsf.card.model.Card</managed-bean-
class>
        <managed-bean-scope>none</managed-bean-scope>
    </managed-bean>

```

6.5. Resource bundle for messages

Create the following file "messages.properties" in your source folder under the package "de.vogella.jsf.card".

```

left=Left Side
right=Right Side
result=Result
show= Check
next= Next

```

6.6. JavaServer Page with JSF components

Create a new JSP page "Trainer.jsp" and change the code to the following:

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<h1> Train your Brain</h1>

<h3>Please calculate the result </h3>
<f:view>
    <f:loadBundle basename="de.vogella.jsf.card.messages" var="msg" />
    <h:form>
        <h:panelGrid columns="3">
            <h:panelGrid columns="2">
                <h:outputLabel value="#{msg.left}"></h:outputLabel>
                <h:outputLabel id="left"
value="#{cardController.card.left}"></h:outputLabel>

```

```

        <h:outputLabel value="#{msg.right}"></h:outputLabel>
        <h:outputLabel id="right"
value="#{cardController.card.right}">
        </h:outputLabel>

        <h:outputLabel value="#{msg.result}"></h:outputLabel>
        <h:inputText id="result"
value="#{cardController.result}"></h:inputText>
        </h:panelGrid>

    </h:panelGrid>
    <h:commandButton action="#{cardController.checkResult}"
        value="#{msg.show}"></h:commandButton>
    <h:commandButton action="#{cardController.next}" value="#{msg.next}"
type="submit"></h:commandButton>
    <h:messages layout="table"></h:messages>

    <h:panelGroup binding="#{cardController.resultPanel}"
rendered="false">
        <h:message for="result"></h:message>
    </h:panelGroup>

</h:form>

</f:view>
</body>
</html>

```

From the previously examples you should be able to read most of the fields. What is new is that is use now the binding. Binding allows to bind certain UIControls to a managed bean. This way be bind the panel for the result to the controller. The controller can then set the rendered attribute of this UIControl depending on the user settings.

6.7. Run your webapplication

To run your webapplication, select Trainer.jsp, right mouse-click- >run as → run on server.

Train your Brain

Please calculate the result

Left Side 9

Right Side 74

Result

Correct

7. A Todo JSF application

Lets now create a JSF application for maintaining a Todo list. The main new thing we will cover is the handling of tables in JSF. These tables will be created based on a Java collection from the managed bean.

7.1. Create JSF Project

Create a new Dynamic Web Project "de.vogella.jsf.todo".

7.2. Domain model

Create a new package de.vogella.jsf.todo.model and the following class.

```
package de.vogella.jsf.todo.model;

import java.util.Calendar;

public class Todo {
    private String id;
    private String title;
    private String description;
    private int priority;
    private Calendar dueDate;

    public Todo(String title, String description, int priority) {
        this.title = title;
        this.description = description;
    }
}
```



```

        this.priority = priority;
    }

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }

    public int getPriority() {
        return priority;
    }

    public void setPriority(int priority) {
        this.priority = priority;
    }

    public Calendar getDueDate() {
        return dueDate;
    }
    public void setDueDate(Calendar dueDate) {
        this.dueDate = dueDate;
    }
}

```

7.3. Controller

Create the package `de.vogella.jsf.todo.controller` and the following class `TodoController`.

```

package de.vogella.jsf.todo.controller;

import java.util.ArrayList;
import java.util.List;

import javax.faces.component.UICommand;
import javax.faces.component.UIForm;
import javax.faces.event.ActionEvent;
import javax.faces.model.SelectItem;

