

# JSF 2.0 PROGRAMMING COOKBOOK

HOT RECIPES FOR JSF DEVELOPMENT



JavaServer™ Faces  
**JSF**

**MARTIN MOIS**



**Java Code Geeks**  
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

# **JSF 2.0 Programming Cookbook**

---

# Contents

<b>1</b>	<b>Eclipse IDE support</b>	<b>1</b>
1.1	Eclipse Project Facets . . . . .	1
1.2	Demo . . . . .	6
<b>2</b>	<b>Hello World Example</b>	<b>7</b>
2.1	Project Environment . . . . .	7
2.2	JSF 2.0 Dependencies . . . . .	8
2.3	JSF Managed Bean . . . . .	9
2.4	JSF Pages . . . . .	10
2.5	JSF 2.0 Servlet Configuration . . . . .	11
2.6	Demo . . . . .	13
2.7	Closing Words . . . . .	13
<b>3</b>	<b>Ajax Example</b>	<b>15</b>
3.1	JSF Page . . . . .	15
3.2	Managed Bean . . . . .	16
3.3	Demo . . . . .	17
<b>4</b>	<b>Managed Beans Example</b>	<b>18</b>
4.1	What is JSF 2.x? . . . . .	18
4.2	Why use JSF 2.x? . . . . .	18
4.3	What is a Managed Bean? . . . . .	18
4.4	What we need to start? . . . . .	19
4.5	Step By Step . . . . .	19
4.5.1	Create a Dynamic Web Project . . . . .	19
4.5.2	Configure Java EE Project . . . . .	20
4.5.3	Configuration Files . . . . .	21
4.5.4	JSF Pages . . . . .	23
4.5.5	Java Managed Bean . . . . .	24
4.5.6	Running the Example . . . . .	26
4.5.7	Key Points . . . . .	26
4.6	Download the Eclipse Project . . . . .	27

---

---

<b>5</b>	<b>TextBox Example</b>	<b>28</b>
5.1	Managed Bean . . . . .	28
5.2	Our Pages . . . . .	28
5.3	Demo . . . . .	29
<b>6</b>	<b>Password Example</b>	<b>31</b>
6.1	Managed Bean . . . . .	31
6.2	Our Pages . . . . .	31
6.3	Demo . . . . .	32
<b>7</b>	<b>TextArea Example</b>	<b>34</b>
7.1	Managed Bean . . . . .	34
7.2	Our Pages . . . . .	34
7.3	Demo . . . . .	35
7.4	Some Closing Words . . . . .	37
<b>8</b>	<b>CheckBox Example</b>	<b>38</b>
8.1	Backing Bean . . . . .	38
8.2	Our JSF Pages . . . . .	41
8.3	Demo . . . . .	42
<b>9</b>	<b>OutputText Example</b>	<b>45</b>
9.1	Managed Bean . . . . .	45
9.2	Our JSF Page . . . . .	46
9.3	Demo . . . . .	47
<b>10</b>	<b>Button and CommandButton Example</b>	<b>48</b>
10.1	h:commandButton tag . . . . .	48
10.2	h:button tag . . . . .	49
<b>11</b>	<b>PanelGrid Example</b>	<b>50</b>
11.1	The Example . . . . .	50
11.2	The Managed Bean . . . . .	51
11.3	The JSF Pages . . . . .	51
11.4	The Demo . . . . .	52
<b>12</b>	<b>Message and Messages Example</b>	<b>54</b>
12.1	The Demo . . . . .	55
<b>13</b>	<b>Param and Attribute Example</b>	<b>58</b>
13.1	Param Tag . . . . .	58
13.1.1	The Full Example . . . . .	58
13.2	Attribute Tag . . . . .	61

---

---

<b>14 Standard Validators Example</b>	<b>63</b>
14.1 Project Environment	63
14.2 JSF Pages	64
14.3 Managed Bean	65
14.4 Demo	66
14.5 Download the Eclipse Project	67
<b>15 Internationalization Example</b>	<b>68</b>
15.1 Project Environment	68
15.2 Properties files	69
15.3 Configuration of faces-config.xml	70
15.4 Managed Bean	70
15.5 JSF Page	71
15.6 Demo	71
15.7 Download the Eclipse Project	72
<b>16 Facelets Templates Example</b>	<b>73</b>
16.1 Create a new Maven Project	73
16.2 Modify POM to include JSF dependency	79
16.3 Add Faces Servlet in web.xml	80
16.4 JSF Facelets Tags	80
16.5 Create a Template	81
16.5.1 5.1 Create Header	81
16.5.2 5.2 Create Content	81
16.5.3 5.3 Create Footer	82
16.5.4 5.4 Finally the Template	82
16.6 Default page using template	83
16.7 Welcome page using template	84
16.8 Download the Eclipse Project	86
<b>17 Standard Converters Example</b>	<b>87</b>
17.1 Create a new Maven Project	87
17.2 Modify POM to include JSF dependency	94
17.3 Add Faces Servlet in web.xml	95
17.4 Standard Converters	95
17.5 How to use Converters	96
17.5.1 5.1 Using converter attribute	96
17.5.2 5.2 Using f:converter tag	96
17.5.3 5.3 Using converter tags	96
17.6 Implicit Converter	97
17.7 DateTimeConverter	101
17.8 NumberConverter	104
17.9 Download the Eclipse Project	108

---

---

<b>18 Components Listeners Example</b>	<b>109</b>
18.1 Create a new Maven Project	109
18.2 Modify POM to include JSF dependency	116
18.3 Add Faces Servlet in web.xml	117
18.4 Value Change Listener	117
18.4.1 4.1 Using valueChangeListener attribute	118
18.4.2 4.2 Using valueChangeListener Tag	120
18.5 Action Listener	122
18.5.1 5.1 Using actionListener attribute	124
18.5.2 5.2 Using actionListener Tag	126
18.6 Download the Eclipse Project	129

---

Copyright (c) Exelixis Media P.C., 2015

All rights reserved. Without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored or introduced into a retrieval system, or transmitted, in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of the copyright owner.

---

# Preface

JavaServer Faces (JSF) is a Java specification for building component-based user interfaces for web applications.[1] It was formalized as a standard through the Java Community Process and is part of the Java Platform, Enterprise Edition.

JSF 2 uses Facelets (an open source Web template system) as its default templating system. Other view technologies such as XUL can also be employed. In contrast, JSF 1.x uses JavaServer Pages (JSP) as its default templating system. (Source: [http://en.wikipedia.org/wiki/JavaServer\\_Faces](http://en.wikipedia.org/wiki/JavaServer_Faces))

In this ebook, we provide a compilation of JSF 2.0 based examples that will help you kick-start your own web projects. We cover a wide range of topics, from setting up the environment and creating a basic project, to more advanced concepts like Internationalization and Facelets Templates. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

---



## About the Author

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at <http://www.javacodegeeks.com/>

---

# Chapter 1

## Eclipse IDE support

As the title clarifies, in my very first tutorial, we are going to take a glance at setting our Eclipse IDE, in order to support **JSF 2.0**. Older Eclipse EE versions, such as Ganymede (v3.4) and Galileo (v3.5), support only JSF 1.2. In order to work with JSF 2.0, you just have to own an Eclipse EE version of Helios (v3.6) or onward, which has by default full support of Java EE 7, including JSF 2.0.

Here's the guide that will show you how to enable JSF 2.0 features in your Eclipse IDE.

Tools Used :

- Eclipse EE Kepler (v 4.3)
- JSF 2.2

### 1.1 Eclipse Project Facets

First, we have to create a JSF custom library, which will be available each time we want to cooperate with JSF. The most common and easy way to do this, is to configure an existing project, in order to support **Web Tools Platform (WTP)**.

Steps to enable the Web Tools Platform (WTP) :

- Right click on an existing project and select **Properties**
- While on Properties window, select **Project Facets**
- Make sure that the version of your **Dynamic Web Module** is at least 2.5
- Tick the **Java** checkbox and select the 1.6 (or 1.7 if it depends on what is the latest version of Java that you have installed) version.
- Tick the **JavaServer Faces** checkbox and select 2.2 version.

You should now be informed that a further configuration is required. Hover over the link and click it!

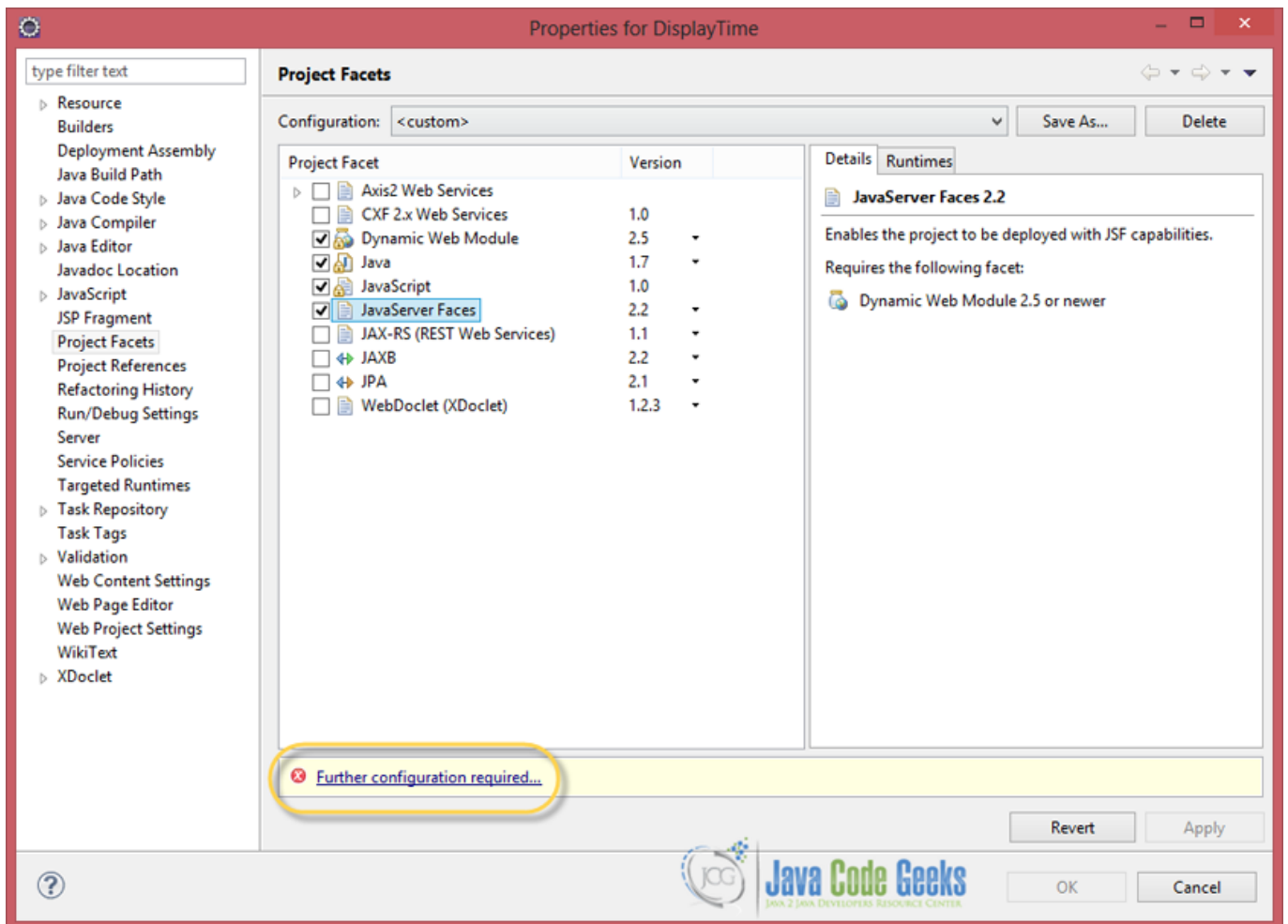


Figure 1.1: image1u

Now it's the time to create our stable User Library:

- In the **Modify Faceted Project** window, click the download icon

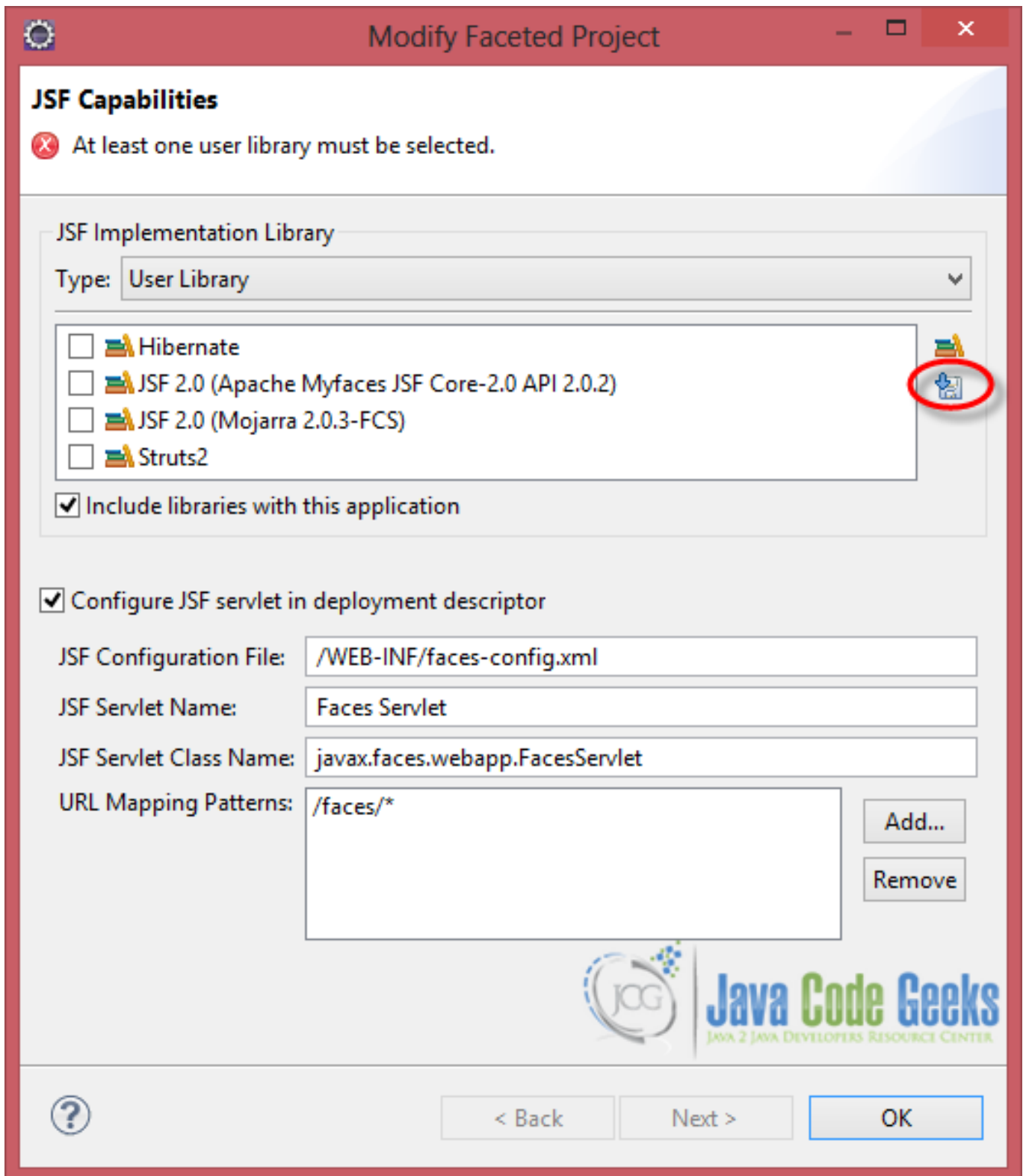


Figure 1.2: configureJSF\_2u

You should at least see a JSF 2.2 library, as shown below (here, we do not have any other available views, because our example pc already supports 2.0, as you correctly noticed) :

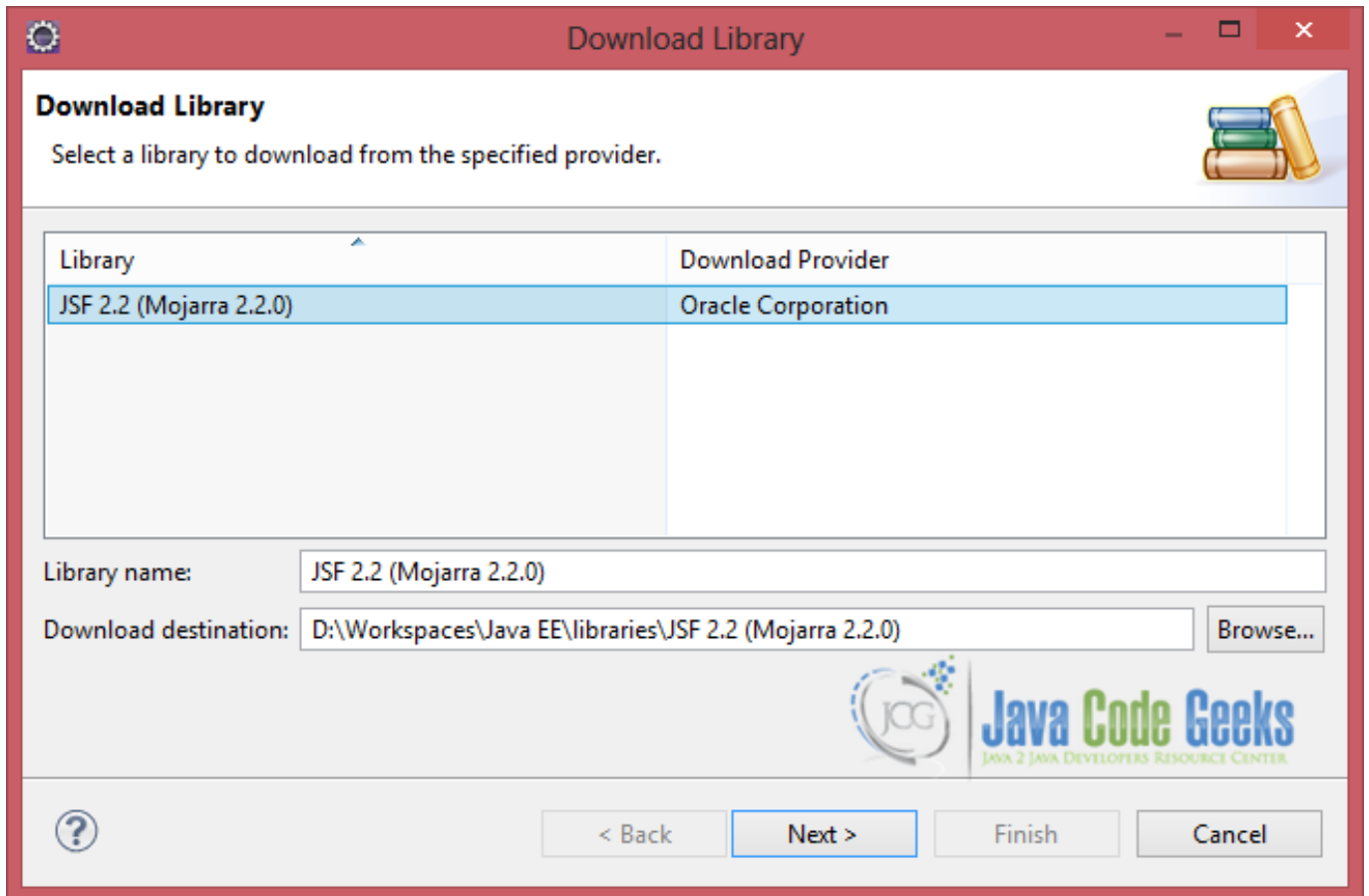


Figure 1.3: configureJSF\_3u

- Click **Next**, accept the License Agreement and hit **Finish**
- Check **JSF 2.2 (Mojarra 2.2.0)** and hit **OK**.

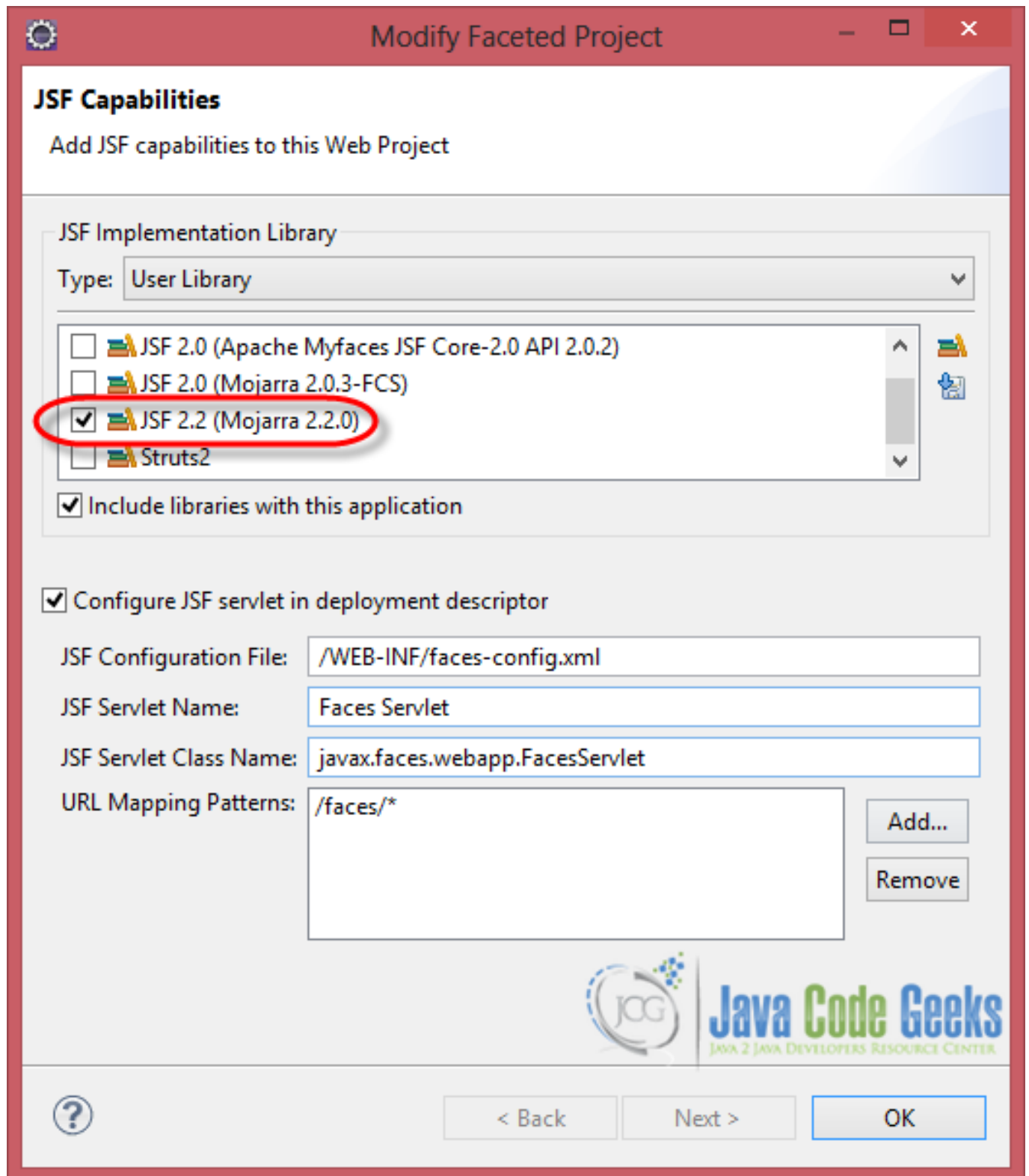


Figure 1.4: configureJSF\_4u

## 1.2 Demo

Success! You just accomplished to configure JSF 2.2 on Eclipse IDE!. Let's try it out, by creating a sample `.xhtml` file, which will be the very first page in our website. I'm sure you know how to implement such an easy action, but just to follow along:

- Right click the project's **Web Content** folder.
- Hit **New** ⇒ **HTML File**.
- Name your file `index.xhtml`
- Select the `1.0 strict xhtml` template and hit **Finish**

And a small annotation : if you want to make your life easier, especially according to the front-end part of your application, Eclipse EE also provides a **Web Page Editor**. Let's see what I mean :

- Right click on the `index.xhtml` you just created.
- Select **Open With** ⇒ **Web Page Editor**.

Yeah, that's it! This view provides a full Web Palette to assist you while developing your J2EE applications! Just in case you didn't noticed what happened, here is the exact screenshot :

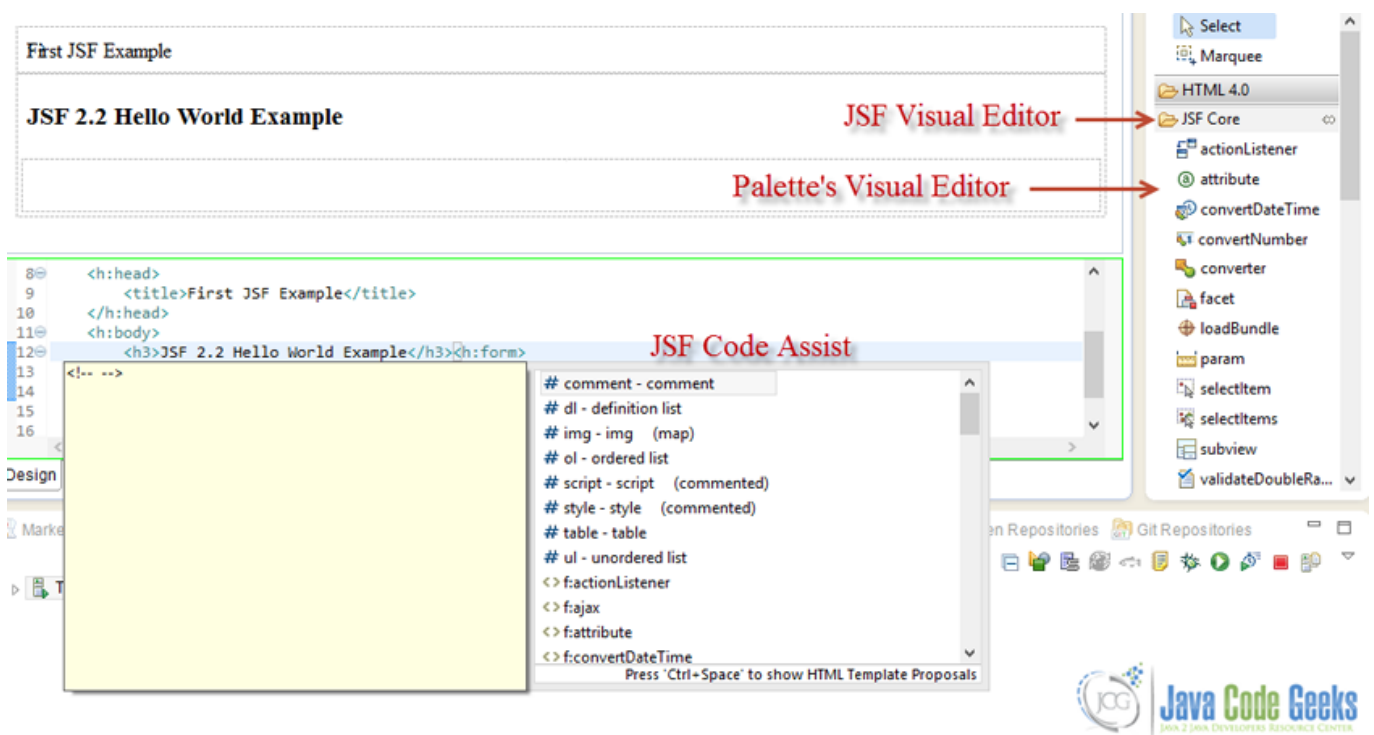


Figure 1.5: configure\_JSf5u

What comes after configuring? Getting up and running, of course ; and you guessed it right, the fore-mentioned picture is from next example's JSF application.