```

```

import de.vogella.jsf.todo.model.TODO;

public class TodoController {
    // domain model related variables
    private List<TODO> todos;
    private TODO todo;

    // JavaServerFaces related variables
    private UIForm form;
    private UIForm tableForm;
    private UICommand addCommand;

    public TodoController() {
        todos = new ArrayList<TODO>();
        todos.add(new TODO("Learn JFS", "Finish this article", 1));
        todos.add(new TODO("Stop drinking too much coffee", "Coffee is evil!",
3));
    }

    public String addNew() {
        todo = new TODO("", "", 3);
        form.setRendered(true);
        addCommand.setRendered(false);
        return null;
    }

    public String save() {
        todos.add(todo);
        form.setRendered(false);
        addCommand.setRendered(true);
        return null;
    }

    public String cancel() {
        todo = null;
        form.setRendered(false);
        addCommand.setRendered(true);
        return null;
    }

    public String delete() {
        todos.remove(todo);
        return null;
    }

    public void displayTable(ActionEvent event) {
        if (event.getComponent().getId().equalsIgnoreCase("hide")) {
            tableForm.setRendered(false);
        } else {
            tableForm.setRendered(true);
        }
    }

    public List<SelectItem> getPriorities() {
        List<SelectItem> list = new ArrayList<SelectItem>();
        list.add(new SelectItem(1, "High"));
        list.add(new SelectItem(2, "Medium"));
    }
}

```

```

        list.add(new SelectItem(3, "Low"));
        return list;
    }

    public List<Todo> getTodos() {
        return todos;
    }

    public void setTodos(List<Todo> todos) {
        this.todos = todos;
    }

    public Todo getTodo() {
        return todo;
    }

    public void setTodo(Todo todo) {
        this.todo = todo;
    }

    public UIForm getForm() {
        return form;
    }

    public void setForm(UIForm form) {
        this.form = form;
    }

    public UICommand getAddCommand() {
        return addCommand;
    }

    public void setAddCommand(UICommand addCommand) {
        this.addCommand = addCommand;
    }

    public UIForm getTableForm() {
        return tableForm;
    }

    public void setTableForm(UIForm tableForm) {
        this.tableForm = tableForm;
    }
}

```

7.4. Register your managed beans

Double-click on faces-config.xml and select the tab "ManagedBeans". Register the TodoController.

7.5. Create css

In your folder WebContent create a folder css. Create a file mystyle.css with the following content.

```
table.todo {
    border: 1px solid #CCCCCC;
}

table.todo th {
    background: #EFEFEF none repeat scroll 0 0;
    border-top: 1px solid #CCCCCC;
    font-size: small;
    padding-left: 5px;
    padding-right: 4px;
    padding-top: 4px;
    vertical-align: top;
}

table td:first-child {
    font-weight: bold;
}

table .first {
    width: 120px;
}

table .rest {
    width: 400px;
}

table.todo {
    border-top: 1px solid #CCCCCC;
    font-size: small;
    padding-left: 5px;
    padding-right: 4px;
    padding-top: 4px;
    vertical-align: top;
}
```

7.6. JavaServer Page with JSF components

Create a new JSP page "Todo.jsp" and change the code to the following:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>

<LINK href="<%=request.getContextPath()%>/css/mystyle.css"
```

```

    rel="stylesheet" type="text/css">
</head>
<body>
<h3>Todo list</h3>
<f:view>
    <h:messages layout="table"></h:messages>

    <!-- Possibility to start a new Todo -->
    <h:form>
        <h:commandLink binding="#{todoController.addCommand}" accesskey="n"
            action="#{todoController.addNew}" value="Add new Todo">
        </h:commandLink>
    </h:form>

    <h:form binding="#{todoController.form}" rendered="false"
        styleClass="todo">
        <h:panelGrid columns="2">
            <h:outputText value="Title"></h:outputText>
            <h:inputText value="#{todoController.todo.title}" required="true"
                requiredMessage="Title is required">
            </h:inputText>
            <h:outputText value="Description"></h:outputText>
            <h:inputTextarea value="#{todoController.todo.description}"
cols="40"
                rows="4"></h:inputTextarea>
            <h:outputText value="Prio"></h:outputText>
            <h:selectOneMenu validatorMessage="required"
                value="#{todoController.todo.priority}">
                <f:selectItems value="#{todoController.priorities}" />
            </h:selectOneMenu>
        </h:panelGrid>
        <h:panelGroup>
            <h:commandButton action="#{todoController.save}" value="Save"
                accesskey="s">
            </h:commandButton>
            <h:commandButton action="#{todoController.cancel}" value="Cancel"
                accesskey="c" immediate="true">
            </h:commandButton>
        </h:panelGroup>
    </h:form>

    <!-- These buttons allow to show and hide the table -->
    <h:form>
        <h:panelGrid columns="2">
            <h:commandLink id="hide"
                actionListener="#{todoController.displayTable}" value="Hide
Table">
            </h:commandLink>
            <h:commandLink id="show"
                actionListener="#{todoController.displayTable}" value="Show
Table">
            </h:commandLink>
        </h:panelGrid>
    </h:form>

    <!-- Here we start the form for the data table -->

```

```

<h:form binding="#{todoController.tableForm}">
  <!-- Here we start the data table -->

  <h:dataTable value="#{todoController.todos}" var="todo"
    styleClass="todo" headerClass="todoheader"
    columnClasses="first, rest">
    <h:column>
      <!-- Via this facet we define the table header (column 1) --
%>
      <f:facet name="header">
        <h:column>
          <h:outputText value="Prio"></h:outputText>
        </h:column>
      </f:facet>
      <h:outputText value="#{todo.priority}"></h:outputText>
    </h:column>
    <h:column>
      <!-- Via this facet we define the table header (column 2) --
%>
      <f:facet name="header">
        <h:column>
          <h:outputText value="Title"></h:outputText>
        </h:column>
      </f:facet>
      <h:outputText value="#{todo.title}"></h:outputText>

    </h:column>

    <h:column>
      <!-- Via this facet we define the table header (column 3) --
%>
      <f:facet name="header">
        <h:column>
          <h:outputText value="Description"></h:outputText>
        </h:column>
      </f:facet>
      <h:outputText value="#{todo.description}"></h:outputText>
    </h:column>

    <h:column>
      <!-- Via this facet we define the table header (column 4) --
%>
      <f:facet name="header">
        <h:column>
          <h:outputText value="Actions"></h:outputText>
        </h:column>
      </f:facet>
      <h:panelGrid columns="2">
        <h:commandLink value="delete"
action="#{todoController.delete}">
          <f:setPropertyActionListener
target="#{todoController.todo}"
          value="#{todo}" />
        </h:commandLink>
      </h:panelGrid>
    </h:column>
  </h:dataTable>

```

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

Using the Facet tag you can create a header for a dataTable component.

You have added a actionListener. This can call a method which can receive an object of type `ActionEvent`. `actionListeners` are nice if you want to use the same method with different parameters. We also use `selectOneMenu` which allows to select a value from a pre-defined list. The main new thing here is `h:datatable` tag. This tag defines a table. `value` can get a list as a parameter and `var` define the variable which will be used to create each row. This is very similar to the `foreach` loop. The other new element is the `setPropertyActionListener`. This allow you to listener to changes for this link, e.g. a mouse click. This copies the current selected row into the field `todo`. The method `delete` from the controller will then remove this elements from the list.

7.7. Run your webapplication

If you run your webapplication you should be seeing the following:

Todo list

Title

Description

This is medium important

^

v

Prio

Medium v

Save

Cancel

[Hide Table](#) [Show Table](#)

Prio	Title	Description	
1	Learn JFS	Finish this article	delete
3	Stop drinking to much coffee	Coffee is evil!	delete

8. Links and Literature

8.1. Tutorials and Websites

[Information about JavaServer Faces](#)

[Getting started with JavaServer Faces 1.2, Part 1: Building basic applications](#)

[Getting started with JavaServer Faces 1.2, Part 2: JSF life cycle, conversion, validation, and phase listeners](#)

[Unified Expression Language](#)

8.2. JSF component libraries

The following lists JSF implementations or JSF extension implementations.

[Apache MyFaces](#)

[JBoss Rich Faces](#)

[Introduction to building JSF with Eclipse](#)

If you need more assistance we offer [Online Training](#) and [Onsite training](#) as well as [consulting](#)