This was an example of configuring Eclipse IDE in order to support JSF 2.0.

So, in our next JSF example, we 'll find out how to develop a simple `Hello World` application. Stay tuned!

## Chapter 2

# Hello World Example

As I promised in my previous [article](#), in this example, we are going to develop a simple Hello World application, with Javaserer Faces (JSF) 2.0. It may seem a bit of handy, but following along will make you understand how to easily configure every related project. So, let's start!

### 2.1 Project Environment

This example was implemented using the following tools :

- JSF 2.2
- Maven 3.1
- Eclipse 4.3 (Kepler)
- JDK 1.6
- Apache Tomcat 7.0.41

Let's first have a look at the final project's structure, just to ensure that you won't get lost anytime.

---



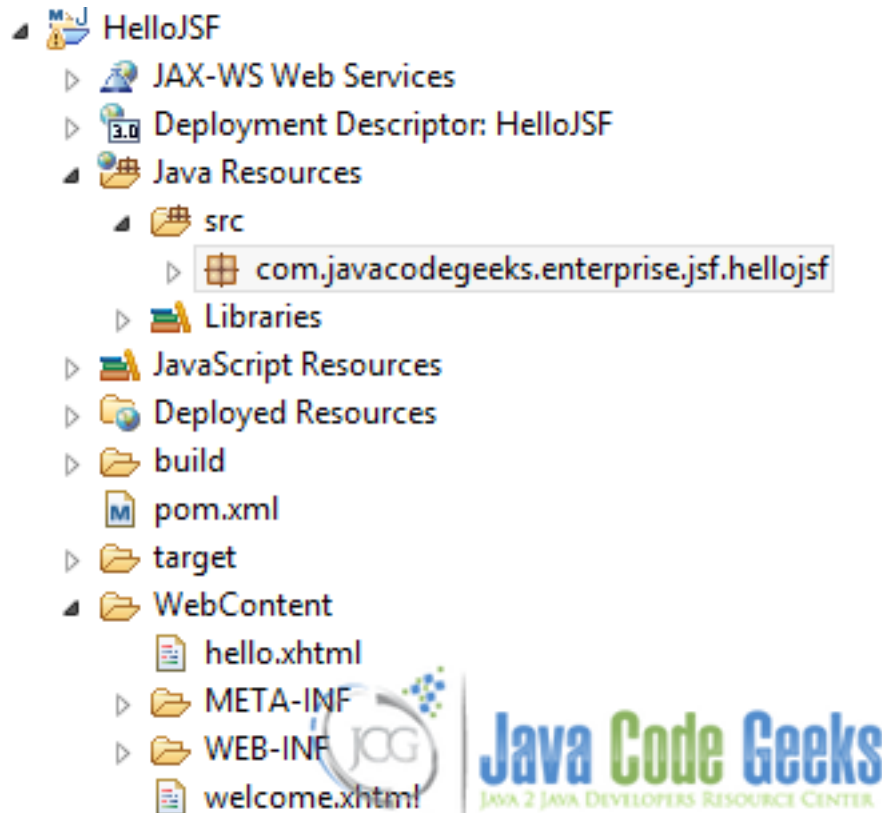


Figure 2.1: helloJSF\_1

That is, just start by creating a **Dynamic Web Project** using **Maven**; I'm sure you're quite experienced about how to do let's get into the more technical part.

## 2.2 JSF 2.0 Dependencies

First, we need to configure the `pom.xml` file, in order to support **JSF**. This can be done with two ways. The first way is to add each single dependency manually, by right-clicking on the project and selecting **Maven** ⇒ **Add Dependency**; this way is accepted as easier, because you can have an auto-generated `pom.xml` file. The second way is what you exactly imagined, you just have to write by hand, everything that is required for this example's purpose. So, here is the `pom.xml` file.

*pom.xml*

```
<modelVersion>4.0.0</modelVersion>
<groupId>com.javacodegeeks.enterprise.jsf</groupId>
<artifactId>hellojsf</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>war</packaging>
<build>
  <sourceDirectory>src</sourceDirectory>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <version>2.3</version>
  <configuration>
    <warSourceDirectory>WebContent</warSourceDirectory>
    <failOnMissingWebXml>>false</failOnMissingWebXml>
  </configuration>
</plugin>
</plugins>
</build>
<dependencies>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.2.4</version>
  </dependency>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>mojarra-jsf-impl</artifactId>
    <version>2.0.0-b04</version>
  </dependency>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>mojarra-jsf-api</artifactId>
    <version>2.0.0-b04</version>
  </dependency>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>2.2.4</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <version>2.3.1</version>
  </dependency>
</dependencies>
</project>
```

## 2.3 JSF Managed Bean

A **Managed Bean** is a regular Java Bean class, registered with JSF. In other words, Managed Bean is a java bean, managed by the JSF framework. For more information about Managed Bean, check [here](#). From **JSF 2.0** and onwards, we can declare a managed bean, just by using the annotation `@ManagedBean`. Let's see how should the managed bean's class structure be.

### >HelloBean.java

```
package com.javacodegeeks.enterprise.jsf.hellojsf;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import java.io.Serializable;

@ManagedBean
@SessionScoped
public class HelloBean implements Serializable {

    private static final long serialVersionUID = 1L;

    private String name;

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

Why do we need to implement the **Serializable** interface? Here is the absolute [answer](#). As for the two annotations that our managed bean uses, any question that may occur, is answered [here](#).

## 2.4 JSF Pages

In **JSF**, we usually treat the static content of our website, using `xhtml`, instead of simple `html`. So, follow along, in order to create our first `xhtml` page, which will prompt the user to enter his name in a text field and provide a button in order to redirect him to a welcome page:

- Right click on the **Web Content** folder
- Select **New** ⇒ **HTML File** (if you can't find it, just select **Other** and the wizard will guide you through it).
- In the **File Name**, type `hello.xhtml` and hit **Next**.
- Select the `xhtml 1.0 strict` template.
- Hit **Finish**.

Ok, good till here, but we have to do one more thing, in order to enable **JSF** components/features in our `xhtml` files: we just need to declare the **JSF namespace** at the beginning of our document. This is how to implement it and make sure that you 'll always care about it, when dealing with **JSF** and **XHTML**, together:

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

Yeah! Now that we got everything set up, let's move into writing the required code for our `hello.xhtml` file.

*hello.xhtml*

```
<?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:f="http://java.sun.com/jsf/core"
```

```

xmlns:h="http://java.sun.com/jsf/html">

<h:head>
<title>First JSF Example</title>
</h:head>
<h:body>
<h3>JSF 2.2 Hello World Example</h3><h:form>
What's your name?
<h:inputText value="#{helloBean.name}"></h:inputText>
<h:commandButton value="Welcome Me" action="welcome"></h:commandButton>
</h:form>
</h:body>
</html>

```

What's going on here? Nothing, absolutely nothing! As I fore-mentioned, we just have an `inputText`, where the user will enter his name and a button (which can be declared by using the `commandButton` xhtml tag), which has an interactive role by redirecting him to a `welcome` page, when clicked. I also know that you almost understood what the `action` parameter is used for: it's the way to tell the browser where to navigate, in case our buttons gets clicked. So, here we want to navigate to the `welcome` page (yes, we don't have to clarify the suffix, too; that's why I left it as is), where the user will get a greeting from our application. Quite experienced right now, you can create by yourself the `welcome.xhtml` and provide a sample greeting, as below.

*welcome.xhtml*

```

<?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://java.sun.com/jsf/html">

  <h:head>
    <title>Welcome</title>
  </h:head>
  <h:body bgcolor="white">
    <h3>Everything went right!</h3>
    <h4>Welcome #{helloBean.name}</h4>
  </h:body>
</html>

```

*Did I miss something? For sure, but not really important, for you, new developers! Things were not so easy, in JSF 1.x, as we had to declare the fore-mentioned **navigation rule**, in `faces-config.xml` file. `faces-config.xml` allows to configure the application, managed beans, convertors, validators, and navigation. As for the navigation, while cooperating with JSF 2.0 and onwards, we can put the page name directly in the button's "action" attribute. To get rid of any other questionmarks that may appear, please read [this](#).*

One last thing, before moving to the last project configuration: just in case you didn't make it clear, a `"#{...}"` indicates a **JSF** expression and in this case, when the page is submitted, JSF will find the "helloBean" with the help of `#{helloBean.name}` expression and set the submitted `inputText`'s value, through the `setName()` method. When `welcome.xhtml` page will get displayed, JSF will find the same session `helloBean` again and display the name property value, through the `getName()` method.

## 2.5 JSF 2.0 Servlet Configuration

Finally, we need to set up JSF in the `web.xml` file, just like we are doing in any other **J2EE** framework.

*web.xml*

```

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

```

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/ ↵
  xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com ↵
  /xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>HelloJSF</display-name>
  <welcome-file-list>
    <welcome-file>faces/hello.xhtml</welcome-file>
  </welcome-file-list>
  <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>
  <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>*.xhtml</url-pattern>
  </servlet-mapping>
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <listener>
    <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
  </listener>
</web-app>
```

What we actually did:

- Defined our `hello.xhtml` page, as the first page that will be displayed, when the project's URL, will be accessed.
- Defined a `javax.faces.webapp.FacesServlet` mapping and mapped the application to the most used **JSF** file extensions (`/faces/*`, `*.jsf`, `*.xhtml`, `*.faces`).

So, all following URLs will end up to the same `hello.xhtml` file :

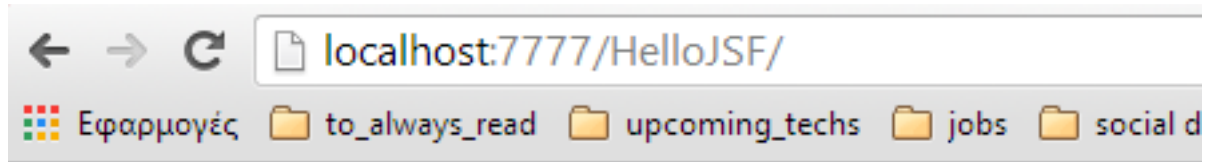
- `http://localhost:8080/HelloJSF/hello.jsf`
- `http://localhost:8080/HelloJSF/hello.faces`
- `http://localhost:8080/HelloJSF/hello.xhtml`
- `http://localhost:8080/HelloJSF/faces/hello.xhtml`

*Tip: In JSF 2.0 development, it's good to set the `javax.faces.PROJECT_STAGE` to `Development`, while you are in a "debugging" mode, 'cause it will provide many useful debugging information to let you track the bugs easily. When in deployment, you can change it to `Production`, because noone of us, wants his customers staring at the debugging information.*

## 2.6 Demo

Thank God, time to run!

This is what you should get (don't get confused about my port number - I just have my 8080 port, occupied):



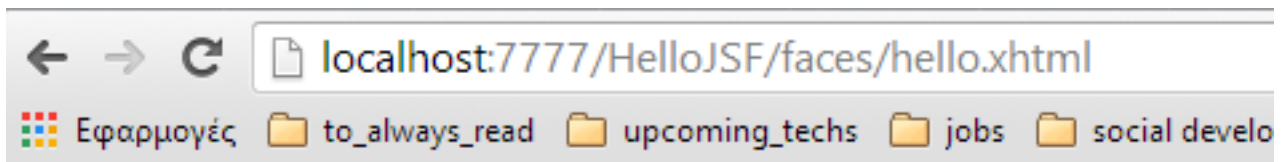
### JSF 2.2 Hello World Example

What's your name?



Figure 2.2: helloJSF2

And after clicking the button:



**Everything went right!**

**Welcome Thodoris**



Figure 2.3: image3

## 2.7 Closing Words

So, that was it! Exhausting? Maybe. Interesting? Definitely! We had to dive into detail in this example, because we'll keep the same structure for the next (where we'll get our hands dirty with Ajax), too.

This was an example of a simple Hello World application using JSF 2.0. You can download the Eclipse Project of this example: [HelloJSF](#)

---

## Chapter 3

# Ajax Example

Hi there, you do remember right from my [last](#) example, today we 're gonna talk about integrating **JSF** together with **Ajax**. Ajax stands for Asynchronous Javascript and XML and is also a helpful technique for creating web pages with dynamic content (i.e. when you want to update a single component in your web page, instead of updating the whole page). This is implemented asynchronously by exchanging small amounts of data with the server, in the back-end. For further details about today's example, refer [here](#).

Everything is set up as is, in the previous two JSF examples, so we 'll dive quickly, into the technical part of the example.

### 3.1 JSF Page

Here is our JSF page that supports Ajax.

*helloAjax.xhtml*

```
<?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:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">

  <h:head>
    <title>Ajax JSF Example</title>
  </h:head>
  <h:body>
    <h3>JSF 2.2 Hello Ajax Example</h3>

    <h:form>
      <h:inputText id="name" value="#{helloBean.name}"></h:inputText>
      <h:commandButton value="Welcome Me">
        <f:ajax execute="name" render="output"/>
      </h:commandButton>
      <h2><h:outputText id="output" value="#{helloBean.sayWelcome}" /></h2>
    </h:form>
  </h:body>
</html>
```

We 'll keep the same base with the previous example, so what will make our site Ajaxable, is the button. Thus, when in clicked, it will send a request to the server, instead of submitting the whole form.

The key code branch of the above snippet is this:



```
<h:commandButton value="Welcome Me">
    <f:ajax execute="name" render="output"/>
</h:commandButton>
<h2><h:outputText id="output" value="#{helloBean.sayWelcome}" /></h2>
```

What enables Ajax features of our page is the tag `f:ajax`. There are two processes that take part there:

- The `execute="name"` property indicated that a component, which id is equal to `name`, will be sent to server for processing. That is, the server is going to search the form component that matches the required id. In our case, we're talking about the value of our `inputText`. Keep in mind, that if we had more than one components that we might want to send to server for processing, we could just split them with a space character; for example, `execute="id1 id2 id3"`.
- What comes after a request (in normal circumstances)? A response, of course, so our `render="output"` means that the server's response will be rendered to a component, which id is `output`. Thus, in our case, the response will be displayed in the `outputText` component.

## 3.2 Managed Bean

Before reading the code for `HelloBean.java`, a small quote for the fore-mentioned key code snippet: the action that will make the `outputText` take the `inputText`'s value, is its property of `value="#{helloBean.sayWelcome}"`. If something isn't clear till here, you have to refer to the previous example.

All right, here's the structure of `HelloBean.java`.

*HelloBean.java*

```
import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class HelloBean implements Serializable{

    private static final long serialVersionUID = 1L;

    private String name;

    public String getName()
    {
        return name;
    }

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

    public String getSayWelcome()
    {
        if("").equals(name) || name == null)
        {
            return "";
        }
        else{
            return "Ajax message : Welcome " + name ;
        }
    }
}
```

### 3.3 Demo

You can test your application by simply hitting **Run** in your IDE, or by accessing the following URL: `http://localhost:8080/AjaxJSFexample/`:

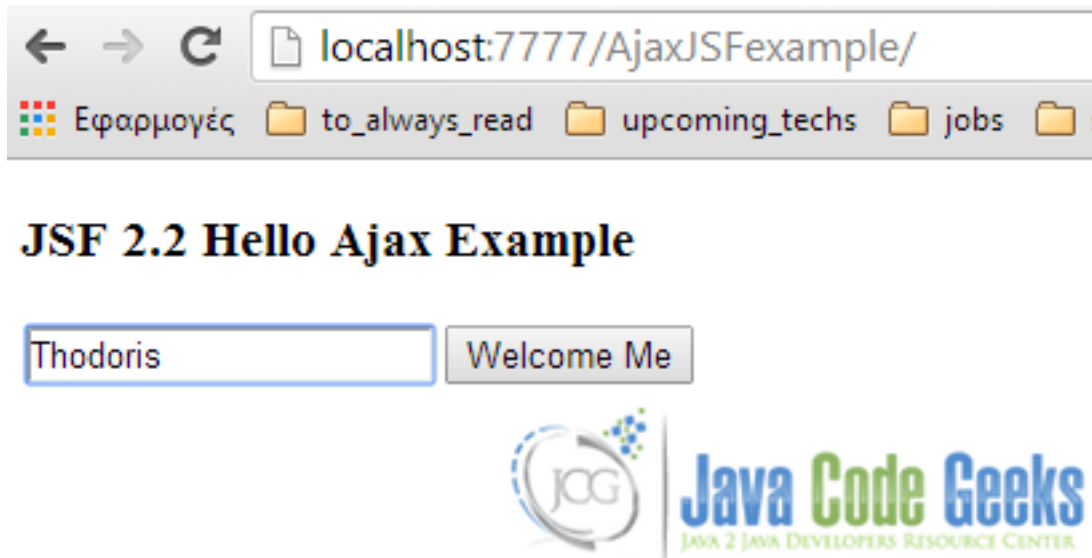


Figure 3.1: imageA

What are we waiting for, after the button is clicked? The server's response, which is implemented from the `getSayWelcome()` method, without having the whole page refreshed:

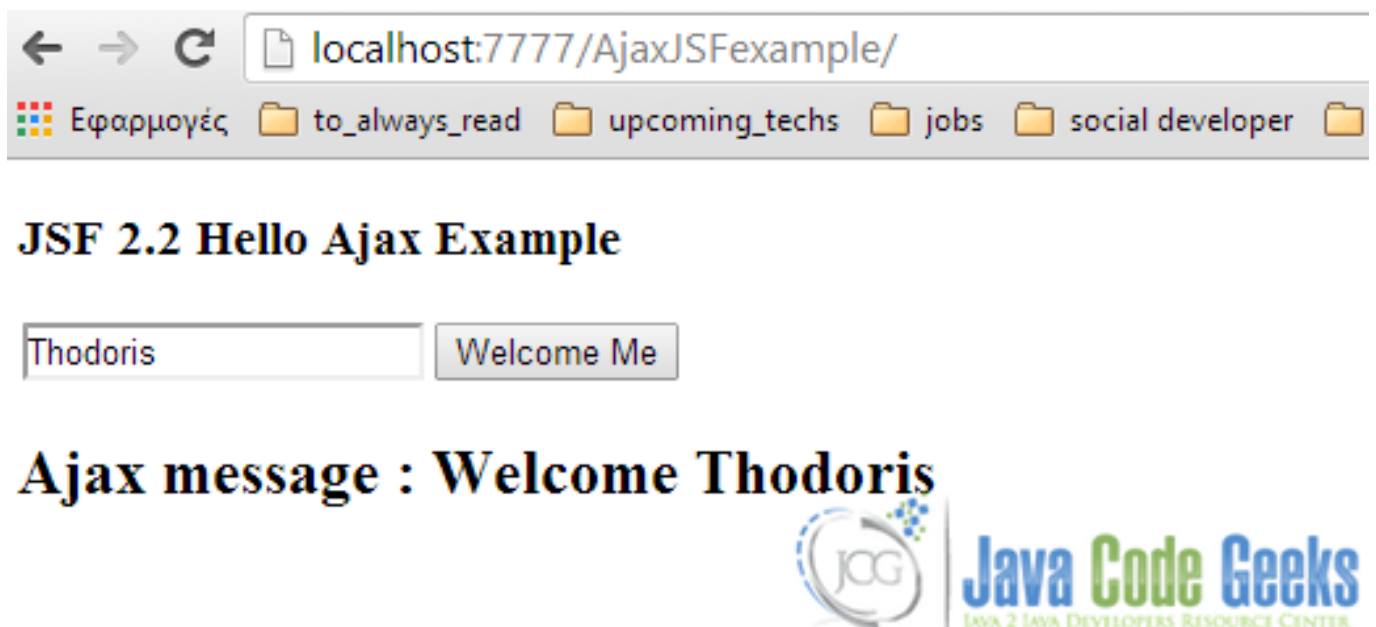


Figure 3.2: AjaxJSF\_B

This was an example of integrating Ajax together with JSF. You can also download the source code: [AjaxJSFexample](#)

## Chapter 4

# Managed Beans Example

In this example we will talk about the basics of the JSF Standard and the Managed Bean concept. With these tools we can develop and support an application's clear architecture, and get an easily integration with java EE modern technologies. Then, we learn how to work with the backend, using JSF Managed Beans.

### 4.1 What is JSF 2.x?

JSF 2.0, 2.1 or 2.2 are the new releases of the Java Server Faces (consider it like a standard and a framework of development). Those improve the development cycle of java web-based applications, applying the Contexts and Dependency Injection (CDI), Model-View-Controller (MVC) architecture and Inversion of control (IoC) . They also include various features that make it a robust framework for java web applications. Well, JSF 2.x is the evolution of JSF 1.x that help us to easily integrate with new technologies like Ajax, Restful, HTML5, new servers like Glassfish 4, Wildfly (the old JBoss) and the last JDK Java EE 7 and 8.

### 4.2 Why use JSF 2.x?

There are many improvements and advantages regarding its previous version, here are some:

- It's the official Java EE Web Application Library
- The configuration fully supports annotations, instead of heavy xml description.
- Full support to ajax
- New GUI components, named Facelets instead of old JSP pages.
- More view scopes and improved component's life cycle
- Support to the new Java EE Servers

### 4.3 What is a Managed Bean?

It's a lightweight Java Bean that is registrered with the framework, to contain the Backend data, and that may contain business logic. Which are the properties of a JSF 2.2 Managed Bean?

- Has the annotation @ManagedBean
  - Has a zero-arg constructor
  - Has a setter and getter public methods, for their private instance variables
-

## 4.4 What we need to start?

As with every java web application, we need a JSF implementation (Oracle Mojarra, Apache MyFaces), Java EE Server and our main IDE. So, for this example we use:

- JSF 2.2.9 (Mojarra version, we recommend them)
- Apache Tomcat 7
- Eclipse Luna IDE
- JDK 1.7.0\_67

## 4.5 Step By Step

### 4.5.1 Create a Dynamic Web Project

First download the JSF 2.2 Mojarra [here](#), then create a new Dynamic Web Project, and add the JSF Library. Our project must look like these:



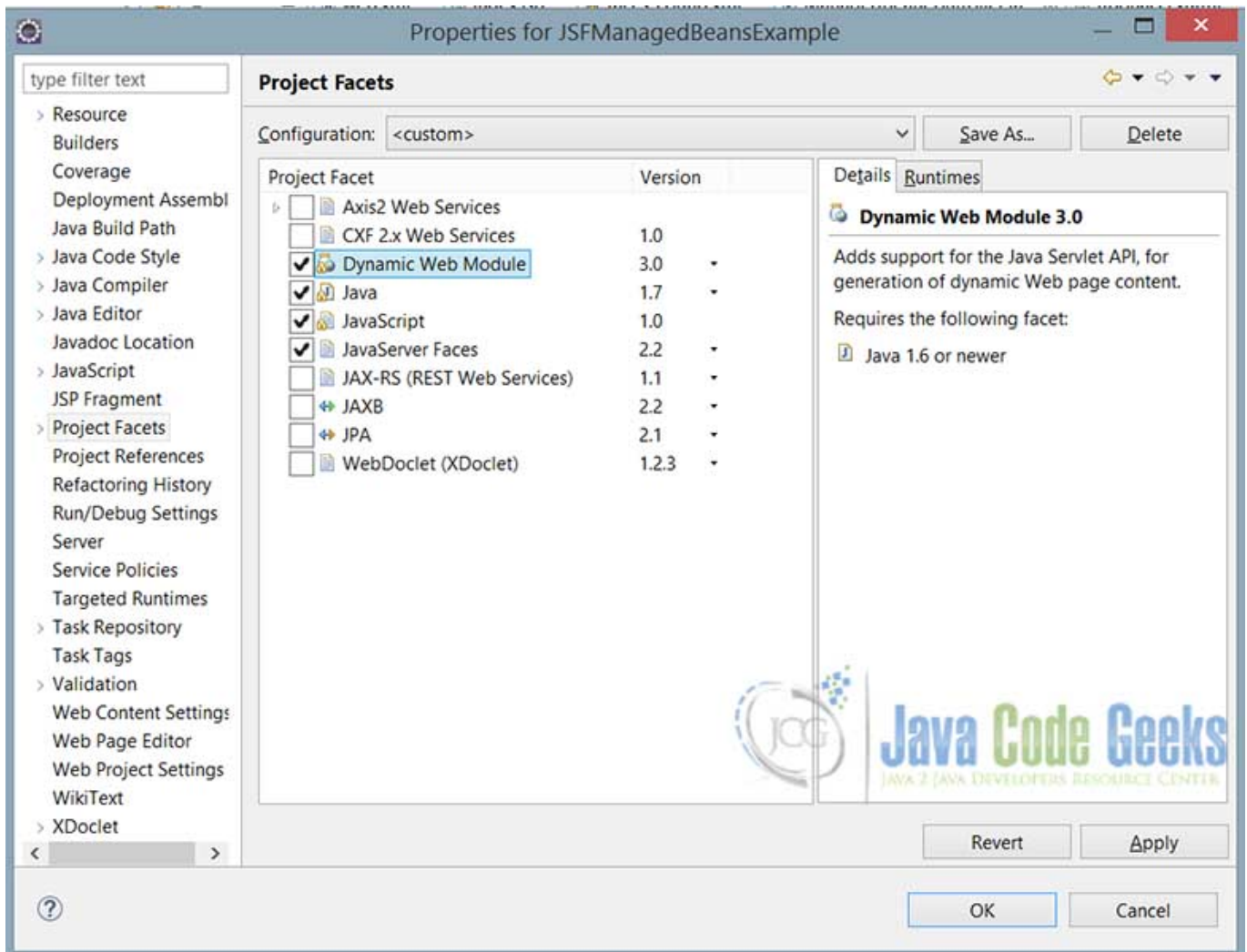


Figure 4.2: JSF Managed Beans Project Facets

### 4.5.3 Configuration Files

In the example we develop a Fibonacci Series Calculator, that based on the number that the user enters we get the value in the F-Series.

Let's Get Started!

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/ ↵
  javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">
  <display-name>JSFManagedBeansExample</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <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>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>
    <context-param>
        <description>State saving method: 'client' or 'server' (=default). See JSF ↔
            Specification 2.5.2</description>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>client</param-value>
    </context-param>
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <context-param>
        <param-name>javax.servlet.jsp.jstl.fmt.localizationContext</param-name>
        <param-value>resources.application</param-value>
    </context-param>
    <listener>
        <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
    </listener>
</web-app>

```

The configuration file `web.xml` is the starting point to load all the dependencies of a java web application. Tomcat Server starts looking for this file and process tag by tag of the xml file. We look into the tags:

- `welcome-file`: The first view to render when application starts
- `servlet`: The Faces Servlet it's the *heart* of JSF, and process every request of the application.
- `servlet-mapping`: Define the url patterns that the Faces Servlet is involved.
- `context-param`: Define some JSF parameters
- `listener`: Parse all relevant JavaServer Faces configuration resources, and configure the Reference Implementation runtime environment

#### faces-config.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<faces-config
    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"
    version="2.2">
</faces-config>

```

The `faces-config` file is no needed because everything is automatic configured for the JSF framework based on Java Annotations.

## 4.5.4 JSF Pages

Now, we create the views, index.html (Simple html page) and fibonacci.xhtml (Facelets View).

index.html

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/ ←
  xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<meta http-equiv="REFRESH"
  content="2; url=http://localhost:8080/JSFManagedBeansExample/fibonacci.jsf" />
<title>JSF Managed Beans JCG Example</title>
</head>
<body>
  <p>This is the home page, will we redirected in 2 seconds to the
    Managed Beans Example.
  </p>
</body>
</html>
```

fibonacci.xhtml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/ ←
  xhtml1-strict.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>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
  <title>JSF Managed Beans JCG Example</title>
</h:head>
<h:body>
  <f:view>
    <h2>Get The N Fibonacci Number</h2>

    <h:form id="fibonnaci">
      Number of the Fibonnaci Series:
      <h:inputText id="number"
        value="#{managedBeanController.numFibonacci}" />
      <h:commandButton action="#{managedBeanController. ←
        performFibonnaciOperation}" value="Calculate">
      </h:commandButton>
      <br />
      Result:
      <h:outputText id="result" value="#{managedBeanController.result}" / ←
        >
      <h:messages infoStyle="color: blue" globalOnly="true" />
    </h:form>

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

The first page, it's a simple HTML view, you can combine both technologies (HTML5 + JSF2.2) and develop powerfull applications, this view only redirects to a view that's processed by Faces Servlet, instead of html isn't processed. How do that? in the 7 line, with the code `meta http-equiv="REFRESH" content="2;url=http://localhost:8080/JSFManagedBeansExample/fibonacci.jsf"`.

The second page, it's a facelets view. What are the differences with HTML? The facelets imports a set of tags allocated in the namespaces defined in the 4 and 5 line, with this get a lot of features that the normal HTML doesn't has. Look at the view, line



by line. With the namespaces abbreviated in H and F we can use the component `f:view` that wrap all the GUI, then, a form with the data that will be send to the Managed Bean. The `h:inputText` set the number in the `numFibonacci` variable of `managedBeanController`, the `commandButton` executes the business operation set in the `action` property, and the last `outputText` component shows the result of the operation.

### 4.5.5 Java Managed Bean

#### ManagedBeanController.java

```
package com.javacodegeeks.jsf.beans;

import java.io.Serializable;

import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;
import javax.faces.event.AjaxBehaviorEvent;

/**
 * @author Andres.Cespedes
 * @version 1.0 $Date: 07/02/2015
 * @since 1.7
 */
@ManagedBean(name = "managedBeanController")
@SessionScoped
public class ManagedBeanController implements Serializable {

    private int numFibonacci;
    private long result;

    /**
     * Number for serialization
     */
    private static final long serialVersionUID = 8150756503956053844L;

    /**
     * No-arg constructor
     */
    public ManagedBeanController() {
        numFibonacci = 0;
        result = 0;
    }

    /**
     * @return the numFibonacci
     */
    public int getNumFibonacci() {
        return numFibonacci;
    }

    /**
     * @param numFibonacci
     *         the numFibonacci to set
     */
    public void setNumFibonacci(int numFibonacci) {
        this.numFibonacci = numFibonacci;
    }

    /**
```

```
    * @return the result
    */
    public long getResult() {
        return result;
    }

    /**
     * @param result
     *         the result to set
     */
    public void setResult(long result) {
        this.result = result;
    }

    /**
     * Business Operation to get the Fibonnaci N-number
     */
    * @param param
    * @return
    */
    private long getFibonnaciNumber(int param) {
        if (param == 1 || param == 2) {
            return 1;
        }
        long actual = 1;
        long next = 1;
        long sum = 0;
        for (int i = 2; i < param; i++) {
            sum = next + actual;
            actual = next;
            next = sum;
        }
        return next;
    }

    /**
     * Non ajax perform Fibonacci Operation
     */
    public void performFibonnaciOperation() {
        if (numFibonacci <= 0) {
            setResult(0L);
        } else {
            setResult(getFibonnaciNumber(numFibonacci));
        }
        FacesMessage facesMsg = new FacesMessage(FacesMessage.SEVERITY_INFO,
            "Fibonacci Calculation for the " + numFibonacci
            + " number was: " + result, "Fibonacci ↔
            Calculation");
        FacesContext.getCurrentInstance().addMessage(null, facesMsg);
    }

    /**
     * Ajax perform Fibonacci Operation
     */
    * @param event
    */
    public void ajaxPerformFibonnaciOperation(final AjaxBehaviorEvent event) {
        if (numFibonacci <= 0) {
            setResult(0L);
        } else {
            setResult(getFibonnaciNumber(numFibonacci));
        }
    }
}
```

```

        FacesMessage facesMsg = new FacesMessage(FacesMessage.SEVERITY_INFO,
            "Fibonacci Calculation for the " + numFibonacci
            + " number was: " + result, "Fibonacci ↵
            Calculation");
        FacesContext.getCurrentInstance().addMessage(null, facesMsg);
    }
}

```

The example is very simple, so we will only review the most important aspects. Every managed bean must be registered with the `@ManagedBean` annotation, we explicitly define a name for the bean, but by default it's the same name with the first letter in lowercase. The scope, you can give a scoped life-time to the `ManagedBean` depending on your need, may be of any types, here we use `@SessionScoped` to do the bean lives as long as the HTTP session lives. The `ajaxPerformFibonacciOperation` has a `AjaxBehaviorEvent` parameter, that is mandatory for every ajax method in jsf.

To invoke an ajax method in the form, we change the `commandButton` definition to:

```

<h:commandButton value="Calculate">
    <f:ajax execute="@form" render="@all"
        listener="#{managedBeanController. ↵
        ajaxPerformFibonacciOperation}" />
</h:commandButton>

```

There are 3 important attributes to look, `execute`, to execute the form; `render`, to update all the components, and `listener` to execute the ajax operation.

#### 4.5.6 Running the Example

Let's make the project run on the Tomcat Server.

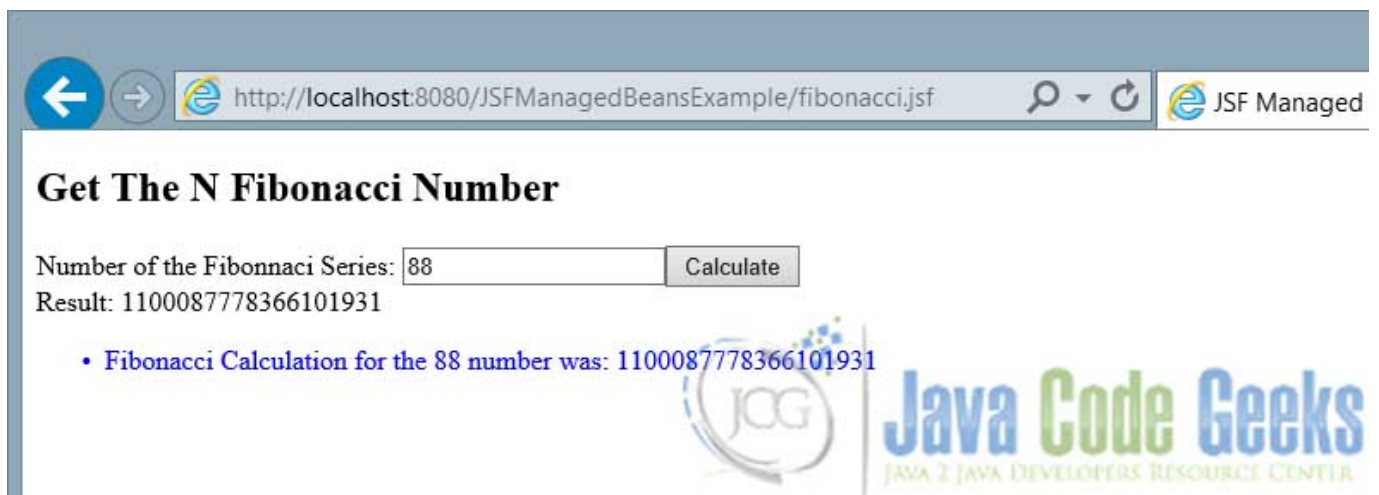


Figure 4.3: JSF Managed Beans Result

#### 4.5.7 Key Points

##### Tips

- Check the compatibility between JSF and every tool that has been used
- Only register each `ManagedBean` with XML descriptor or Annotations, but never with both!

- Check the JSF properties and customize your application with them, through the `context-param` tag.
- Always wrap all the GUI components with `view` element.
- ManagedBeans names are case sensitive, check the names carefully.

## 4.6 Download the Eclipse Project

**Download** You can download the full source code of this example here: [JSFManagedBeansExample](#)

---

## Chapter 5

# TextBox Example

In this example, we are going to demonstrate a simple application, whose purpose is to transfer data inserted into a page's textbox (in our case, a sample username), to another page. While on JSF, we can use the following tag, in order to render an HTML input of a textbox: `<h:inputText/>`.

To get the meaning, imagine that the fore-mentioned XHTML's tag is equal to HTML's `<input type="text">`. So, let's get into the full example.

### 5.1 Managed Bean

Here is our simple Managed Bean, which handles the username.

*UserBean.java*

```
package com.javacodegeeks.enterprise.jsf.textbox;

import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class UserBean implements Serializable{

    private String username;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

}
```

### 5.2 Our Pages

Like we said, we need two separate pages; the first will get the user's input and the second will render it back. Let's have a look at them:

*index.xhtml*

---

```
<?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://java.sun.com/jsf/html">

  <h:head>
    <title>JSF TextBox Example</title>
  </h:head>
  <h:body>
    <h1>JSF 2.0 TextBox Example</h1>

    <h:form>
      <h:inputText value="#{userBean.username}" />
      <h:commandButton value="Submit" action="response" />
    </h:form>
  </h:body>
</html>
```

*response.xhtml*

```
<?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://java.sun.com/jsf/html">

  <h:head>
    <title>JSF TextBox Example</title>
  </h:head>
  <h:body>
    <h1>JSF 2.0 TextBox Example - Response Page</h1>

    Welcome, <h:outputText value="#{userBean.username}" /> !
  </h:body>
</html>
```

## 5.3 Demo

Now that we got our two view pages set up, let's have a quick demo, by trying to access the following URL: `http://localhost:8080/TextboxJSF`



Figure 5.1: textboxJSF1

And after clicking the button, our response page:



Figure 5.2: textboxJSF2

This was an example of TextBox in JSF 2.0. You can also download the source code of this example: [TextboxJSF](#)

## Chapter 6

# Password Example

If you didn't notice from the previous [example](#), we started a mini example series for the **JSF Tag Library**, so in the next couple examples, we are going to deal with several simple, but quite useful JSF tags. Today, we'll deal with a password field. While on JSF, we can use the following tag, in order to render an HTML input of a password field: `<h:inputSecret>`

To get the meaning, imagine that the fore-mentioned xhtml's tag is equal to HTML's `<input type="password">`. So, let's get into the full example.

### 6.1 Managed Bean

Here is our simple Managed Bean, which handles the password.

*UserBean.java*

```
package com.javacodegeeks.enterprise.jsf.password;

import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class UserBean implements Serializable{

    private String password;

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

### 6.2 Our Pages

As in the previous example, we need two separate pages; Let's have a look at them:

*index.xhtml*

---



```
<?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://java.sun.com/jsf/html">

  <h:head>
    <title>JSF Password Example</title>
  </h:head>
  <h:body>
    <h1>JSF 2.0 Password Example</h1>

    <h:form>
      Password : <h:inputSecret value="#{userBean.password}" />
      <h:commandButton value="Submit" action="response" />
    </h:form>
  </h:body>
</html>[source,xml]
```

*response.xhtml*

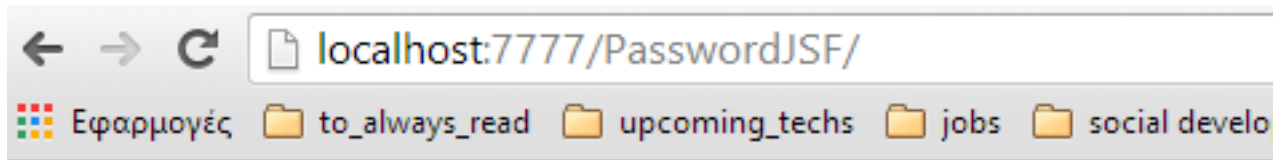
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

  <h:head>
    <title>JSF Password Example</title>
  </h:head>
  <h:body>
    <h1>JSF 2.0 Password Example - Response Page</h1>

    The password is : <h:outputText value="#{userBean.password}" />
  </h:body>
</html>
```

## 6.3 Demo

Let's take a quick demo, by trying to access the following URL: <http://localhost:8080/PasswordJSF>

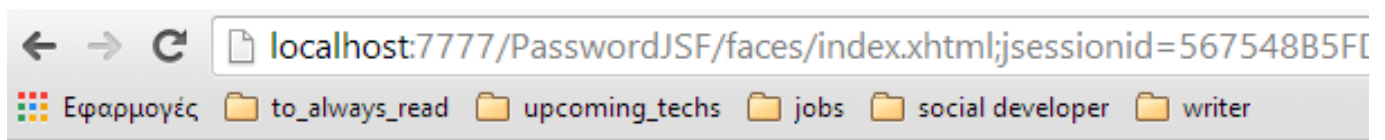


## JSF 2.0 Password Example



Figure 6.1: image

And after clicking the button, our response page:



## JSF 2.0 Password Example - Response Page

The password is : thodoris



Figure 6.2: image2

This was an example of TextBox in JSF 2.0. You can also download the source code for this example: [PaswordJSF](#)

## Chapter 7

# TextArea Example

In this example of <B>JSF Tag Library</b> series, we are going to show an effective way to implement a textarea field. Suppose that we want to insert a textarea of 20 columns and 10 rows. In HTML, this means `<textarea cols="20" rows="10"></textarea>`. According to JSF, we can use the following tag, to implement it: `<h:inputTextarea cols="20" rows="10" />`.

So, let's get the work done!

### 7.1 Managed Bean

As usually, I'll first provide the source code for the Managed Bean, but for this time, we'll hack it a little bit, by changing its name property, to `user`. That is, our web pages, can refer to the Managed Bean by using the custom names that we had given to them, during development.

*UserBean.java*

```
package com.javacodegeeks.enterprise.jsf.textarea;

import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name="user")
@SessionScoped
public class UserBean implements Serializable{

    private String address;

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

### 7.2 Our Pages

As in the previous example, we need two separate pages; Let's have a look at them:

*index.xhtml*

---

```
<?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://java.sun.com/jsf/html">

  <h:head>
    <title>JSF Textarea Example</title>
  </h:head>
  <h:body>
    <h1>JSF 2.0 Textarea Example</h1>
    <h:form>
      Address:<h:inputTextarea value="#{user.address}" cols="20" rows="10" />
      <h:commandButton value="Submit" action="response" />
    </h:form>
  </h:body>
</html>
```

#### *response.xhtml*

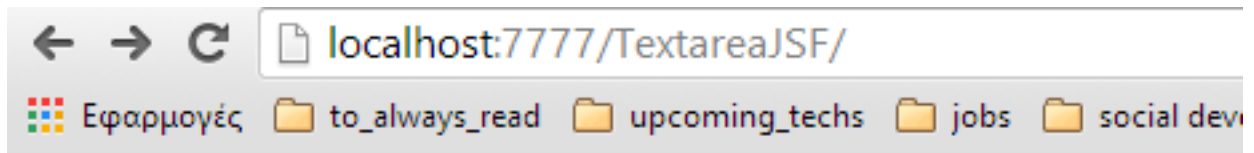
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

  <h:head>
    <title>JSF Textarea Example</title>
  </h:head>
  <h:body>
    <h1>JSF 2.0 Textarea Example - Response Page</h1>

    Your address is : <h:outputText value="#{user.address}" />
  </h:body>
</html>
```

## 7.3 Demo

Let's take a quick demo, by trying to access the following URL: <http://localhost:8080/TextareaJSF>



## JSF 2.0 Textarea Example

Address:

Submit



Figure 7.1: image

And after clicking the button, our response page:



## JSF 2.0 Textarea Example - Response Page

Your address is : 1 CITY HALL SQ, RM 615



Figure 7.2: image2

## 7.4 Some Closing Words

There is a chance to get an execution error; you will be informed that the class `user` cannot be found. In this case, we usually try to handle the Managed Bean's name annotation by hand, by simply adding a managed bean dependency in `faces-config.xml` file. For this reason, I'm attaching this version of the project, just in order to keep every newbie to J2EE following along, without facing any difficulty.

This was an example of Textarea in JSF 2.0. You can also download the source code for this example: [TextareaJSF](#)

## Chapter 8

# CheckBox Example

Hi there, hope you had an interesting day. Today we 're gonna talk about checkboxes in JSF 2.0. To represent a checkbox in JSF, we use the tag `h:selectBooleanCheckbox`. Ok, that's really easy, but what if we 'd like to create a group of checkboxes, where the user could select more than one checkbox? This can be done using the `h:selectManyCheckbox` tag; the HTML renderings are exactly the same, as we saw in my previous [example](#). To be more specific, before getting into the example's structure, here is a small example that implements a group of three checkboxes, where the user can select more than one of them:

```
<h:selectManyCheckbox value="#{user.favNumber1}">
  <f:selectItem itemValue="1" itemLabel="Number1 - 1" />
  <f:selectItem itemValue="2" itemLabel="Number1 - 2" />
  <f:selectItem itemValue="3" itemLabel="Number1 - 3" />
</h:selectManyCheckbox>
```

Ok, enough said, let's have a quick example with that type of checkboxes, but I have to first notice the four different ways that we can populate a group of checkboxes:

- Hardcoded value in a `f:selectItem` tag.
- Generated values from an Array and passed into the fore-mentioned tag.
- Generated values using a Map and passed into the same tag.
- Generate values using an Object Array and passed again into the `f:selectItem` tag, then represent the value using a `var` attribute.

### 8.1 Backing Bean

Here's the structure of the Bean that holds the submitted values.

*UserBean.java*

```
package com.javacodegeeks.enterprise.jsf.checkboxes;

import java.io.Serializable;
import java.util.Arrays;
import java.util.LinkedHashMap;
import java.util.Map;

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

@ManagedBean
@SessionScoped
```

```
public class UserBean implements Serializable{

    private static final long serialVersionUID = -3953324291794510390L;

    public boolean rememberMe;
    public String[] favoriteCar1;
    public String[] favoriteCar2;
    public String[] favoriteCar3;
    public String[] favoriteCar4;
    public boolean isRememberMe() {
        return rememberMe;
    }
    public void setRememberMe(boolean rememberMe) {
        this.rememberMe = rememberMe;
    }
    public String[] getFavoriteCar1() {
        return favoriteCar1;
    }
    public void setFavoriteCar1(String[] favoriteCar1) {
        this.favoriteCar1 = favoriteCar1;
    }
    public String[] getFavoriteCar2() {
        return favoriteCar2;
    }
    public void setFavoriteCar2(String[] favoriteCar2) {
        this.favoriteCar2 = favoriteCar2;
    }
    public String[] getFavoriteCar3() {
        return favoriteCar3;
    }
    public void setFavoriteCar3(String[] favoriteCar3) {
        this.favoriteCar3 = favoriteCar3;
    }
    public String[] getFavoriteCar4() {
        return favoriteCar4;
    }
    public void setFavoriteCar4(String[] favoriteCar4) {
        this.favoriteCar4 = favoriteCar4;
    }

    public String getFavoriteCar1InString()
    {
        return Arrays.toString(favoriteCar1);
    }

    //generated by Array
    public String[] getFavoriteCar2Value()
    {
        favoriteCar2 = new String [5];
        favoriteCar2[0] = "116";
        favoriteCar2[1] = "118";
        favoriteCar2[2] = "X1";
        favoriteCar2[3] = "Series 1 Coupe";
        favoriteCar2[4] = "120";

        return favoriteCar2;
    }

    public String getFavoriteCar2InString()
    {
        return Arrays.toString(favoriteCar2);
    }
}
```



```
}

//generated by Map
private static Map<String, Object> car3Value;
static
{
    car3Value = new LinkedHashMap<String, Object>();
    car3Value.put("Car3 - 316", "BMW 316");
    car3Value.put("Car3 - 318", "BMW 318");
    car3Value.put("Car3 - 320", "BMW 320");
    car3Value.put("Car3 - 325", "BMW 325");
    car3Value.put("Car3 - 330", "BMW 330");
}

public Map<String, Object> getFavoriteCar3Value()
{
    return car3Value;
}
public String getFavoriteCar3InString() {
    return Arrays.toString(favoriteCar3);
}

//generated by Object Array
public static class Car
{
    public String carLabel;
    public String carValue;

    public Car(String carLabel, String carValue)
    {
        this.carLabel = carLabel;
        this.carValue = carValue;
    }

    public String getCarLabel()
    {
        return carLabel;
    }

    public String getCarValue()
    {
        return carValue;
    }
}

public Car[] car4List;
public Car[] getFavoriteCar4Value()
{
    car4List = new Car[5];

    car4List[0] = new Car("Car 4 - M3", "BMW M3 SMG");
    car4List[1] = new Car("Car 4 - X3", "BMW X3");
    car4List[2] = new Car("Car 4 - X5", "BMW X5");
    car4List[3] = new Car("Car 4 - X6", "BMW X6");
    car4List[4] = new Car("Car 4 - 745", "BMW 745");

    return car4List;
}

public String getFavoriteCar4InString()
{
    return Arrays.toString(favoriteCar4);
}
```

```

    }
}

```

## 8.2 Our JSF Pages

First, the welcome page, where we have a single checkbox and the four afore-mentioned different ways, which populate group checkboxes.

*index.xhtml*

```

<?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://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core" >

  <h:body>
    <h1>JSF 2.2 CheckBoxes Example</h1>
    <h:form>
      <h2>1. Single checkbox</h2>
      <h:selectBooleanCheckbox value="#{user.rememberMe}" />Remember Me

      <h2>2. Group of checkboxes</h2>
      1. Hardcoded using the tag "f:selectItem" :
      <h:selectManyCheckbox value="#{user.favoriteCar1}">
        <f:selectItem itemLabel="Car1 - E10" itemValue="BMW E10" />
        <f:selectItem itemLabel="Car1 - E36" itemValue="BMW E36" />
        <f:selectItem itemLabel="Car1 - E46" itemValue="BMW E46" />
        <f:selectItem itemLabel="Car1 - E87" itemValue="BMW E87" />
        <f:selectItem itemLabel="Car1 - E92" itemValue="BMW E92" />
      </h:selectManyCheckbox>

      <br/>

      2. Generated by Array :
      <h:selectManyCheckbox value="#{user.favoriteCar2}">
        <f:selectItems value="#{user.favoriteCar2Value}" />
      </h:selectManyCheckbox>

      <br/>

      3. Generated by Map :
      <h:selectManyCheckbox value="#{user.favoriteCar3}">
        <f:selectItems value="#{user.favoriteCar3Value}" />
      </h:selectManyCheckbox>

      <br/>

      4. Generated by Object, displayed using var
      <h:selectManyCheckbox value="#{user.favoriteCar4}">
        <f:selectItems value="#{user.favoriteCar4Value}" var="last"
          itemLabel="#{last.carLabel}" itemValue="#{last.carValue}" />
        </f:selectItems>
      </h:selectManyCheckbox>

      <br/>

      <h:commandButton value="Submit" action="results" />
      <h:commandButton value="Reset" type="reset" />

```

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

Then, just to ensure that every submitted value saved correctly, we'll try to access the related getters through a JSF page:

*results.xhtml*

```
<?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://java.sun.com/jsf/html">

    <h:body>
        <h1>JSF 2.2 CheckBoxes Example - Response Page</h1>

        <ol>
            <li>user.rememberMe : #{user.rememberMe}</li>
            <li>user.favoriteCar1 : #{user.favoriteCar1InString}</li>
            <li>user.favoriteCar2 : #{user.favoriteCar2InString}</li>
            <li>user.favoriteCar3 : #{user.favoriteCar3InString}</li>
            <li>user.favoriteCar4 : #{user.favoriteCar4InString}</li>
        </ol>
    </h:body>
</html>
```

## 8.3 Demo

I'll just select my favorites from each group:

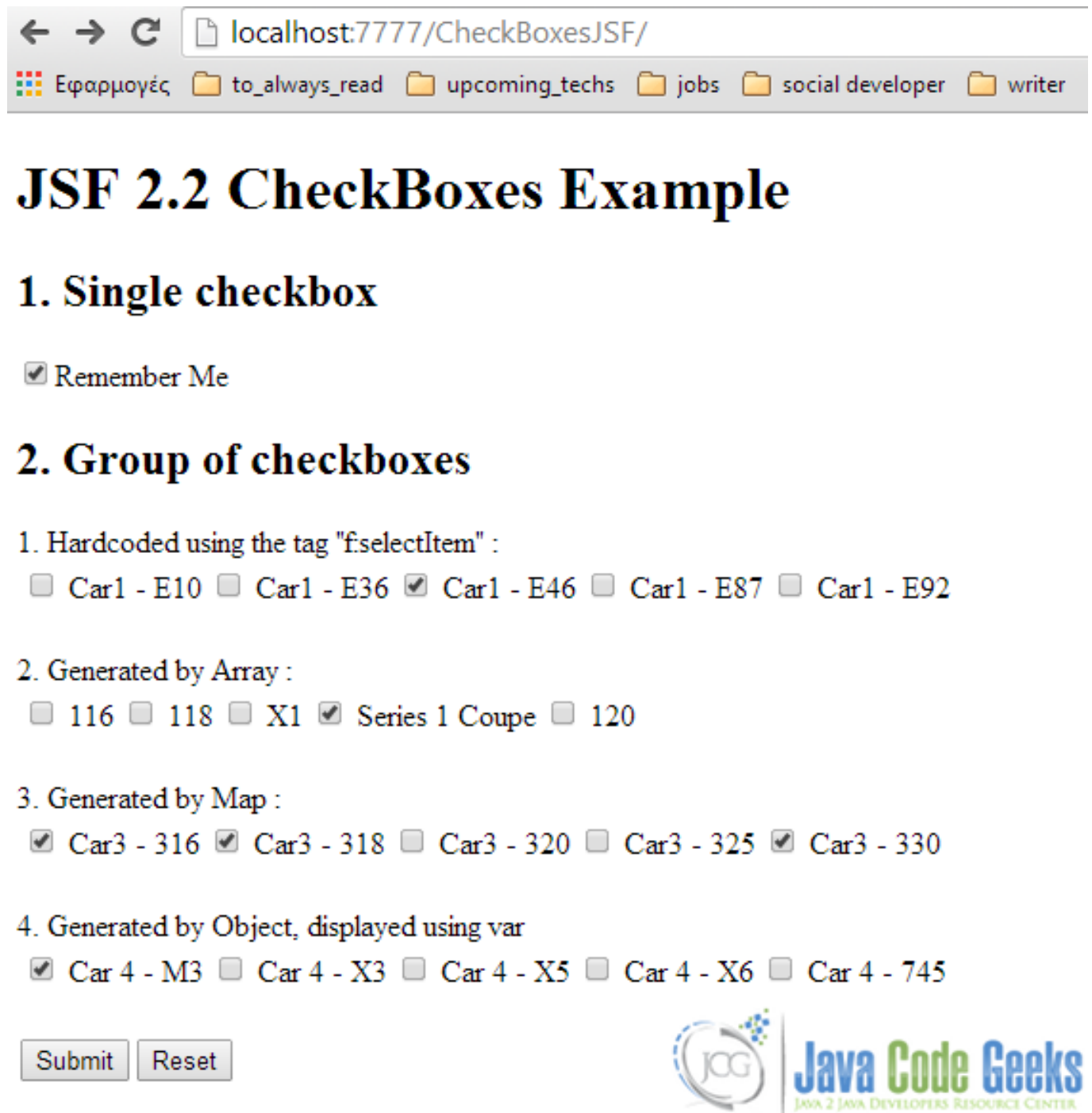
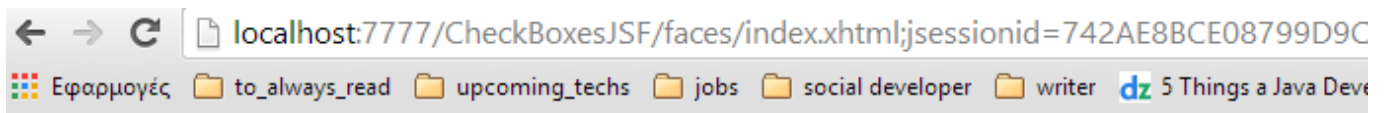


Figure 8.1: img

Let's see what happened:



## JSF 2.2 CheckBoxes Example - Response Page

1. user.rememberMe : true
2. user.favoriteCar1 : [BMW E46]
3. user.favoriteCar2 : [Series 1 Coupe]
4. user.favoriteCar3 : [BMW 316, BMW 318, BMW 330]
5. user.favoriteCar4 : [BMW M3 SMG]



Figure 8.2: img2

This was an example of CheckBoxes in JSF 2.0. You can also download the source code for this example: [CheckBoxesJSF](#)

## Chapter 9

# OutputText Example

Hi there, pretty short time since my last example! I'll firstly try to give a short explanation about the connection of my [last example](#) with this. So, let's give it a try!

The reason that we're now won't have a full example of **multiple selectable dropdown list**, as we used to in the last few examples, is that the nested elements are not displayed consistently in different browsers.

We could use the `<h:selectManyMenu />` to render a multiple selectable dropdown list, but just take a look of how this JSF element could be rendered across [Internet Explorer](#), [Mozilla Firefox](#) and [Google Chrome](#). That is, this case is one of the worst nightmares of a developer, so please avoid using it.

Back to this example and according to output text in JSF, I'll try to get you deep into the meaning of it, by showing a multiple case example.

The useable tag is obvious as you might thought, just give a tag of `<h:outputText />` to render an output text element.

### 9.1 Managed Bean

A demonstration bean to contain two sample strings:

*UserBean.java*

```
package com.javacodegeeks.enterprise.jsf;

import java.io.Serializable;
import java.util.Arrays;
import java.util.LinkedHashMap;
import java.util.Map;

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

@ManagedBean
@SessionScoped
public class UserBean implements Serializable{

    private static final long serialVersionUID = 4256272866128337548L;

    public String text = "Hello Java Code Geeks!" ;
    public String htmlInput = "<input type='text' size='20' /> " ;

    public String getText() {
        return text;
    }
    public void setText(String text) {
```

```

        this.text = text;
    }
    public String getHtmlInput() {
        return htmlInput;
    }
    public void setHtmlInput(String htmlInput) {
        this.htmlInput = htmlInput;
    }
}

```

## 9.2 Our JSF Page

*index.xhtml*

```

<?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://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core" >

    <h:body>
        <h1>JSF 2.2 OutputText Example</h1>
        <ol>
            <li>#{user.text}</li>
            <li><h:outputText value="#{user.text}" /></li>
            <li><h:outputText value="#{user.text}" styleClass="sampleClass" /></li>
            <li><h:outputText value="#{user.htmlInput}" /></li>
            <li><h:outputText value="#{user.htmlInput}" escape="false" /></li>
        </ol>
    </h:body>
</html>

```

And if you didn't make it so clear, this is what is being generated to HTML:

```

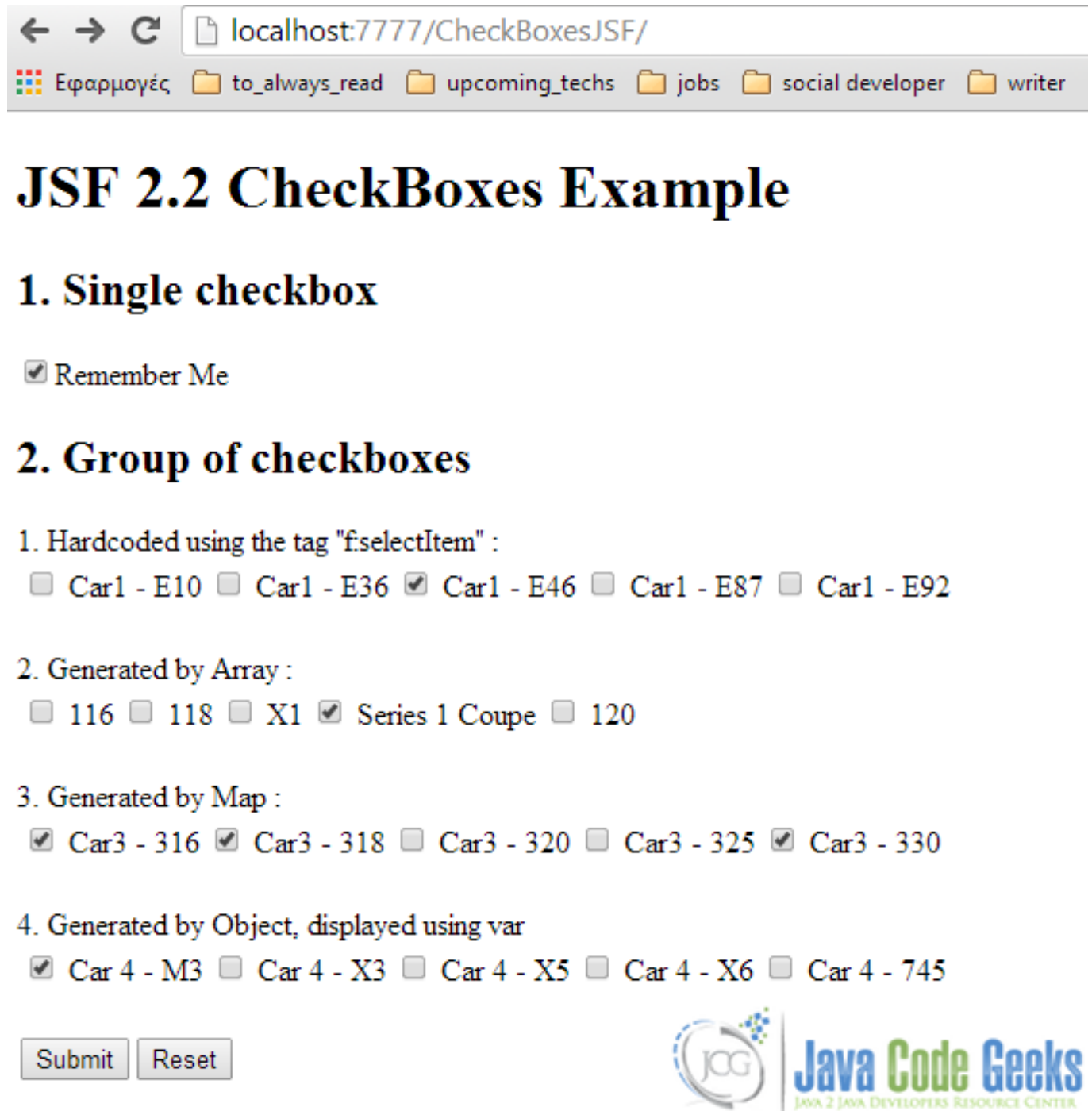
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <body>
        <h1>JSF 2.2 OutputText Example</h1>
        <ol>
            <li>Hello Java Code Geeks!</li>
            <li>Hello Java Code Geeks!</li>
            <li><span class="sampleClass">Hello Java Code Geeks!</span></li>
            <li><input type='text' size='20' /></li>
            <li><input type='text' size='20' /></li>
        </ol>
    </body>
</html>

```

And a little bit of more analysis:

- Cases 1 and 2: we don't have to really use the `h:outputText` tag, since this can be achieved using the value expression `"#{user.text}"`.
- Case 3: if we have any tag of kind `styleClass`, `style`, `dir` or `lang`, just render the text and wrap it around a `span` element.
- Case 4 and 5: we use the `escape` attribute in `h:outputText` tag in order to convert sensitive HTML and XML markup to the corresponding valid HTML characters (i.e. "<" is converted to "&lt;"); `escape` attribute is set to true by default.

## 9.3 Demo



← → ↻ localhost:7777/CheckBoxesJSF/

Εφαρμογές to\_always\_read upcoming\_techs jobs social developer writer

# JSF 2.2 CheckBoxes Example

## 1. Single checkbox

Remember Me

## 2. Group of checkboxes

1. Hardcoded using the tag "fselectItem" :  
 Car1 - E10  Car1 - E36  Car1 - E46  Car1 - E87  Car1 - E92
2. Generated by Array :  
 116  118  X1  Series 1 Coupe  120
3. Generated by Map :  
 Car3 - 316  Car3 - 318  Car3 - 320  Car3 - 325  Car3 - 330
4. Generated by Object, displayed using var  
 Car 4 - M3  Car 4 - X3  Car 4 - X5  Car 4 - X6  Car 4 - 745


 Java Code Geeks  
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Figure 9.1: img

This was an example of ListBox in JSF 2.0. You can also download the source code for this example: [OutputTextJSF](#)



## Chapter 10

# Button and CommandButton Example

Hello Java code geeks! Today we 're gonna take a look at navigation handling, using buttons. Regarding JSF 2.0, both `<h:button />` and `<h:commandButton />` are used to render HTML input element mechanisms that can guide the navigation through a web application.

### 10.1 h:commandButton tag

*We first have to declare the bean, which returns the navigation's outcome in the `action` attribute. This navigation is handled through a form post, so it will also work in a browser with disabled Javascript.*

Let's have a look at a sample *submit* button:

```
<h:commandButton value="Submit" type="submit" action="#{user.login}" />
```

And here is what is generated to HTML:

```
<input type="submit" name="xxx" value="Submit" />
```

Pretty cool, huh?! Ok, let's see a *reset* button now:

```
//JSF
<h:commandButton value="Reset" type="reset" />

//HTML output
<input type="reset" name="xxx" value="Reset" />
```

Regarding basic understanding efforts, let's investigate the difference between a normal button and a same one, with an event listener attached.

*Normal button*

```
//JSF
<h:commandButton type="button" value="Click Me!" />

//HTML output
<input type="button" name="xxx" value="Click Me!" />
```

*Normal button with onClick event listener*

```
//JSF
<h:commandButton type="button" value="Click Me!" onclick="alert('Hi from commandButton!');" ←
/>
```

```
//HTML output
<input type="button" name="xxx" value="Click Me!" onclick="alert('Hi from commandButton!'); ←
" />
```

## 10.2 h:button tag

While on JSF 2.0, we can use the latest button tags, such as `<h:commandButton>`, so *generally*, there is no need for the deprecated one, `<h:commandButton>`, which is available, since JSF 1.x. And if you're still wondering about last sentence's *italics* format, this comes from our tag's attributes:

- We can have a direct navigation declaration, without the need to call a bean first.
- If our browser has disabled its Javascript feature, navigation may fail, as the `<h:button>` tag generates an `onclick` event listener, that handles the navigation through `window.location.href`.

Just to ensure that noone has questionmarks, here are some very basic examples, using the fore-mentioned tag:

*Normal button - no outcome*

```
//JSF
<h:button value="Click Me!" />

//HTML output
<input type="button"
  onclick="window.location.href='/JavaServerFaces/faces/index.xhtml; return false;"
  value="Click Me!" />
```

*Normal button - with outcome*

```
//JSF
<h:button value="Click Me!" outcome="login" />

//HTML output
<input type="button"
  onclick="window.location.href='/JavaServerFaces/faces/login.xhtml; return false;"
  value="Click Me!" />
```

*Normal button - with Javascript*

```
//JSF
<h:button value="Click Me!" onclick="alert('Hi from button!');" />

//HTML output
<input type="button"
  onclick="alert('Hi from button!');window.location.href='/JavaServerFaces/faces/ ←
  sample_page.xhtml;return false;"
  value="Click Me!" />
```

This was an example of Button and CommandButton in JSF 2.0.

## Chapter 11

# PanelGrid Example

Today we 're gonna talk about table formatting in JSF.

Remember yourself on your very first steps to Web Development, how easy was it to manipulate a table with raw HTML? You had to follow a strict writing with a bunch of specific HTML tags, in order to get what you wanted to.

JSF is here to make our lives easier, in table manipulation, too, so let's take a look at a comparative initial example, between HTML and JSF:

### HTML

```
<h:inputText id="number" value="#{sample_bean.number}"
              size="20" required="true"
              label="Number" >
  <f:convertNumber />
</h:inputText>
```

Piece of cake, but still a waste of time. That's the corresponding format of the above table in **JSF**:

```
<h:panelGrid columns="3">
  Please enter a number:

  <h:inputText id="number" value="#{sample_bean.number}"
              size="20" required="true"
              label="Number" >
    <f:convertNumber />
  </h:inputText>

  <h:message for="number" style="color:red" />
</h:panelGrid>
```

*Ok, pretty easier with JSF, but what's the functionality of <f:convertNumber /> ?*

Generally, the fore-mentioned tag creates a number formatting converter and associates it with the nearest parent UIComponent; for more information, please refer [here](#).

The final goal of this tutorial, is to transfer the number inserted from the user, to another page, after validating that it actually was a number (and not a string, for example). So, we 're here using the <f:convertNumber /> for **validation** purpose.

### 11.1 The Example

Here is a demonstration JSF app, using panelGrid for the elements' proper layout.

## 11.2 The Managed Bean

```
package com.javacodegeeks.jsf.panelgrid;

import java.io.Serializable;

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

@ManagedBean(name="sample_bean")
@SessionScoped
public class SampleBean implements Serializable{

    int number;

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }
}
```

## 11.3 The JSF Pages

*index.xhtml*

```
<?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://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:c="http://java.sun.com/jsp/jstl/core" >

    <h:head>
        <title>JSF PanelGrid</title>
    </h:head>

    <h:body>
        <h1>JSF 2.2 PanelGrid Example</h1>

        <h:form>
            <h:panelGrid columns="3">
                Please enter a number:
                <h:inputText id="number"
                    value="#{sample_bean.number}"
                    size="15" required="true"
                    label="Number" >
                    <f:convertNumber />
                </h:inputText>
                <h:message for="number" style="color:red" />
            </h:panelGrid>
            <h:commandButton value="Submit" action="result"/>
        </h:form>
    </h:body>
</html>
```

result.xhtml

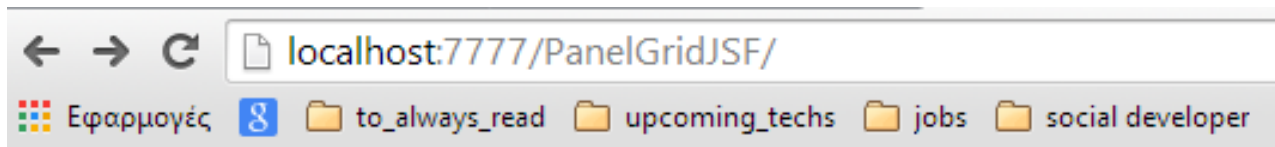
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      >

  <h:head>
    <title>JSF PanelGrid</title>
  </h:head>

  <h:body>
    <h1>JSF 2.2 PanelGrid Example</h1>
    Number : <h:outputText value="#{sample_bean.number}" />
  </h:body>
</html>
```

## 11.4 The Demo

Let's first insert a string, just to see our app's exception handling; we'll then insert a number (75 in our case), to conform with the success path.



# JSF 2.2 PanelGrid Example

Please enter a number:

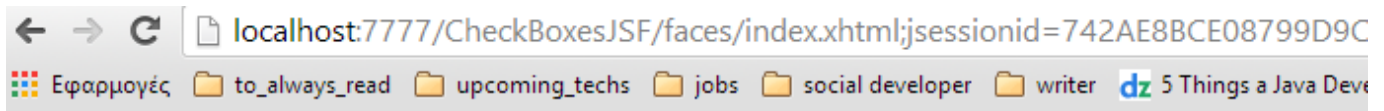
Submit

↑                    ↑                    ↑

Column 1          Column 2          Column 3 - Errors



Figure 11.1: img

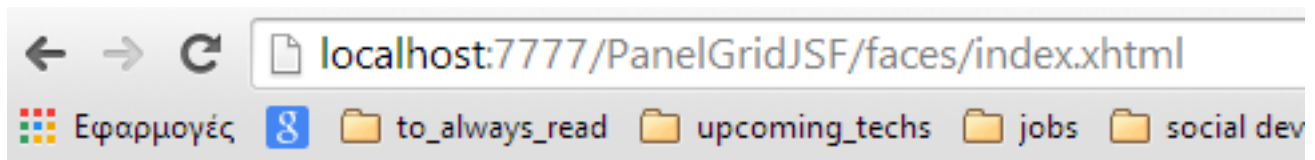


## JSF 2.2 CheckBoxes Example - Response Page

1. user.rememberMe : true
2. user.favoriteCar1 : [BMW E46]
3. user.favoriteCar2 : [Series 1 Coupe]
4. user.favoriteCar3 : [BMW 316, BMW 318, BMW 330]
5. user.favoriteCar4 : [BMW M3 SMG]



Figure 11.2: img2



## JSF 2.2 PanelGrid Example

Number : 75



Figure 11.3: img3

This was an example of PanelGrid in JSF 2.0. You can also download the source code for this example: [PanelGridJSF](#)

## Chapter 12

# Message and Messages Example

Hi there, today we 'll see how to display special messages (i.e. for validation purpose) in JSF.

In JSF, we can use the following two tags to render a message:

- `<h:message>` : displays a single message for a specific component.
- `<h:messages>` : displays all messages in the current page.

Here is a good example, demonstrating a page with form validation, to get a better understanding of these tags:

*Please keep in mind that this example will be tested using our [last JSF project's](#) structure, so there isn't a simple reason to upload the same project again, including only two changes, but if, there is any problem from your side, please refer to this [repository](#) and search for my latest commit, named as "necessary changes to sync with Messages example"*

`default.xhtml`

```
<?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://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:c="http://java.sun.com/jsp/jstl/core" >

  <h:body>
    <h1>JSF 2.2 PanelGrid Example</h1>

    <h:form>
      <h:messages style="color:red;margin:8px;" />
      <br/>
      <h:panelGrid columns="3">

        Enter your username :

        <h:inputText id="username" value="#{sample_bean.username}"
          size="20" required="true"
          label="UserName" >
          <f:validateLength minimum="4" maximum="12" />
        </h:inputText>

        <h:message for="username" style="color:red" />

        Enter your age :
        <h:inputText id="age" value="#{sample_bean.age}"
          size="20" required="true"
```

```
                label="Age" >
                <f:validateLongRange for="age" minimum="1" maximum="115" />
            </h:inputText>
            <h:message for="age" style="color:red" />
        </h:panelGrid>
        <h:commandButton value="Submit" action="result" />
    </h:form>
</h:body>
</html>
```

*SampleBean.java*

```
package com.javacodegeeks.jsf.panelgrid;

import java.io.Serializable;

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

@ManagedBean(name="sample_bean")
@SessionScoped
public class SampleBean implements Serializable{

    int number;
    private String username;
    private int age;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public int getAge() {
        return age;
    }

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

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }
}
```

## 12.1 The Demo

*Invalid username, age*



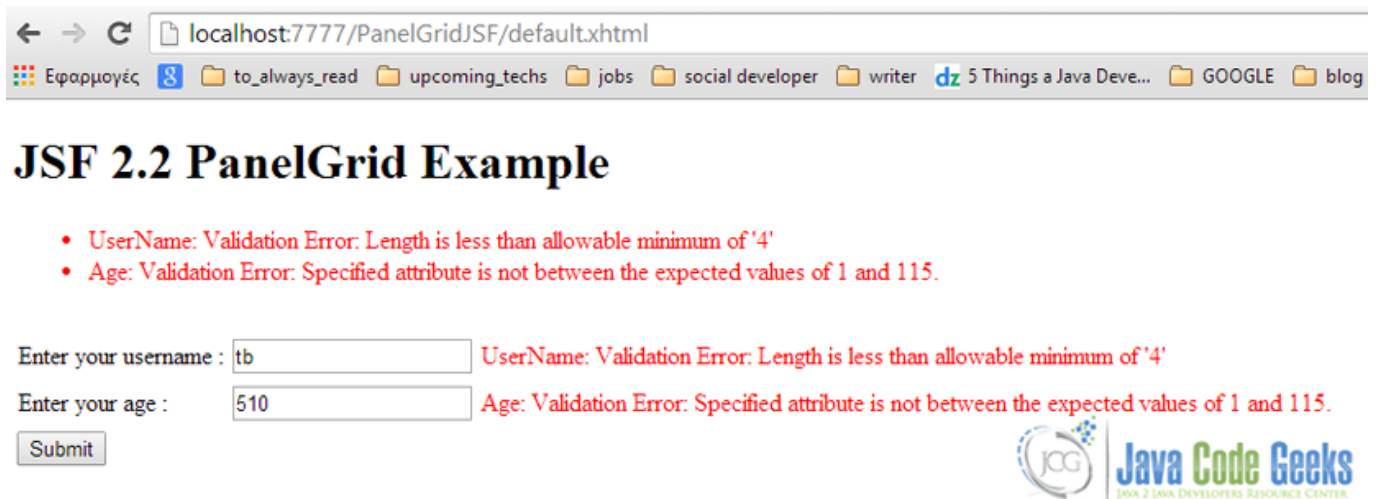


Figure 12.1: img1

*Invalid age*

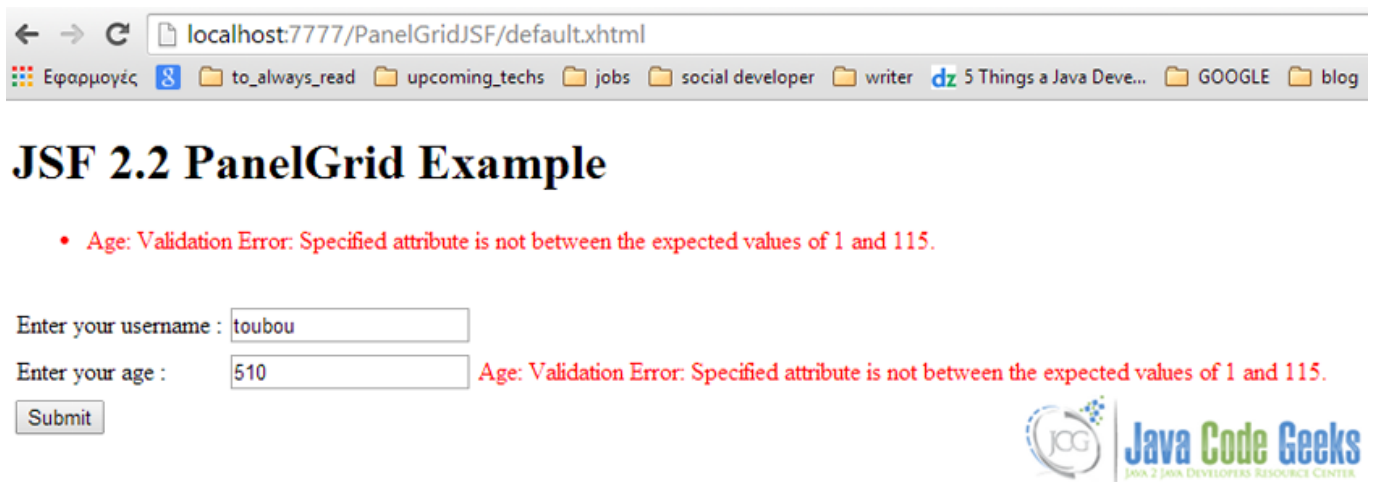


Figure 12.2: img22

*Invalid username*



Figure 12.3: img3

*Valid username, age*

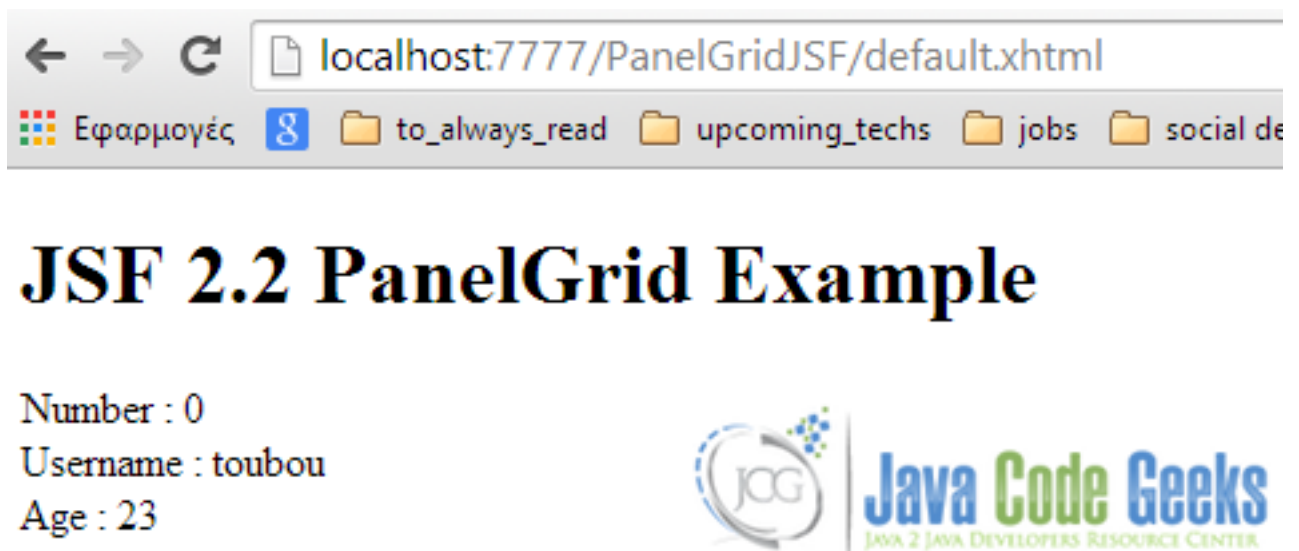


Figure 12.4: img4

This was an example of Message and Messages in JSF 2.0.

## Chapter 13

# Param and Attribute Example

Today we're gonna talk about parameter manipulation in JSF, using `param` and `attribute` tags.

### 13.1 Param Tag

What about parameters in JSF? In JSF, we can use the `<f:param>` tag, in order to pass a parameter to a component (or pass request parameters), but things here, are not so clear, as it's behavior depends on the component that it is attached.

The current example's goal is, to pass two parameters from a JSF page, to another. Let's see some introductory examples, before diving into the full example.

*Passing parameters to UI components*

```
<h:outputFormat value="Hello {0}. It seems like you support {1} in this World Cup.">
    <f:param value="Thodoris" />
    <f:param value="Greece" />
</h:outputFormat>
```

*Passing request parameters (attached to commandButton)*

```
<h:commandButton id="submit"
    value="Submit" action="#{user.outcome}">
    <f:param name="country" value="Greece" />
</h:commandButton>
```

*But how could we get these values in the back-end? How could we conform our usual bean to read them from the JSF page?*

```
Map<String,String> params = FacesContext.getExternalContext().getRequestParameterMap();
String country = params.get("country");
```

#### 13.1.1 The Full Example

Suppose that our super-clever web application has a page that prompts the user to insert his name (see *Param Tag - Prompt Page*); the application then, transfers his name to a welcome page, where a guess is made, also. In our case, it's about the user's favourite team in the World Cup 2014. Both parameters will be manipulated through the `param` tag.

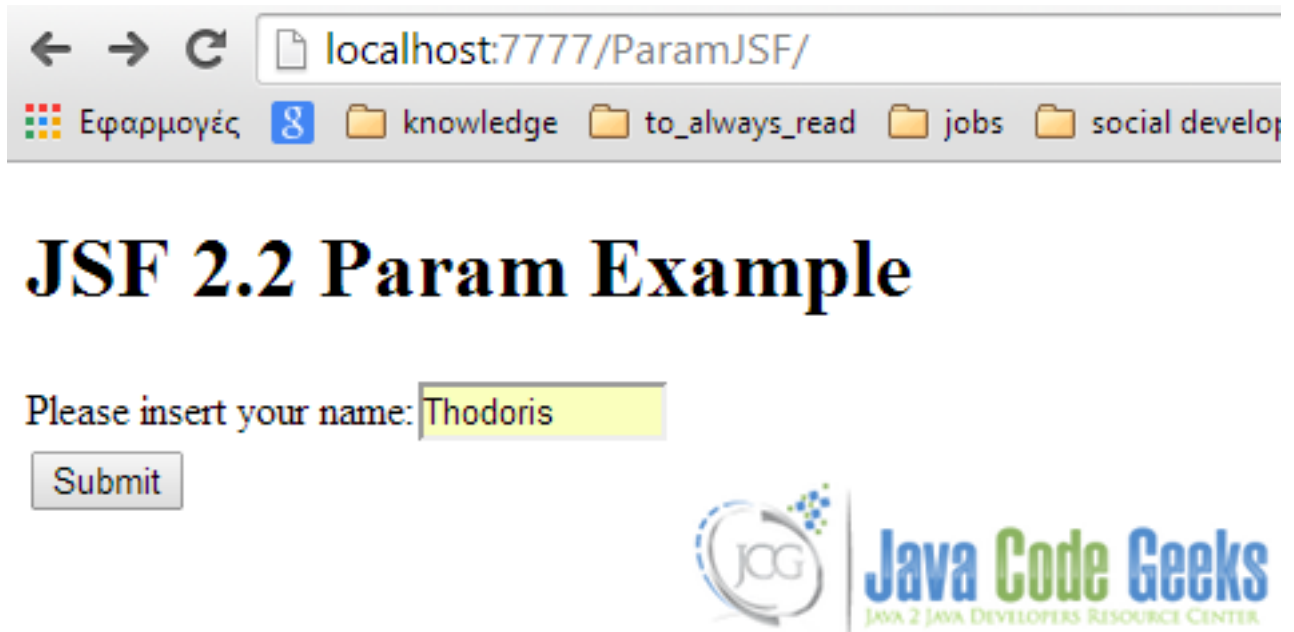


Figure 13.1: Param Tag - Prompt Page

So, regarding the technical part of our example:

- The fore-mentioned parameter manipulation will take place in the java bean
- The "index" page will contain one submit button, which will also send a parameter to the "welcome" page
- The "welcome" page will welcome the user in personal level, using the name that he provided in the "index" page; it will also display the parameter transferred from the button's click.

Here is the full example, demonstrating the usage for both of the two fore-mentioned mini-examples.

We'll first take a look at our usual UserBean class:

```
package com.javacodegeeks.jsf.param;

import java.io.Serializable;
import java.util.Map;

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

@ManagedBean(name="user")
@SessionScoped
public class UserBean implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    private String name;
    private String country;

    private String getCountryFromJSF(FacesContext context) {
        Map<String, String> parameters = context.getExternalContext().
            getRequestParameterMap();
```

```

        return parameters.get("country");
    }

    public String outcome() {
        FacesContext context = FacesContext.getCurrentInstance();
        this.country = getCountryFromJSF(context);

        return "result";
    }

    public String getName() {
        return name;
    }

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

    public String getCountry() {
        return country;
    }

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

```

Now, let's see the code structure of our JSF pages that interact with the "backing-bean" class:

*index.xhtml*

```

<?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://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:c="http://java.sun.com/jsp/jstl/core">

<h:head>
    <title>JSF Param Example</title>
</h:head>
<h:body>
    <h1>JSF 2.2 Param Example</h1>
    <h:form id="simpleform">
        Please insert your name:<h:inputText size="10"
            value="#{user.name}" />
        <br />
        <h:commandButton id="submit" value="Submit" action="#{user.outcome}">
            <f:param name="country" value="Greece" />
        </h:commandButton>
    </h:form>
</h:body>
</html>

```

*result.xhtml*

```

<?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://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:c="http://java.sun.com/jsp/jstl/core">

<h:head>
  <title>JSF Param Example</title>
</h:head>
<h:body>
  <h1>JSF 2.2 Param Example - Result</h1>
  <h:outputFormat
    value="Hello {0}.
    It seems like you support {1} in this
    World Cup.">
    <f:param value="#{user.name}" />
    <f:param value="#{user.country}" />
  </h:outputFormat>
</h:body>
</html>

```

## 13.2 Attribute Tag

The purpose and usage of this tag are somehow similar to the `param` tag, but here, we have to deal with an *actionListener*. According to the tag's definition, *it provides an option to pass an attribute's value to a component, or a parameter to a component via action listener.*

What a better example to be experimented with the newly introduced tag, rather than a simple button? Yes, that is, let's keep it simple by investigating the tag's behavior through a button (a simple button is enough to fire the *actionListener*).

*So, assuming that we have a web page with a button, we want this button to "carry" some parameters, when it is clicked (especially, a password, in our case).*

In order to get familiar with the `attribute`, keep in mind that we can also use it to assign a value to our button:

```

<h:commandButton>
  <f:attribute name="value" value="Submit" />
</h:commandButton>

// Similar to:
<h:commandButton value="Submit" />

```

Now, let's see how we can assign and retrieve a parameter from a component: *(The concept is exactly the same with the above example.*

```

<h:commandButton actionListener="#{user.actionListenerSampleMethod}" >
  <f:attribute name="password" value="test" />
</h:commandButton>

```

Yes, that was our button, "carrying" a password value. As described above (*and -I'm sure- made clear by yourself*), except from our results page (where we just want to display the password's value, so there isn't anything more complicated than `#{user.password}`), we have to implement the "interaction" method. That is, this method is written in the class that connects the two pages, in the back-end, as we already know from all the previous examples. Here it is:

```

package com.javacodegeeks.jsf.attribute;

import java.io.Serializable;

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

```

```
@ManagedBean(name="user")
@SessionScoped
public class UserBean implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    private String password;

    // The action listener method.
    public void actionListenerSampleMethod(ActionEvent event) {
        password = (String)event.getComponent().getAttributes().get("password");
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

This was an example of Param and Attribute in JSF 2.0.

## Chapter 14

# Standard Validators Example

Hey geeks, today we 're gonna talk about [JSF Standard Validators](#).

Just like any other framework, JSF is here to help us save time from common development tasks, such as form validations. We can obviously write our own, custom validators for our site's forms, but there are some standard validators provided from JSF that handle checks for string lengths and numeric ranges.

To begin with, suppose a sample form, where the user is prompted to submit his username, age and salary.

### 14.1 Project Environment

This example was implemented using the following tools:

- JSF 2.2
- Maven 3.1
- Eclipse 4.3 (Kepler)
- JDK 1.7
- Apache Tomcat 7.0.41

*Just like any other of my previous JSF examples, you need to create a Dynamic Web Project with Maven and JSF should be included in it. At any case, if you don't remember some configurations, consult my very first [example](#) according to JSF.*

This is the project's final structure, just to ensure that you won't get lost anytime.

---



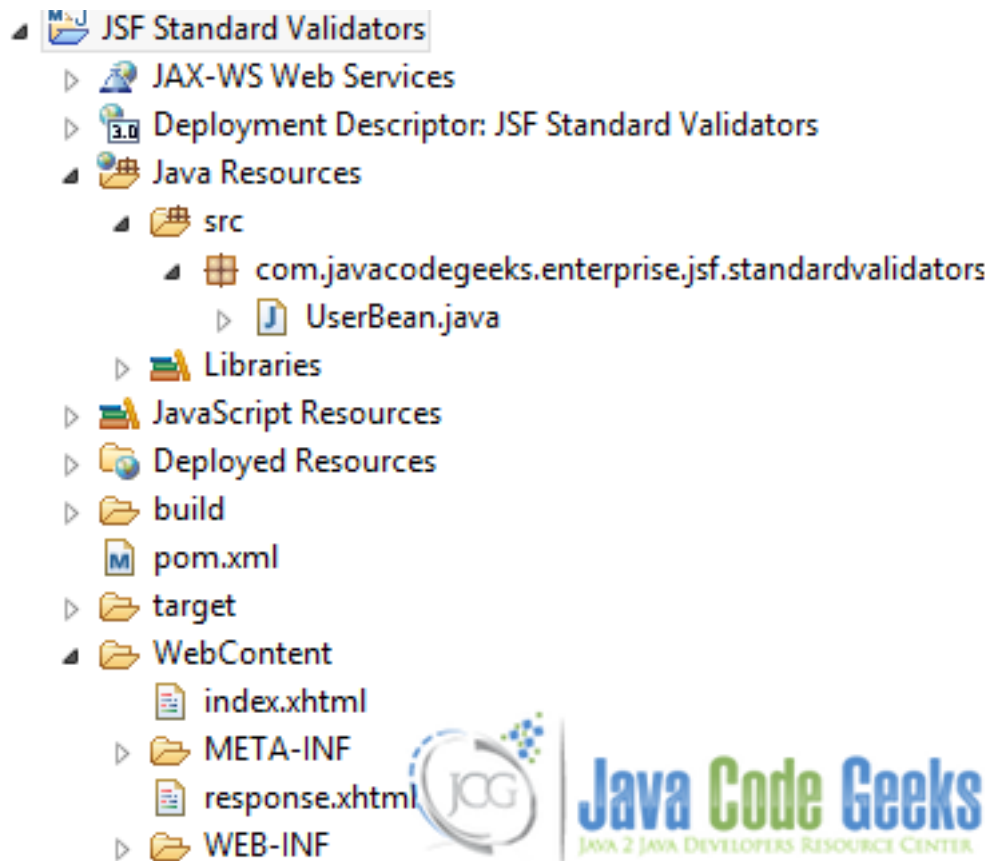


Figure 14.1: Project Structure

## 14.2 JSF Pages

As I fore-mentioned, we 'll here use three form fields: username, age and salary. A typical validation process could be:

- Username should be 5-12 (five to twelve) characters long.
- Age should be between 12 and 110 years.
- Salary should be between 653.90 and 3500.1.

We have different validation criteria in each one of our form fields, so each validation has to be done inside the `h:inputText` tag, according to our constraints.

*index.xhtml*

```
<?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://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core" >

  <h:head>
    <title>JSF Standard Validators Example</title>
  </h:head>
  <h:body>
    <h1>JSF 2.0 Standard Validators Example</h1>
```

```

    <h:form>
      <h:panelGrid columns="3">
        Username:
        <h:inputText id="username" value="#{user.username}"
          size="15" required="true" label="Username">
          <f:validateLength minimum="5" maximum="12" />
        </h:inputText>
        <h:message for="username" style="color:red" />

        Age:
        <h:inputText id="age" value="#{user.age}"
          size="2" required="true" label="Age">
          <f:validateLongRange minimum="12" maximum="110" />
        </h:inputText>
        <h:message for="age" style="color:red" />

        Salary:
        <h:inputText id="salary" value="#{user.salary}"
          size="7" required="true" label="Salary">
          <f:validateDoubleRange minimum="653.90" maximum="
            3500.1" />
        </h:inputText>
        <h:message for="salary" style="color:red" />

      </h:panelGrid>
      <h:commandButton value="Submit" action="response"/>
    </h:form>
  </h:body>
</html>

```

The `f:validateLength` tag is used to verify that a component's length is within a specified range, which in our case, is translated between 5 and 12. It is used for **String** types.

The `f:validateLongRange` tag is used to verify that a component's value is within a specified range, which in our case, is translated between 12 and 110. It supports any numeric type or **String** that can be converted to **long**.

The `f:validateDoubleRange` tag is used to verify that a component's value is within a specified range, which in our case, is translated between 653.90 and 3500.1. It supports any value that can be converted to floating-point type or floating point.

Just for demonstration purposes, we are creating a response page, to show the success scenario of the validation.

*response.xhtml*

```

<?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://java.sun.com/jsf/html">
  <h:head>
    <title>JSF Standard Validators Example</title>
  </h:head>
  <h:body>
    <h1>JSF 2.0 Standard Validators Example - Response Page</h1>
    Your username is : <h:outputText value="#{user.username}" />
    <br/>Your salary is : <h:outputText value="#{user.salary}" />
    <br/>Your age is: <h:outputText value="#{user.age}" />
  </h:body>
</html>

```

## 14.3 Managed Bean

Our JSF pages interact with each other because of a **ManagedBean**, so here it is:

*ManagedBean.java*

```
package com.javacodegeeks.enterprise.jsf.standardvalidators;

import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class UserBean implements Serializable{

    private static final long serialVersionUID = 7134492943336358840L;

    private double salary;
    private String username;
    private int age;

    public int getAge() {
        return age;
    }

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

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
}
[source, java]
```

## 14.4 Demo

As a first example and just to ensure our validators' proper functionality, I am providing invalid data to all fields:

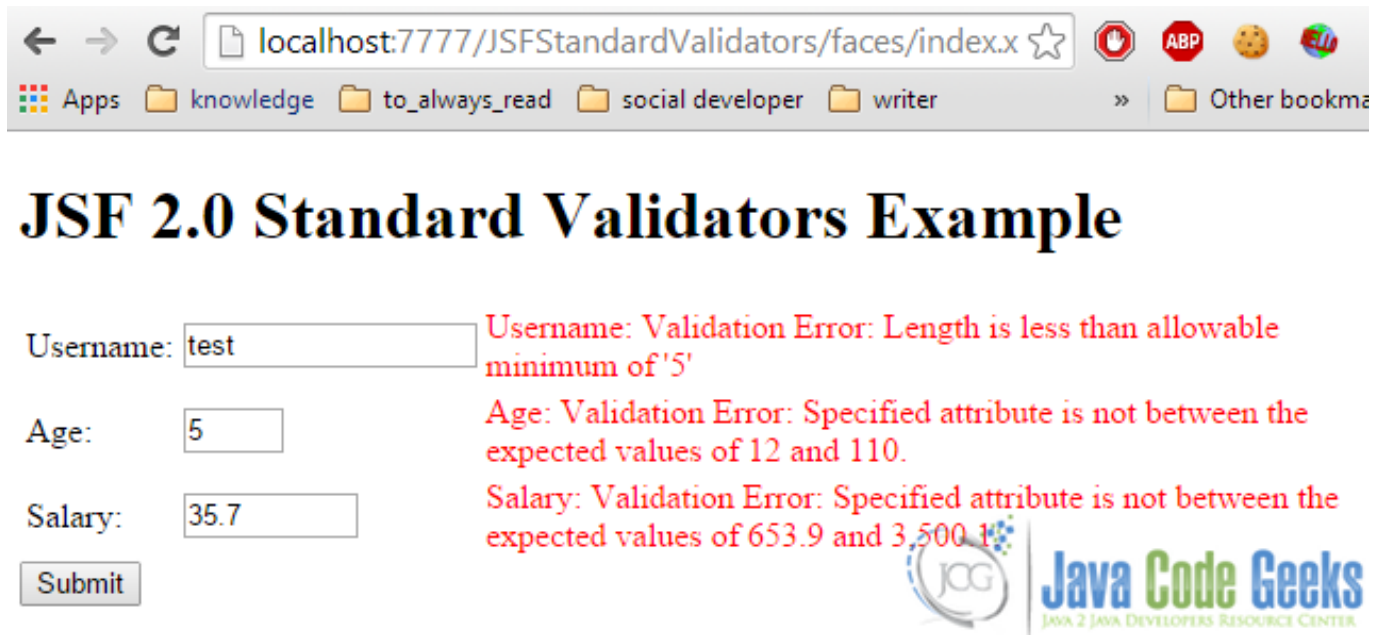


Figure 14.2: Providing invalid data

Well, it seems to work properly, so let's go for some valid values:



Figure 14.3: Providing acceptable values

## 14.5 Download the Eclipse Project

This was an example of JSF Standard Validators.

**Download** You can download the full source code of this example here : [JSFStandardValidators.zip](#)

## Chapter 15

# Internationalization Example

So, you want your web app to be famous all-over the world?! Then you obviously have to support multiple languages, something also known as **i18n** or **internationalization** in the software industry. Today's example is a simple yet interesting guide according to JSF internationalization.

Specifically, our sample application will be able to translate a message between English and Greek (there will be two image buttons, respectively).

### 15.1 Project Environment

This example was implemented using the following tools:

- JSF 2.2
- Maven 3.1
- Eclipse 4.3 (Kepler)
- JDK 1.7
- Apache Tomcat 7.0.41

*Just like any other of my previous JSF examples, you need to create a Dynamic Web Project with Maven and JSF should be included in it. At any case, if you don't remember some configurations, consult my very first [example](#) according to JSF.*

This is the project's final structure, just to ensure that you won't get lost anytime.

---

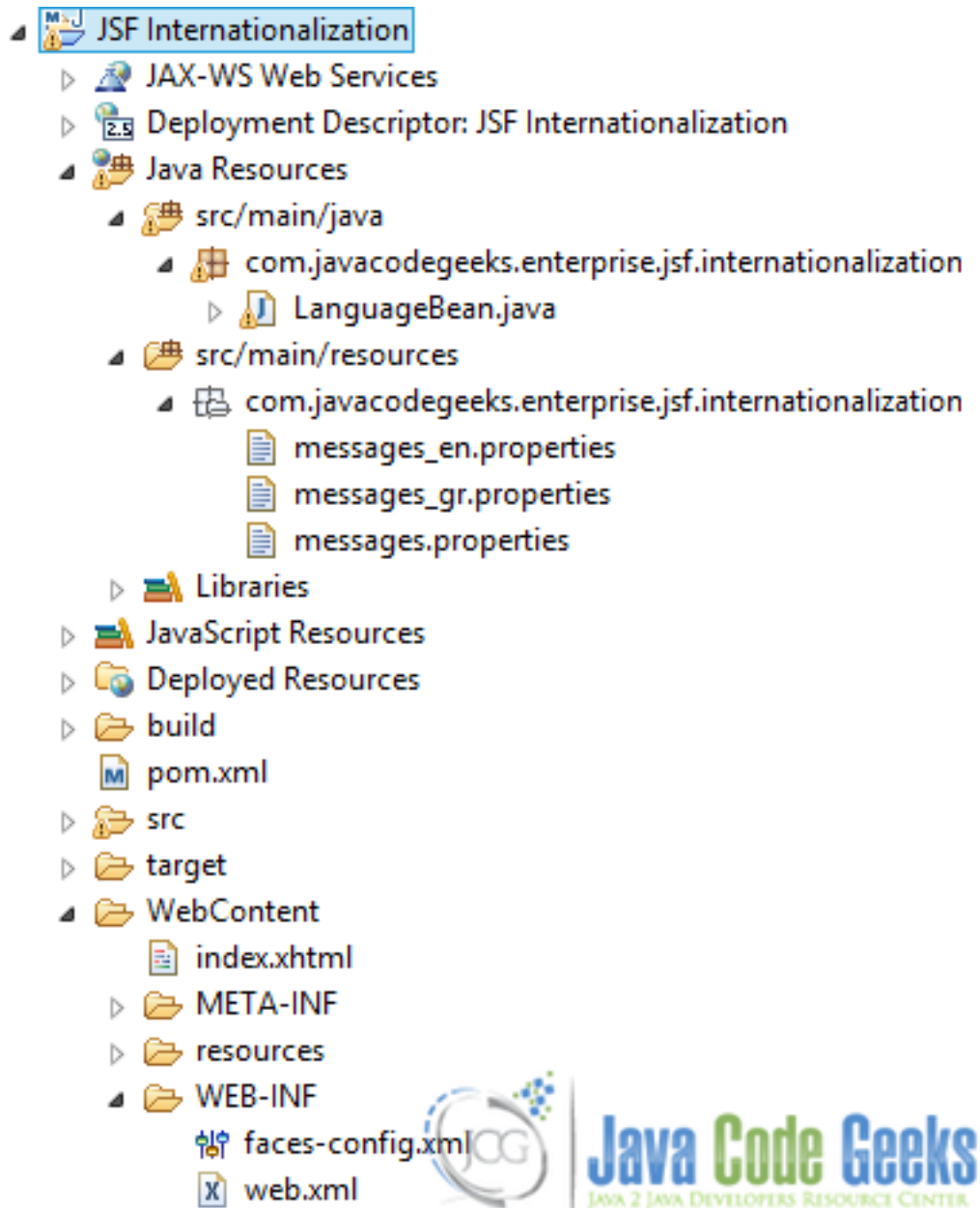


Figure 15.1: Project Structure

## 15.2 Properties files

Let's start with the easiest part of the example, which is creating the necessary translations for our app. Supposing we want to support English and Greek, we should have two properties (translations) files, placed under the `src/main/resources` folder.

```
messages.properties
```

```
motivation = When you do what you fear most, then you can do anything.
```

Greek characters are **UTF-8** and generally, we handle non-english characters in a different way, in JSF; we have to convert them into ascii characters. This can be done with an online [tool](#).

```
messages_gr.properties
```

```
motivation = \u038c\u03c4\u03b1\u03bd \u03ba\u03ac\u03bd\u03b5\u03b9\u03c2 \u03b1\u03c5\ \u03c4  
u03c4\u03cc \u03c0\u03bf\u03c5 \u03c6\u03bf\u03b2\u03ac\u03c3\u03b1\u03b9 \u03c0\u03b5\ \u03c1  
u03c1\u03b9\u03c3\u03c3\u03cc\u03c4\u03b5\u03c1\u03bf, \u03c4\u03c4\u03c4\u03c4\u03b5 \u03bc\ \u03c0  
u03c0\u03bf\u03c1\u03b5\u03af\u03c2 \u03bd\u03b1 \u03ba\u03ac\u03bd\u03b5\u03b9\u03c2 \ \u03c4  
u03c4\u03b1 \u03c0\u03ac\u03bd\u03c4\u03b1.
```

## 15.3 Configuration of faces-config.xml

*faces-config.xml*

```
<?xml version="1.0" encoding="UTF-8"?>  
<faces-config  
  xmlns="http://java.sun.com/xml/ns/javaee"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  
  http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"  
  version="2.0">  
  <application>  
    <locale-config>  
      <default-locale>en</default-locale>  
      <supported-locale>gr</supported-locale>  
    </locale-config>  
    <resource-bundle>  
      <base-name>com.javacodegeeks.enterprise.jsf.internationalization. \u03c4  
      messages</base-name>  
      <var>msg</var>  
    </resource-bundle>  
  </application>  
</faces-config>
```

We'll obviously choose English as default locale. This means that even if the requested from the JSF page locale isn't supported, English will be provided as a locale. In addition, we define as a secondary (supported) locale, Greek.

Our locales depend on the given resource bundle, so, in our case, we tell JSF to search them under `com.javacodegeeks.enterprise.jsf.internationalization.messages` package, match them with the requested from JSF page property file and return them into a variable named `msg` (we'll see how this is feasible in the JSF page analysis section).

## 15.4 Managed Bean

*Language.java*

```
package com.javacodegeeks.enterprise.jsf.internationalization;  
  
import java.io.Serializable;  
import java.util.Locale;  
  
import javax.faces.bean.ManagedBean;  
import javax.faces.bean.SessionScoped;  
import javax.faces.context.FacesContext;  
  
@ManagedBean(name="language")  
@SessionScoped  
public class LanguageBean implements Serializable{  
  
    private static final long serialVersionUID = 1L;  

```

```

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

    public Locale getLocale() {
        return locale;
    }

    public String getLanguage() {
        return locale.getLanguage();
    }

    public void changeLanguage(String language) {
        locale = new Locale(language);
        FacesContext.getCurrentInstance().getViewRoot().setLocale(new Locale(language));
    }
}

```

A usual ManagedBean with a `changeLanguage()` method, which will be used to change app's language, according to the requested from the JSF page locale-parameter.

We initialize the default locale with the help of *FacesContext*.

## 15.5 JSF Page

*index.xhtml*

```

<?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://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
<h:body>
    <h1>JSF 2.2 Internationalization Example</h1>

    <h:form>
        <h:commandButton action="#{language.changeLanguage('')}"
            value="English" image="resources/img/icon/be.png" />
        <h:outputText value="&#160;" />
        <h:commandButton action="#{language.changeLanguage('gr')}"
            value="Greek" image="resources/img/icon/us.png" />
    </h:form>

    <h:outputText value="#{msg['motivation']}" />

</h:body>
</html>

```

We display our message using the fore-mentioned `msg` variable, which searches the value of "motivation" message, in the corresponding translation file, according to the clicked image button.

For demonstration purposes, we call the `changeLanguage()` method with an empty argument, just to show that the default locale will be used, instead of a possible exception.

## 15.6 Demo

Hitting the url of our newly created project, results to the default locale:





Figure 15.2: Default locale

If we click the Greek flag image button, we instantly see the message translated:



Figure 15.3: Greek locale

## 15.7 Download the Eclipse Project

This was an example of JSF Internationalization.

**Download** You can download the full source code of this example here : [JSFInternationalization.zip](#)

## Chapter 16

# Facelets Templates Example

In this example of JSF Facelets Templates, we will show you how to use JSF Facelets to create templates in JSF. To provide better user experience web pages in the applications uses similar layout for all pages. In JSF we can use templates to create standard web layouts.

We will also discuss about JSF Facelets tags that are used to create the Template. Let's begin with setting up a JSF project and do all the necessary configuration to run the application.

Our preferred environment is Eclipse. We are using Eclipse Luna SR1 with Maven Integration Plugin, JDK 8u25 (1.8.0\_25) and Tomcat 8 application server. Having said that, we have tested the code against JDK 1.7 and Tomcat 7 as well.

### 16.1 Create a new Maven Project

Go to File → New → Other → Maven Project

---

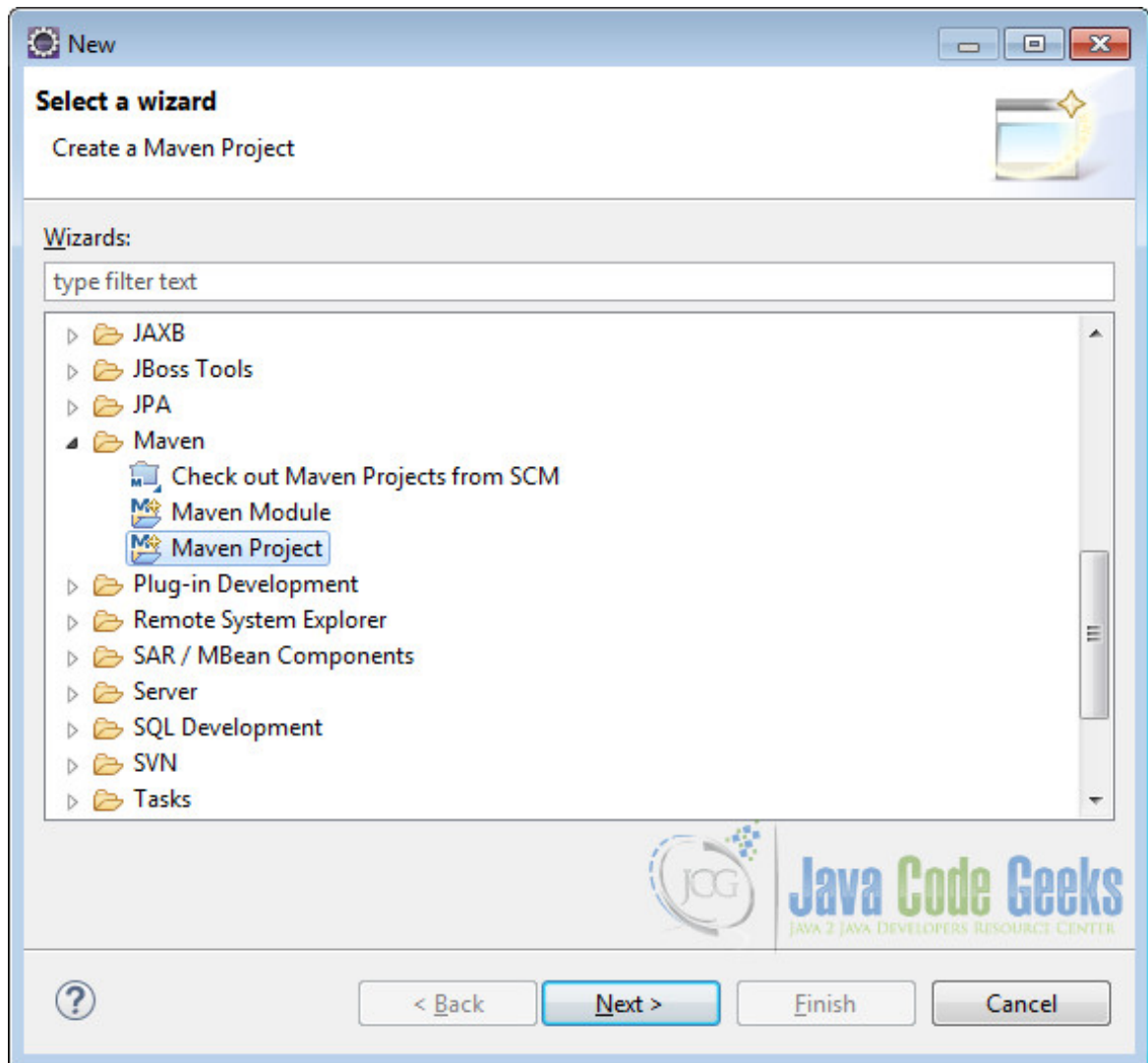


Figure 16.1: Maven Setup - Step 1

In the “Select project name and location” page of the wizard, make sure that “Create a simple project (skip archetype selection)” option is **unchecked**, hit “Next” to continue with default values.

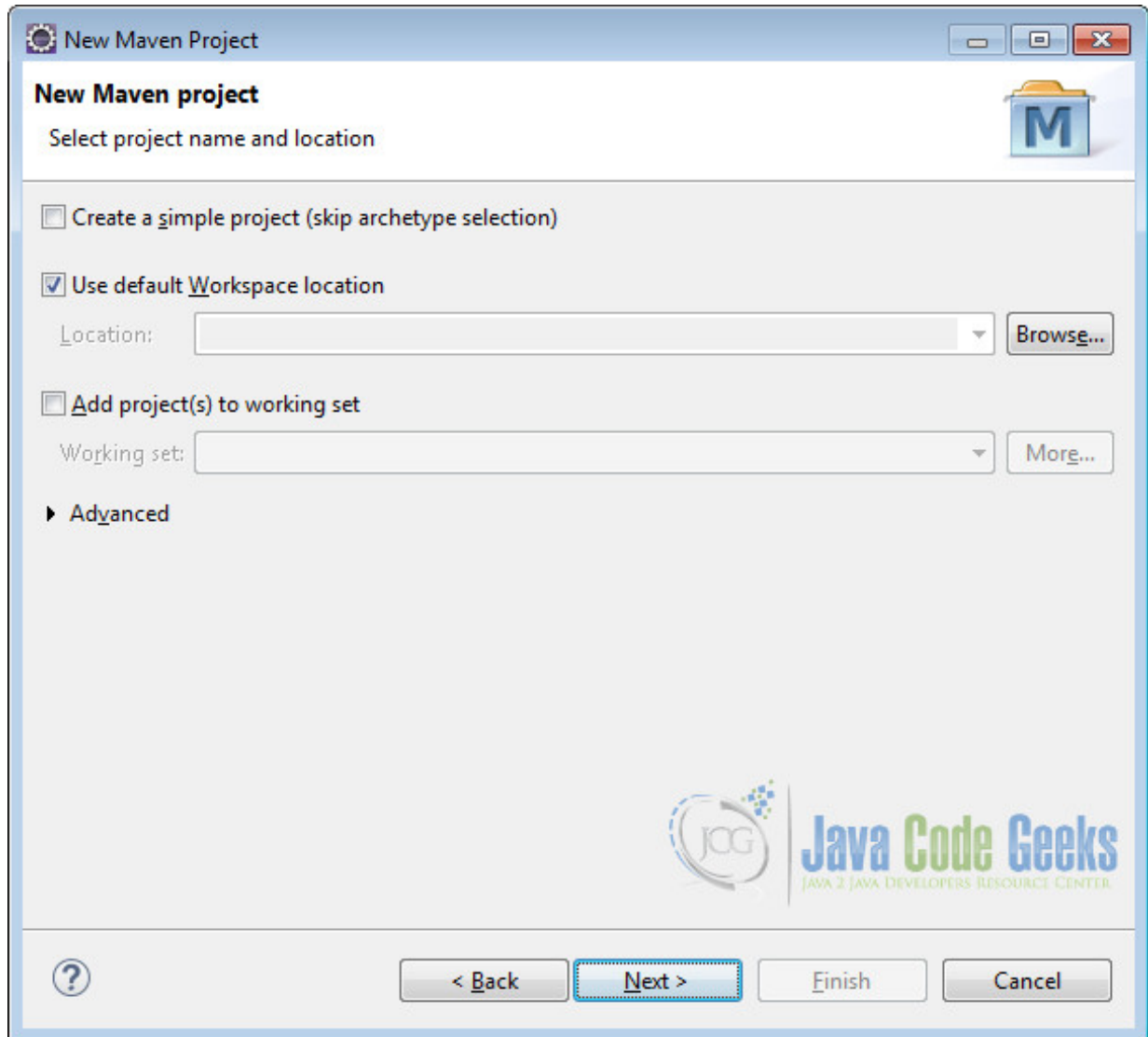


Figure 16.2: Maven setup - step 2

Here choose “maven-archetype-webapp” and click on Next.

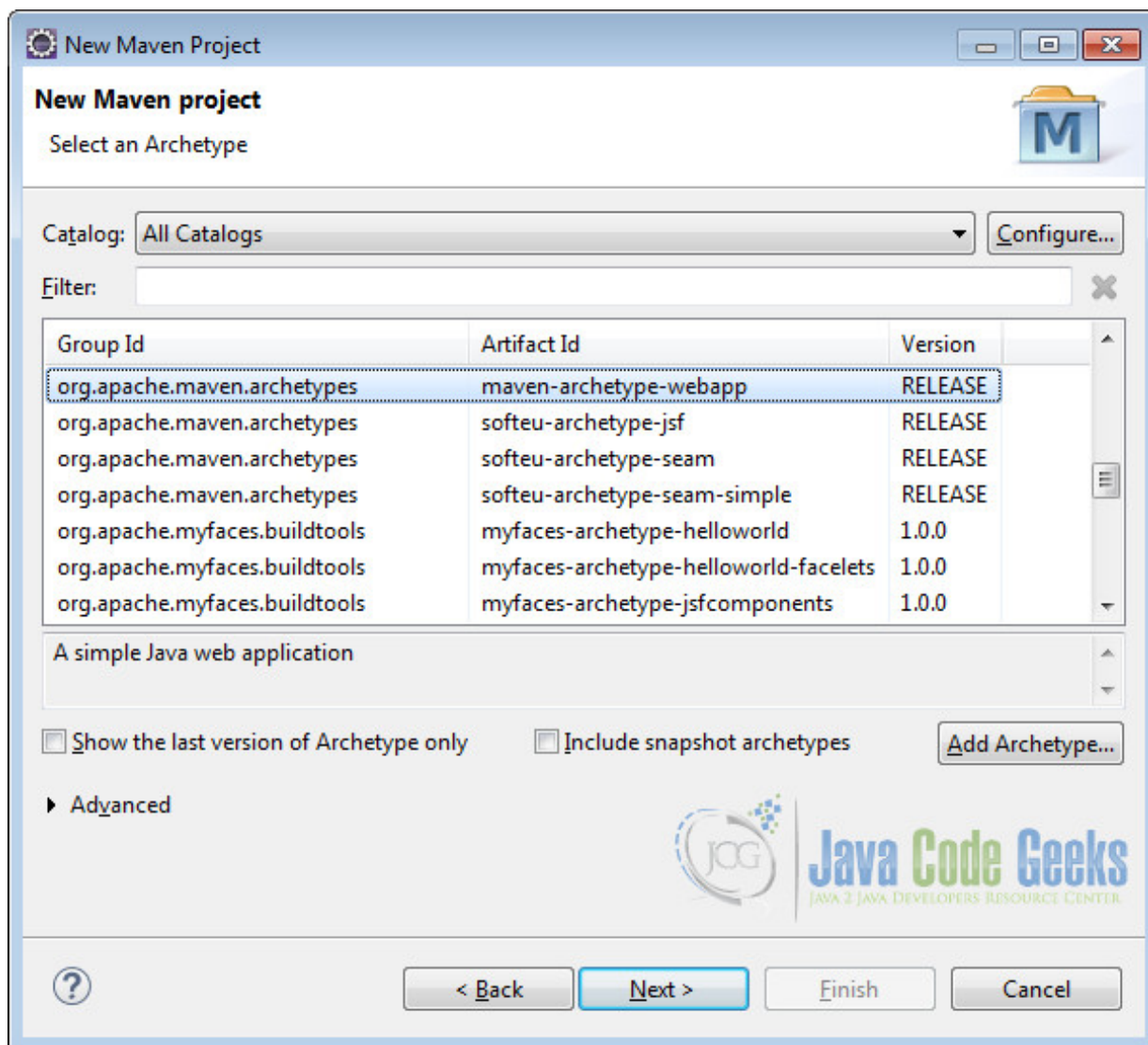


Figure 16.3: Maven setup - step 3

In the “Enter an artifact id” page of the wizard, you can define the name and main package of your project. Set the “Group Id” variable to "com.javacodegeeks.snippets.enterprise" and the “Artifact Id” variable to "jsftemplate". For package enter "com.javacodegeeks.jsftemplate" and hit “Finish” to exit the wizard and to create your project.

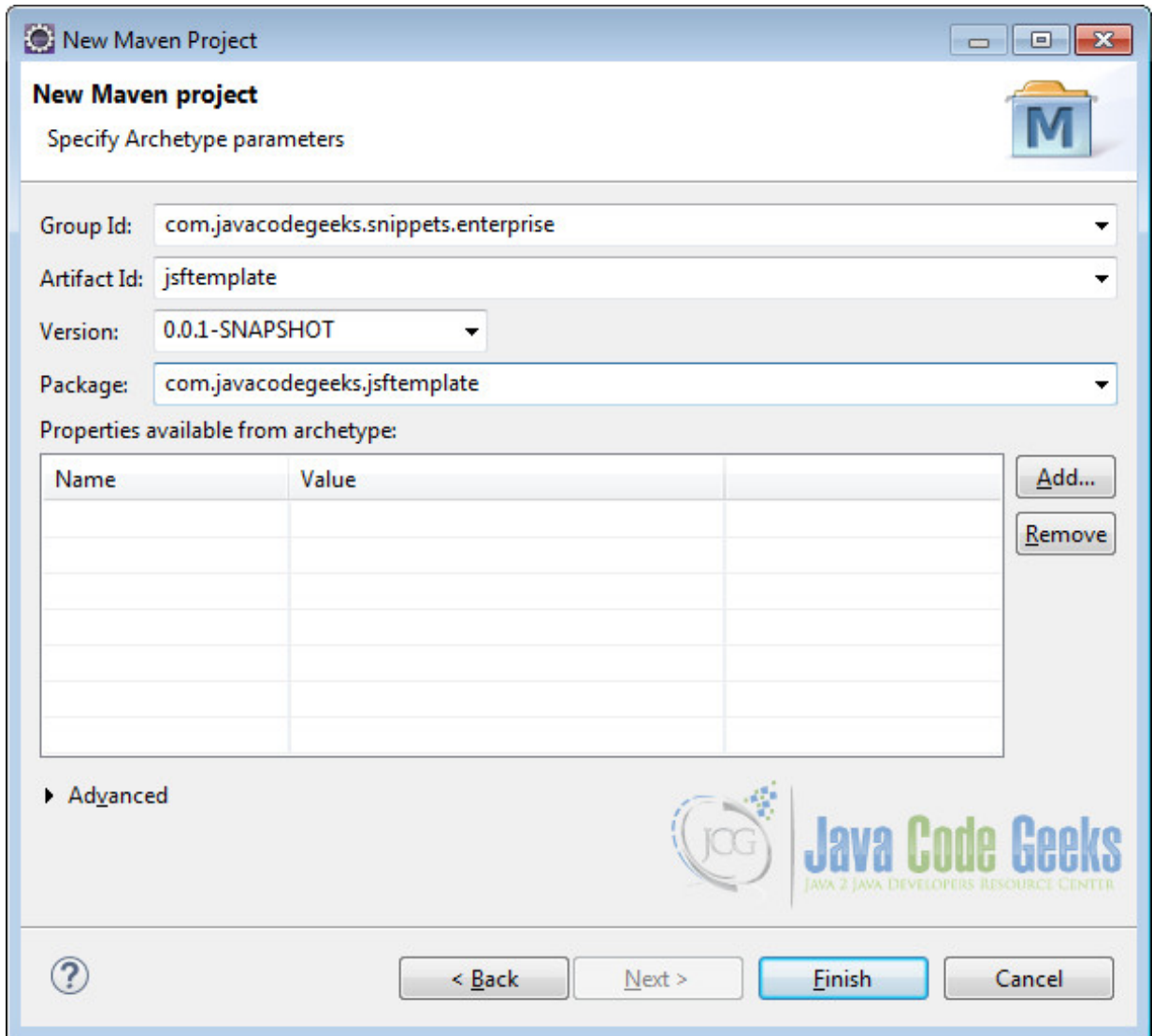


Figure 16.4: Maven setup - step 4

Refresh the project. Finally, the project structure will look like the below.

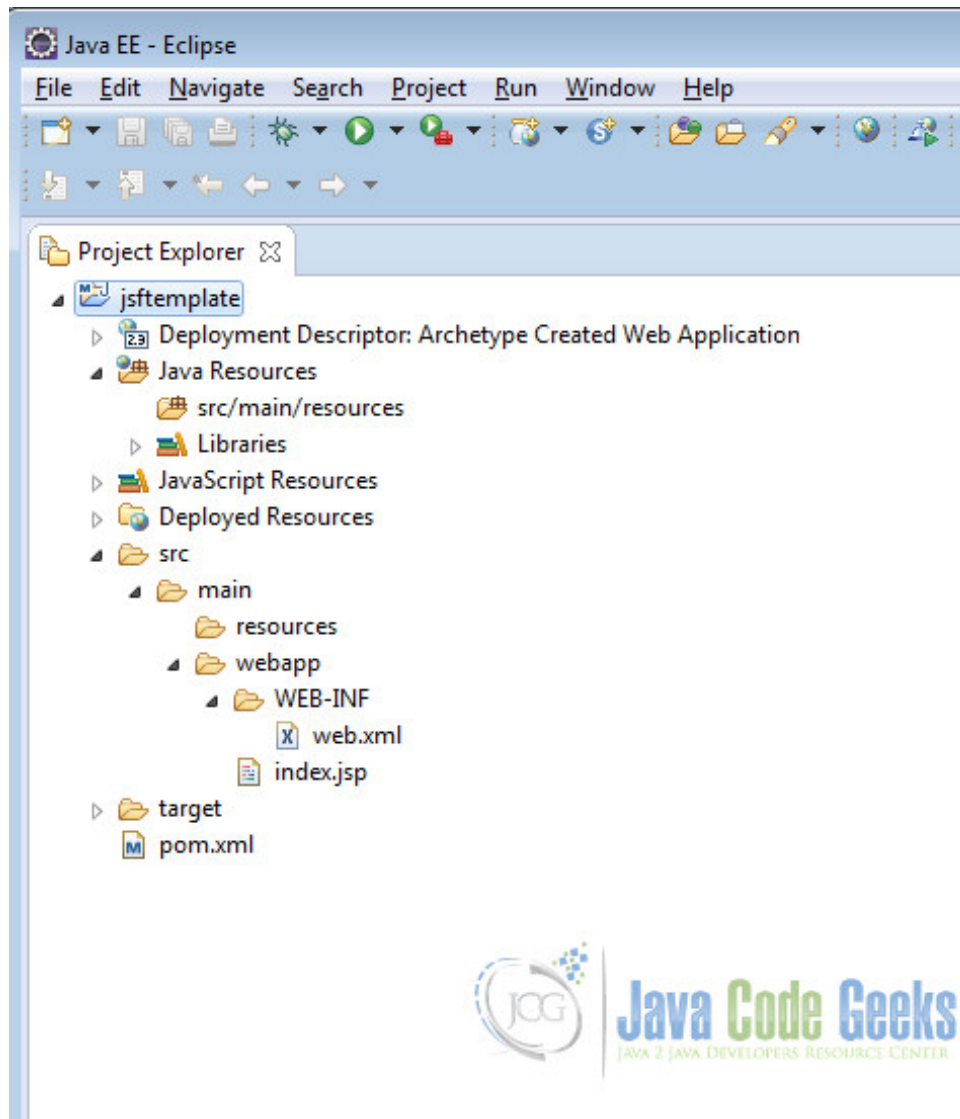


Figure 16.5: Maven setup - step 5

If you see any errors in the `index.jsp`, set target runtime for the project.

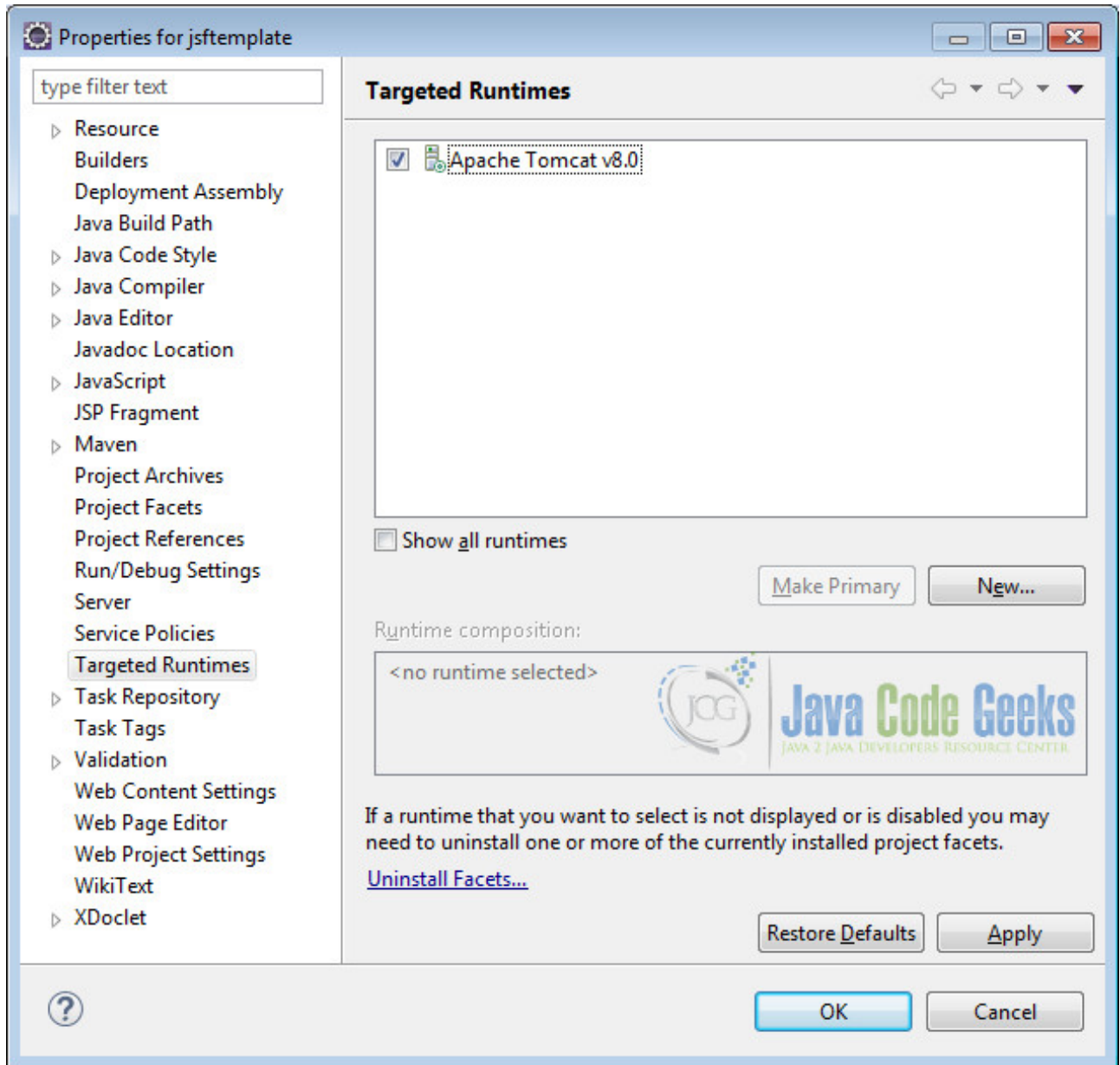


Figure 16.6: Maven setup - step 6

## 16.2 Modify POM to include JSF dependency

Add the dependencies in Maven's pom.xml file, by editing it at the "Pom.xml" page of the POM editor

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0
    .xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.snippets.enterprise</groupId>
  <artifactId>jsftemplate</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>jsftemplate Maven Webapp</name>
```



```
<url>http://maven.apache.org</url>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.2.9</version>
  </dependency>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>2.2.9</version>
  </dependency>
</dependencies>
<build>
  <finalName>jsftemplate</finalName>
</build>
</project>
```

## 16.3 Add Faces Servlet in web.xml

The `web.xml` file has to be modified to include the faces servlet configuration as below.

```
<web-app>
  <display-name>Archetype Created Web Application</display-name>

  <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>
</web-app>
```

## 16.4 JSF Facelets Tags

JSF provides the following facelets tags, using which we can create the template.

- `ui:composition` - Defines the composition that optionally uses a template. JSF disregards everything outside the composition tags which allows embedding the compositions into well formed XHTML pages
- `ui:insert` - It is used to insert content into the template file.
- `ui:include` - It is very similar to `jsp:include` to encapsulate and reuse content among multiple XHTML page.
- `ui:define` - It is used to define content matching the `ui:insert` tag of the template.

## 16.5 Create a Template

Template in a web application is nothing but the layout of the webpage. We will create a simple layout with header, content and footer sections. For modularity we will create each section in a different file and include them in the template.

Create a folder called `templates` under `webapp` to place all the template files.

### 16.5.1 5.1 Create Header

In the `/webapp/templates/` folder create a file called `header.xhtml`. We will use the facelets tag `ui:composition` to add the content for the page. As discussed earlier, JSF ignores everything outside the `ui:composition` tag.

`header.xhtml`

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
  xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Insert title here</title>
</head>
<body>
  <ui:composition>
    <h1>Header</h1>
  </ui:composition>
</body>
</html>
```

### 16.5.2 5.2 Create Content

In the `/webapp/templates/` folder create a file called `content.xhtml`. We will use the `ui:composition` to add the content for the page.

`content.xhtml`

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
  xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Insert title here</title>
</head>
<body>
  <ui:composition>
    <h1>Content goes here...</h1>
  </ui:composition>
</body>
</html>
```

### 16.5.3 5.3 Create Footer

In the `/webapp/templates/` folder create a file called `footer.xhtml`. We will use the `ui:composition` to add the content for the page.

`footer.xhtml`

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
  xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Insert title here</title>
</head>
<body>
<ui:composition>
  <h1>Footer</h1></ui:composition>
</body>
</html>
```

### 16.5.4 5.4 Finally the Template

In the `/webapp/templates/` folder create a file called `template.xhtml`. We will insert the header, content and footer sections into a template file by using `ui:insert`. The source for the different sections is defined by the facelets tag `ui:include` by means of pointing the `src` attribute to the relevant file.

`template.xhtml`

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
  xmlns:ui="http://java.sun.com/jsf/facelets">
<h:head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
  <title>Insert title here</title>
</h:head>
<h:body>

  <div style="border-width: 1px; border-style: solid; height: 25%">
  <ui:insert name="header">
  <ui:include src="/templates/header.xhtml"></ui:include>
  </ui:insert>

  <ui:insert name="content">
  <ui:include src="/templates/content.xhtml"></ui:include>
  </ui:insert>

  <ui:insert name="footer">
  <ui:include src="/templates/footer.xhtml"></ui:include>
  </ui:insert>

  </div>
</h:body>
</html>
```

## 16.6 Default page using template

We will show you how to make a webpage using the template. In this default page, we will use the template as it is. In other words, we will not use `ui:define` to modify the template content.

We will create a file called `defaultPage.xhtml` under `/webapp` folder. The `template` attribute in the `ui:composition` facelets tag is used to define the template for the page. In this example we are pointing the attribute to `templates/template.xhtml`

`defaultPage.xhtml`

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
  xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Insert title here</title>
</head>
<h:body>
  <ui:composition template="templates/template.xhtml">
    </ui:composition>
</h:body>
</html>
```

Now we can create the deployment package using `Run as → Maven clean` and then `Run as → Maven install`. This will create a `war` file in the target folder. The `war` file produced must be placed in `webapps` folder of tomcat. Now we can start the server.

Open the following URL in the browser

<http://localhost:8080/jsftemplate/defaultPage.xhtml>

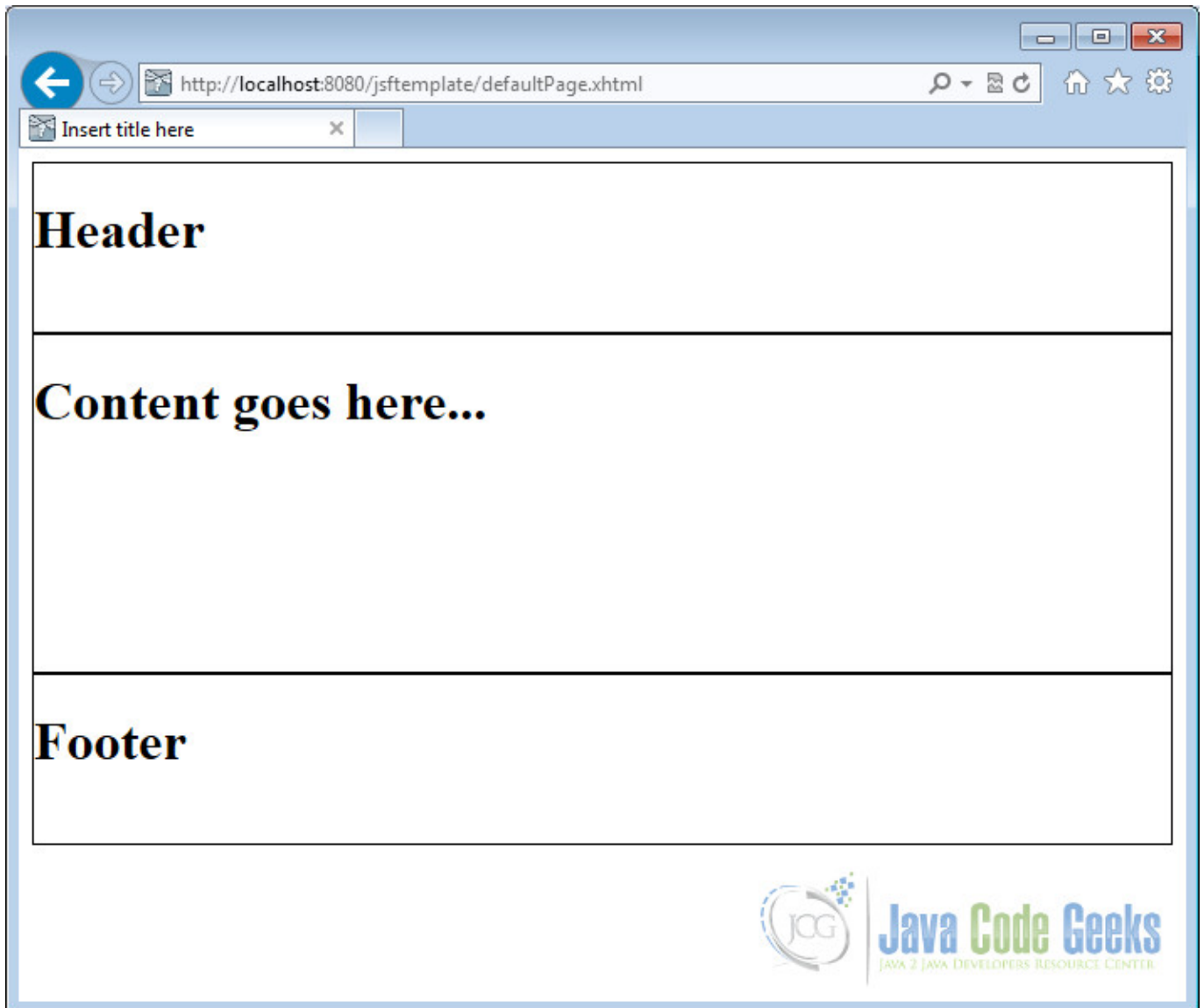


Figure 16.7: Default Page

## 16.7 Welcome page using template

In the welcome page we will modify the content of the template using `ui:define` tag. Create a file called `welcomePage.xhtml` under `/webapp` folder.

We will first use `ui:composition` to define the template for the web page. Next, we will use the `ui:define` to modify the content representing the current page. Note that, the name attribute of the `ui:define` tag should match with the name attribute of the `ui:insert` specified in the template file.

`welcomePage.xhtml`

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Insert title here</title>
</head>
<h:body>
  <ui:composition template="/templates/template.xhtml">
    <ui:define name="content">
      <h1>Welcome Page Content...</h1>
      <h3>Now we know how to use JSF Templates</h3>
      <h4>We can create multiple pages with the same template</h4>
    </ui:define>
  </ui:composition>
</h:body>
</html>
```

Now again package using maven and copy the war to the apache tomcat webapps folder. Open the following URL in the browser

<http://localhost:8080/jsftemplate/welcomePage.xhtml>

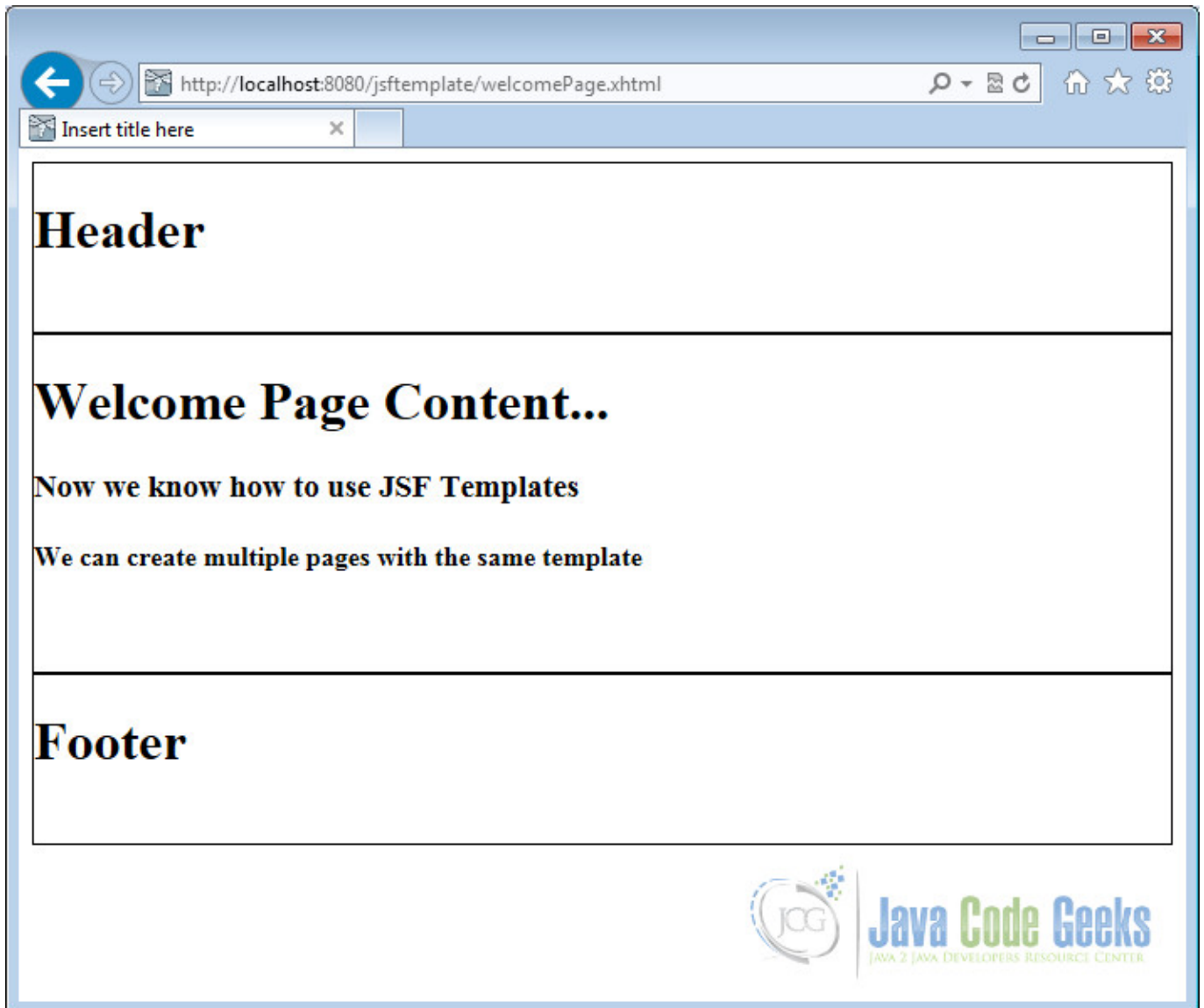


Figure 16.8: Welcome Page

## 16.8 Download the Eclipse Project

This was an example for creating templates using JSF Facelets.

**Download** You can download the full source code of this example here : [JSF Facelets Template](#)

## Chapter 17

# Standard Converters Example

In this example of JSF Standard Converters, we will show you how Java Server Faces standard converters work and also discuss the various options available to use standard converters.

When a request is sent from the browser to the application server, the form values are sent as String. To convert the String into Java Objects, we need to use converters. Similarly when the Java Object is passed back from the application server and gets converted into HTML, we need to convert it to String. JSF provides a set of standard converters as well as give option to create custom converters. Let's begin with setting up a JSF project and do all the necessary configuration to run the application.

Our preferred environment is Eclipse. We are using Eclipse Luna SR1 with Maven Integration Plugin, JDK 8u25 (1.8.0\_25) and Tomcat 8 application server. Having said that, we have tested the code against JDK 1.7 and Tomcat 7 as well.

### 17.1 Create a new Maven Project

Go to File → New → Other → Maven Project



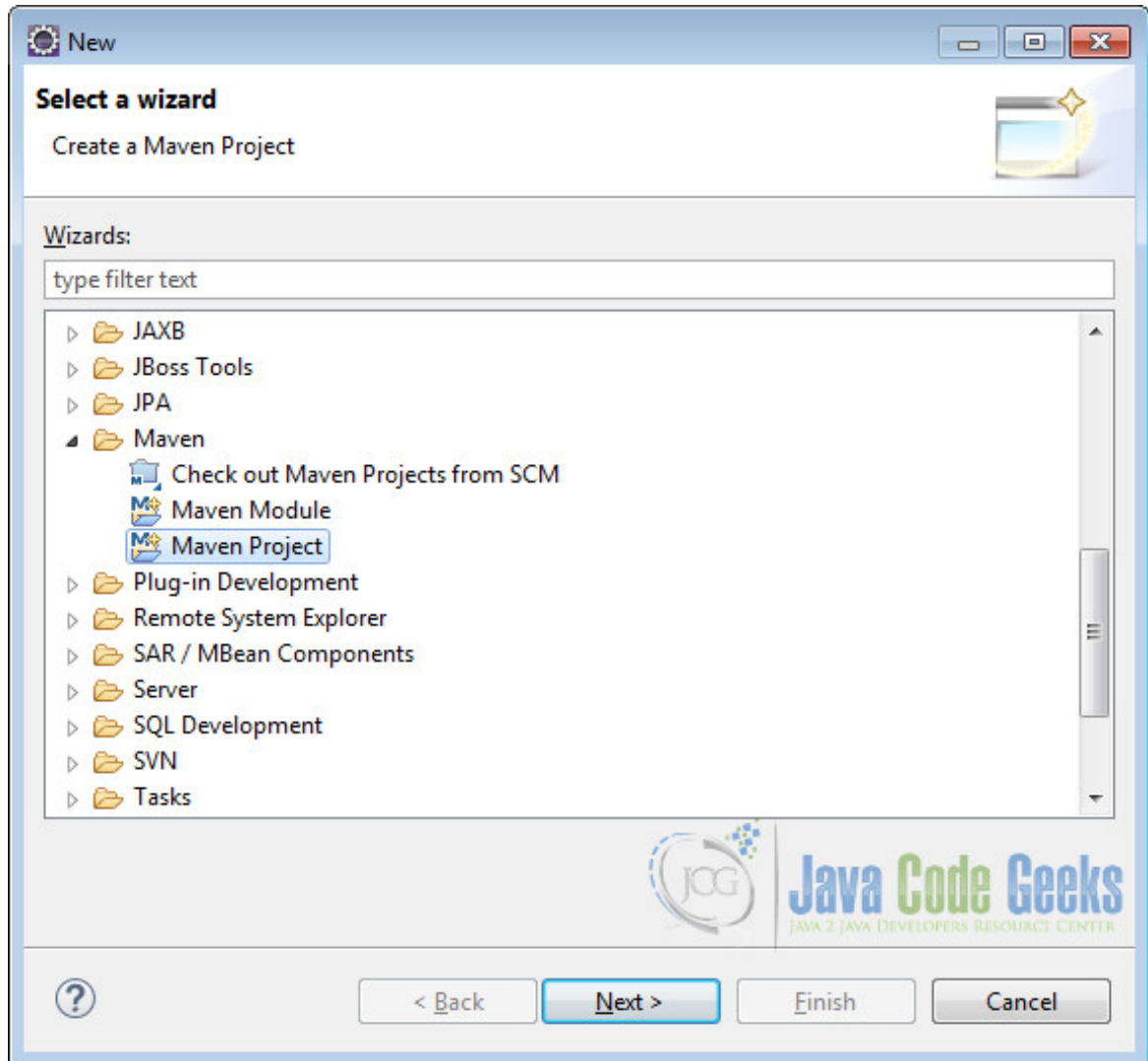


Figure 17.1: Maven Setup - Step 1

In the “Select project name and location” page of the wizard, make sure that “Create a simple project (skip archetype selection)” option is **unchecked**, hit “Next” to continue with default values.

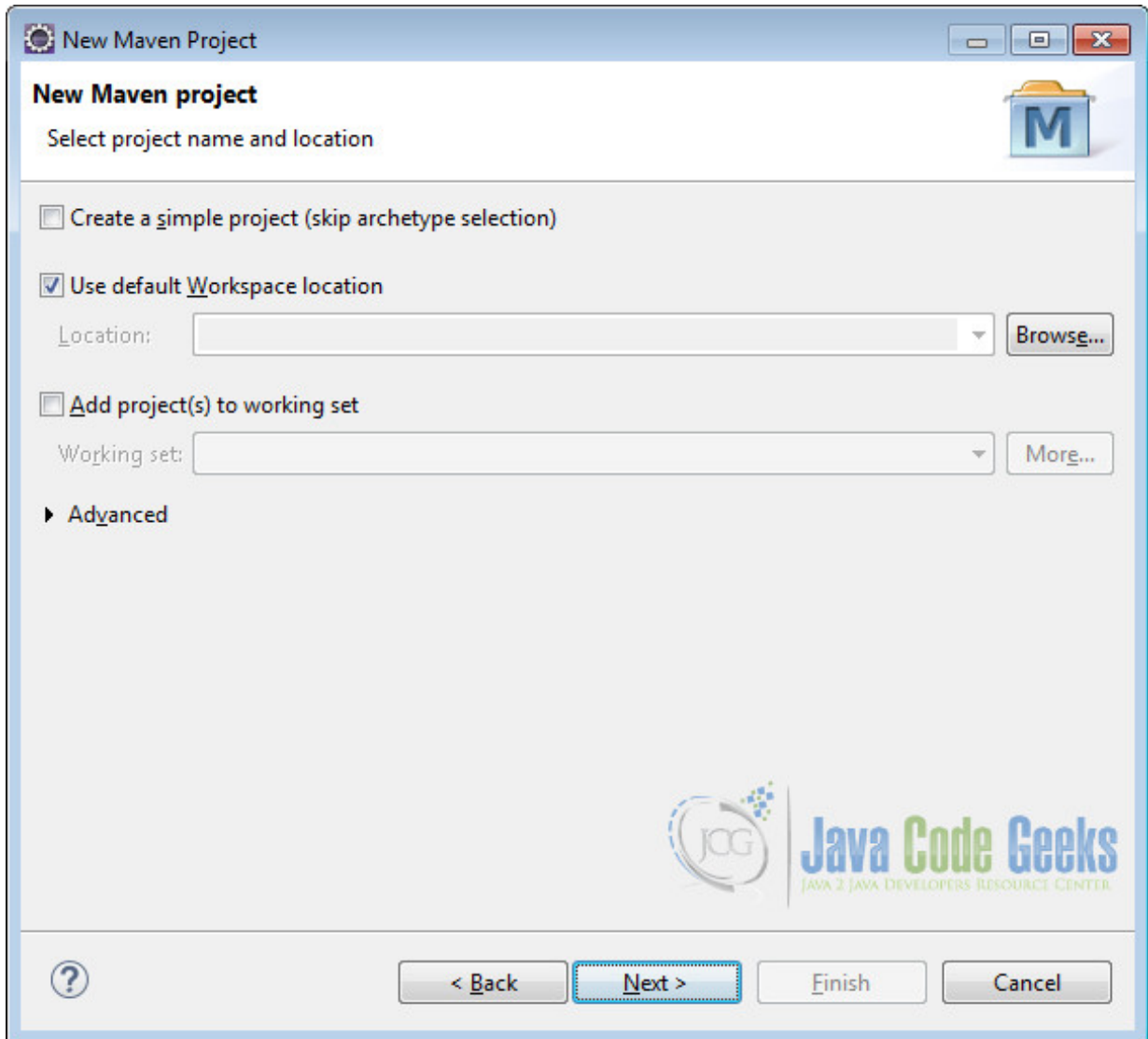


Figure 17.2: Maven setup - step 2

Here choose “maven-archetype-webapp” and click on Next.

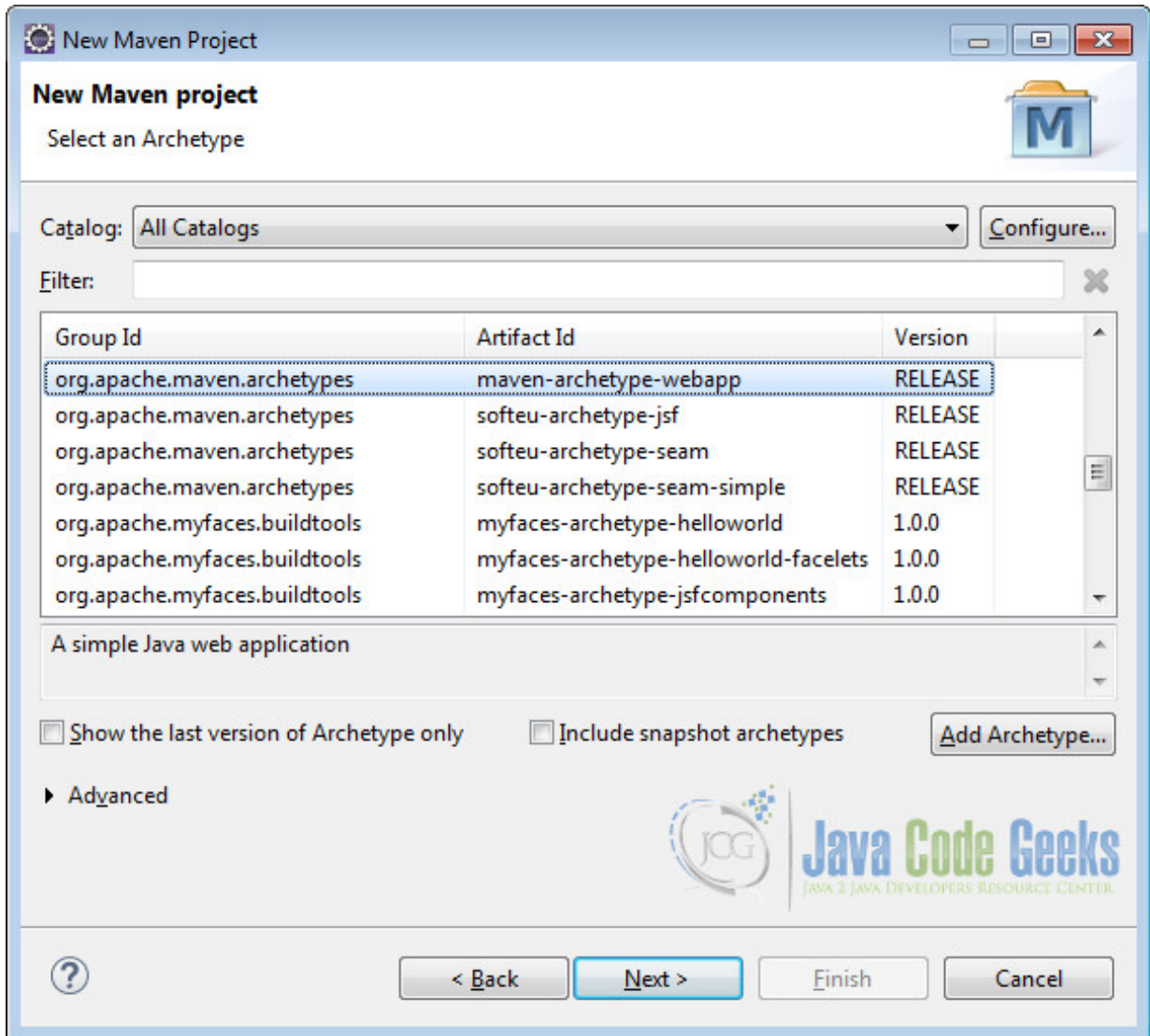


Figure 17.3: Maven setup - step 3

In the “Enter an artifact id” page of the wizard, you can define the name and main package of your project. Set the “Group Id” variable to `com.javacodegeeks.snippets.enterprise` and the “Artifact Id” variable to `jsfconverters`. For package enter `com.javacodegeeks.jsfconverters` and hit “Finish” to exit the wizard and to create your project.

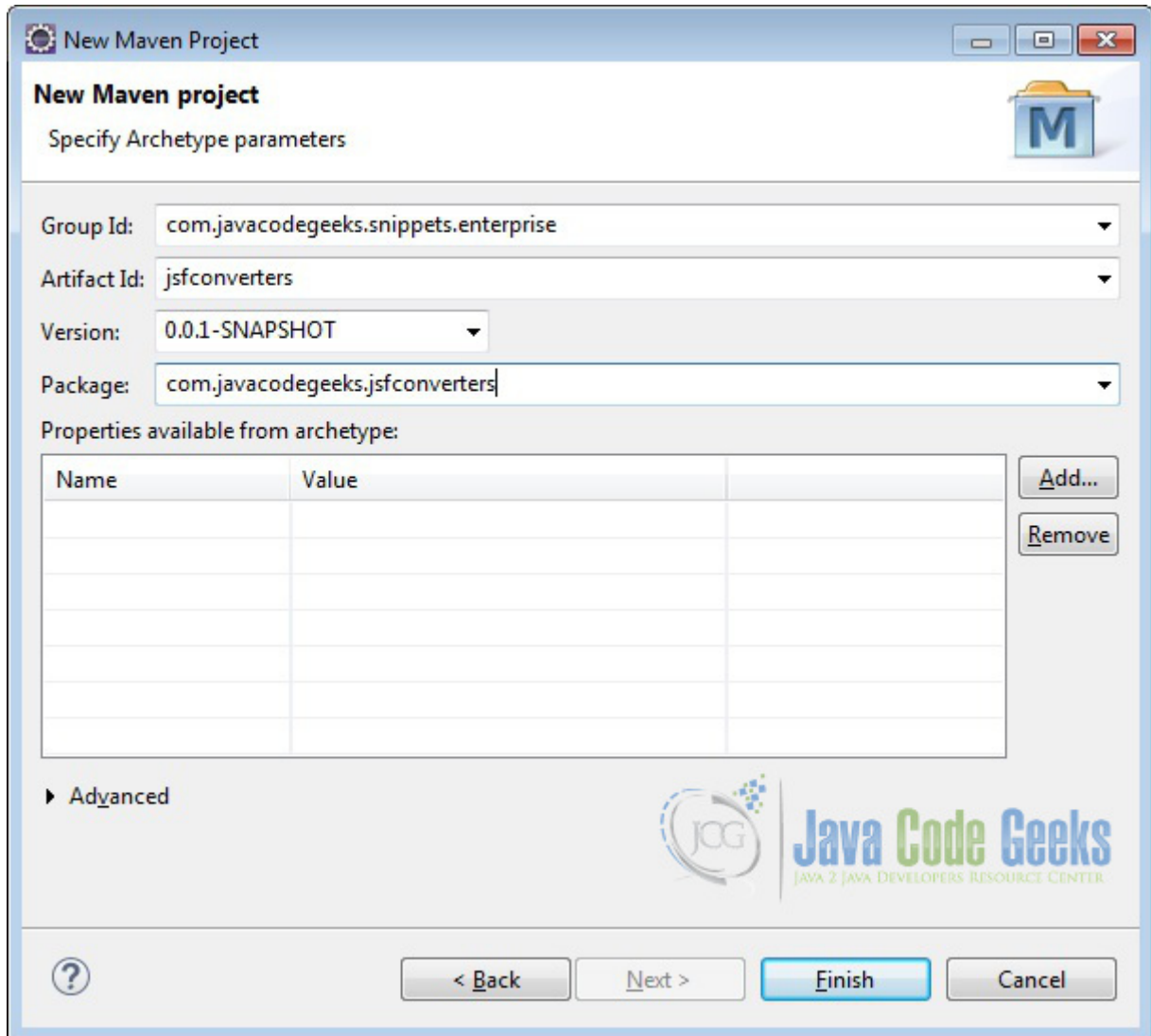


Figure 17.4: Maven setup - step 4

Now create a folder called `java` under `src/main`.

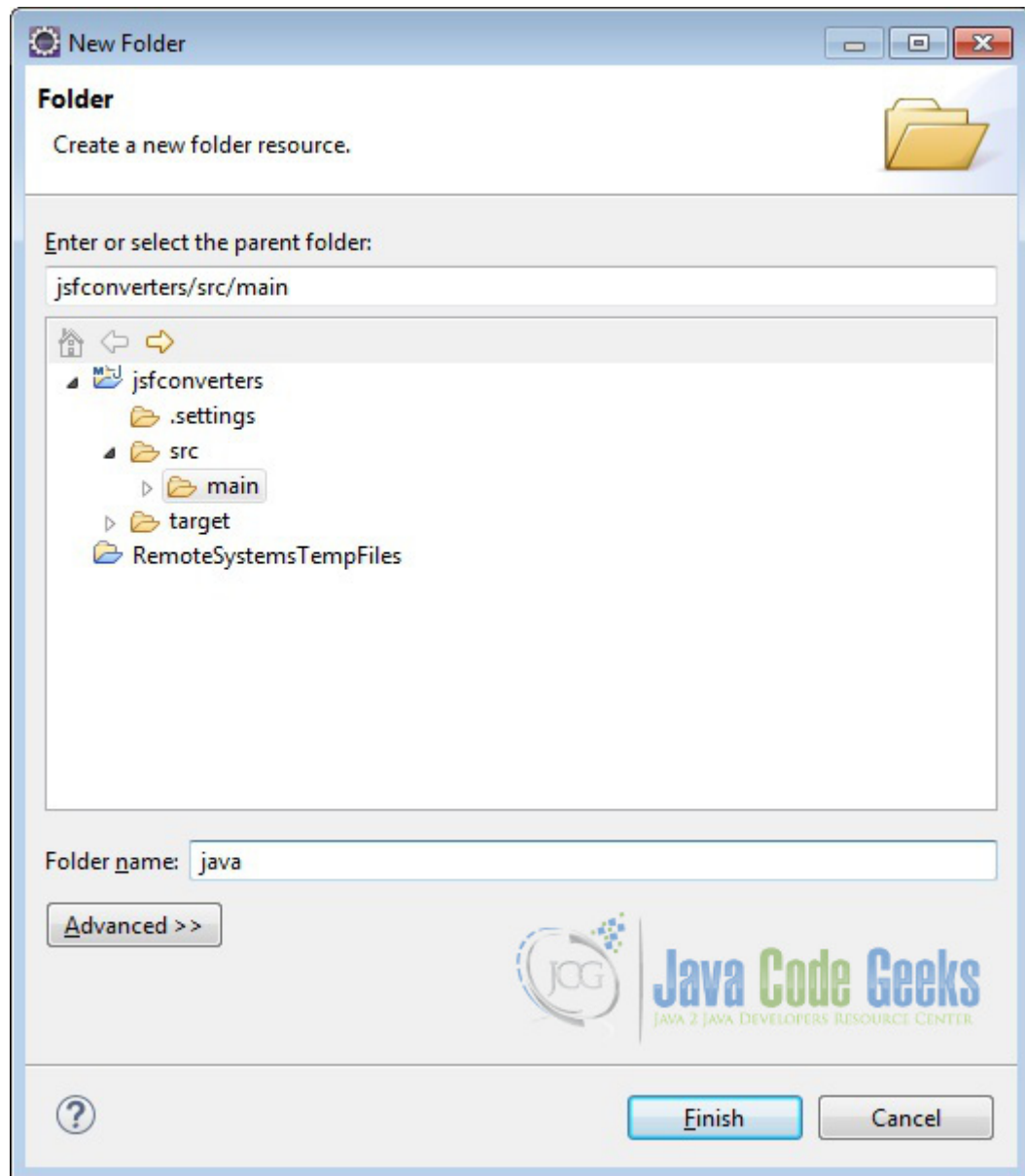


Figure 17.5: Maven setup - step 5

Refresh the project. Finally, the project structure will look like the below.

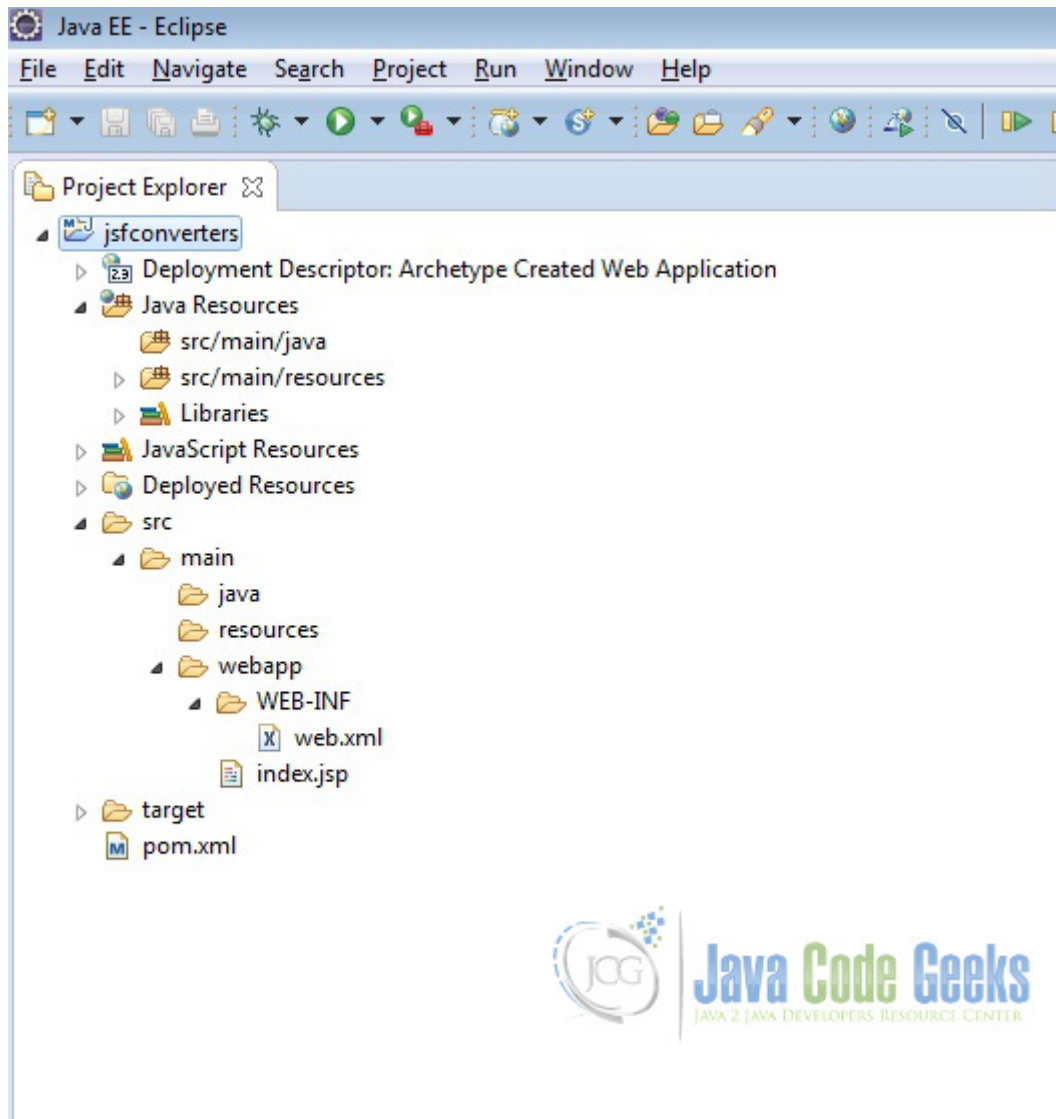


Figure 17.6: Maven setup - step 6

If you see any errors in the `index.jsp`, set target runtime for the project.

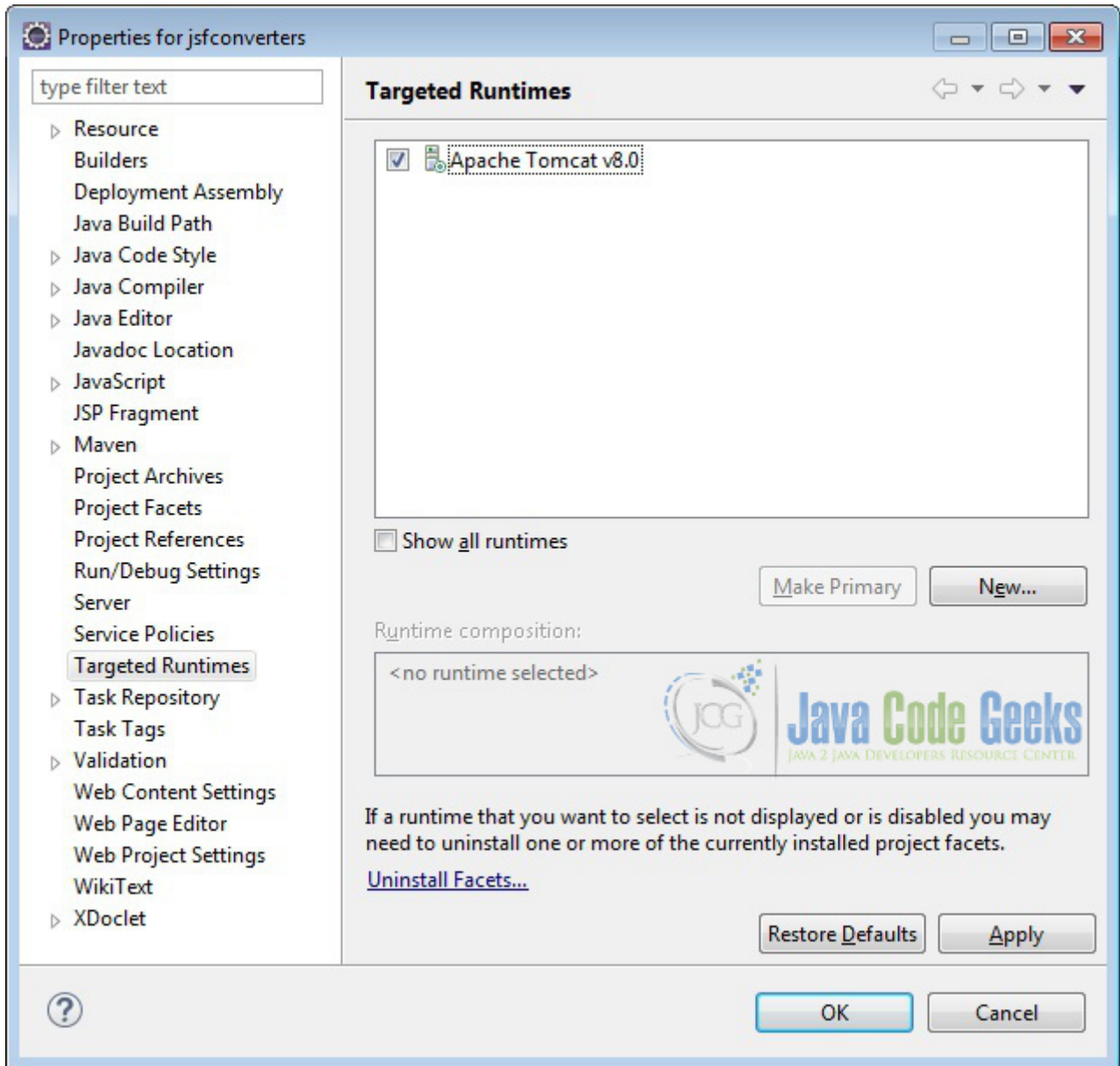


Figure 17.7: Maven setup - step 7

## 17.2 Modify POM to include JSF dependency

Add the dependencies in Maven's `pom.xml` file, by editing it at the "Pom.xml" page of the POM editor.

`pom.xml`

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.javacodegeeks.snippets.enterprise</groupId>
<artifactId>jsfconverters</artifactId>
<packaging>war</packaging>
<version>0.0.1-SNAPSHOT</version>
```



```
<name>jsfconverters Maven Webapp</name>
<url>http://maven.apache.org</url>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.2.9</version>
  </dependency>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>2.2.9</version>
  </dependency>
</dependencies>
<build>
  <finalName>jsfconverters</finalName>
</build>
</project>
```

## 17.3 Add Faces Servlet in web.xml

The `web.xml` file has to be modified to include the faces servlet configuration as below.

`web.xml`

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <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>
</web-app>
```

[code](#)

## 17.4 Standard Converters

JSF provides a set of standard converters. The following section shows all the available converters and state the purpose for each of the converter.

- **BigDecimalConverter** - For Conversion between String and `java.math.BigDecimal`



- **BigIntegerConverter** - For Conversion between String and `java.math.BigInteger`
- **BooleanConverter** - For Conversion between String and `java.lang.Boolean`
- **ByteConverter** - For Conversion between String and `java.lang.Byte`
- **CharacterConverter** - For Conversion between String and `java.lang.Character`
- **DateTimeConverter** - For Conversion between String and `java.util.Date`
- **DoubleConverter** - For Conversion between String and `java.lang.Double`
- **FloatConverter** - For Conversion between String and `java.lang.Float`
- **IntegerConverter** - For Conversion between String and `java.lang.Integer`
- **LongConverter** - For Conversion between String and `java.lang.Long`
- **NumberConverter** - For Conversion between String and `java.lang.Number`
- **ShortConverter** - For Conversion between String and `java.lang.Short`

The **DateTimeConverter** and **NumberConverter** have their own tags and provide attributes for data conversion. We will discuss about the two converters in detail later.

## 17.5 How to use Converters

JSF provides three ways to use converters. We can use any of the method depending on the type of converter.

### 17.5.1 5.1 Using converter attribute

We can add the `converter` attribute to the UI component using the fully qualified class name.

```
<h:inputText id="age" converter="javax.faces.Integer" />
```

### 17.5.2 5.2 Using f:converter tag

We can include the `f:converter` tag within a component. The `converterID` attribute point to reference the converter's name.

```
<h:inputText id="age">
  <f:converter converterID="javax.faces.Integer" />
</h:inputText>
```

### 17.5.3 5.3 Using converter tags

We can use the standard converter tags provided in the JSF.

```
<h:outputText value="#{userBean.height}">
  <f:convertNumber maxFractionDigits="2" />
</h:outputText>
```

Or by using custom converter

```
<h:outputText value="#{userBean.ssn}">
  <my:ssnConverter country="US" />
</h:outputText>
```

## 17.6 Implicit Converter

JSF provides standard converters that automatically perform conversion for basic Java types. We will see it working by creating an example.

First we will create a package called `com.javacodegeeks.jsfconverters` under the folder `src/main/java`. Now, we need to create a managed bean called `UserBean`. The `@ManagedBean` annotation makes the POJO as managed bean. The `@SessionScoped` annotation on the bean makes the bean available to the entire user session.

`UserBean.java`

```
package com.javacodegeeks.jsfconverters;

import java.util.Date;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name="userBean", eager=true)
@SessionScoped
public class UserBean {

    private String firstName;
    private String lastName;
    private int age;
    private Date dateOfBirth;
    private Double height;

    public String getFirstName() {
        return firstName;
    }

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

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public int getAge() {
        return age;
    }

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

    public Date getDateOfBirth() {
        return dateOfBirth;
    }

    public void setDateOfBirth(Date dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
    }

    public Double getHeight() {
        return height;
    }
}
```

```

public void setHeight(Double height) {
    this.height = height;
}
}

```

Now, we will create a view called `adduser.xhtml` under `/src/main/webapp`. We have used `h:outputLabel` to display the label and `h:inputText` to get the user input. We will submit the form using the component `h:commandButton`. The action tag of `h:commandButton` is set to `"viewuser"`. We will use the implicit navigation feature of JSF for navigating to the `"viewuser"` page.

`adduser.xhtml`

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Add User</title>
</head>
<body>
    <h:form>
        <h2>Add User</h2>

        <h:outputLabel for="firstName">First Name</h:outputLabel>

        <h:inputText id="firstName" label="First Name"
                    value="#{userBean.firstName}"></h:inputText>

        <h:outputLabel for="lastName">Last Name</h:outputLabel>

        <h:inputText id="lastName" label="Last Name"
                    value="#{userBean.lastName}"></h:inputText>

        <h:outputLabel for="age">Age</h:outputLabel>

        <h:inputText id="age" label="age" value="#{userBean.age}">
        </h:inputText>

        &nbsp;&nbsp;&nbsp;
        &nbsp;&nbsp;&nbsp;

        <h:commandButton value="Submit" action="viewuser"></h:commandButton>

    </h:form>
</body>
</html>

```

We will create another view called `viewuser.xhtml` under `/src/main/webapp` to display the values entered by the user.

`viewuser.xhtml`

---

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>View User</title>
</head>
<body>
  <h:form>
    <h2>View User</h2>
    <h4>
      <h:outputText value="#{userBean.firstName}"></h:outputText>
      <br />
      <h:outputText value="#{userBean.lastName}"></h:outputText>
      <br />
      <h:outputText value="#{userBean.age}"></h:outputText>
    </h4>
  </h:form>
</body>
</html>
```

Now we can create the deployment package using Run as → Maven clean and then Run as → Maven install. This will create a war file in the target folder. The war file produced must be placed in webapps folder of tomcat. Now we can start the server.

Open the following URL in the browser.

<http://localhost:8080/jsfconverters/adduser.xhtml>



The screenshot shows a web browser window with the address bar containing `http://localhost:8080/jsfconverters/adduser.xhtml`. The browser tab is titled "Add User". The main content area displays a form with the following elements:

- Add User** (Section Header)
- First Name** (Label) with an empty text input field.
- Last Name** (Label) with an empty text input field.
- Age** (Label) with a text input field containing the value `0`.
- Submit** (Button)

In the bottom right corner, there is a logo for "Java Code Geeks" with the tagline "JAVA 2 JAVA DEVELOPERS RESOURCE CENTER".

Figure 17.8: Add User -1

Enter the values for First Name, Last Name and age. Now, click on submit button. JSF will use the implicit navigation and display the `viewuser.xhtml`. Here the value for age is converted to `int` by the standard converter automatically.



Figure 17.9: View User - 1

To validate the implicit converter, try to enter **some characters** in the age field and click on the submit. You will see error in **Tomcat server window**.

## 17.7 DateTimeConverter

The JSF DateTimeConverter provides the following attributes to convert and format the Date.

- **dateStyle** - Predefined formatting style which determines how the date component of a date string is to be formatted and parsed.
- **locale** - Locale to be used during formatting.
- **pattern** - Custom formatting pattern can be used using this attribute.
- **timeStyle** - Predefined formatting style which determines how the time component of a date string is to be formatted and parsed.
- **timeZone** - Time zone in which to interpret any time information in the date String.
- **type** - Specifies date or time or both to be used during formatting.

Now, we modify the `adduser.xhtml` to accept the Date of Birth as user input.

`adduser.xhtml`


```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
```



```
<h:form>
  <h2>View User</h2>
  <h4>
    <h:outputText value="#{userBean.firstName}"></h:outputText>
    <br />
    <h:outputText value="#{userBean.lastName}"></h:outputText>
    <br />
    <h:outputText value="#{userBean.age}"></h:outputText>
    <br />
    <h:outputText value="#{userBean.dateOfBirth}">
      <f:convertDateTime pattern="dd-MMM-yy"></f:convertDateTime>
    </h:outputText>
  </h4>
</h:form>
</body>
</html>
```

Now again package using maven and copy the war to the apache tomcat webapps folder. Open the following URL in the browser

<http://localhost:8080/jsfconverters/adduser.xhtml>



The screenshot shows a web browser window with the address bar containing `http://localhost:8080/jsfconverters/adduser.xhtml`. The page title is "Add User". The form contains the following elements:

- First Name**: An empty text input field.
- Last Name**: An empty text input field.
- Age**: A text input field containing the value "0".
- Date of Birth (dd-mm-yyyy)**: An empty text input field.
- Submit**: A button located at the bottom left of the form.

In the bottom right corner, there is a logo for "Java Code Geeks" with the tagline "JAVA 2 JAVA DEVELOPERS RESOURCE CENTER".

Figure 17.10: Add User -2

Enter the values for first name, last name, age and date of birth. Now, click on the submit button. The JSF will use the implicit navigation to forward the request to `viewuser.xhtml`. We will see the date of birth being displayed in the new format ``dd-MMM-yy`` defined using the pattern attribute.





Figure 17.11: View User -2

## 17.8 NumberConverter

The JSF NumberConverter provides the following attributes to convert and format the number.

- **currencyCode** - To apply the currency code.
- **currencySymbol** - To apply the currency symbol.
- **groupingUsed** - Flag specifying whether formatted output will contain grouping separators.
- **integerOnly** - Flag specifying whether only the integer part of the value will be formatted and parsed.
- **locale** - `<span class="changed_modified_2_0">Locale</span>` whose predefined styles for numbers are used during formatting and parsing.
- **maxFractionDigits** - Maximum number of digits in the fractional portion.
- **maxIntegerDigits** - Maximum number of digits in the integer portion.
- **minFractionDigits** - Minimum number of digits in the fractional portion.
- **minIntegerDigits** - Minimum number of digits in the integer portion.
- **pattern** - To define the custom formatting pattern.
- **type** - Specifies one of number, currency and percent.

Now, we modify the adduser.xhtml to accept the height as user input.

adduser.xhtml

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Add User</title>
</head>
<body>
  <h:form>
    <h2>Add User</h2>
    <div>
      <h:outputLabel for="firstName">First Name</h:outputLabel>

      <h:inputText id="firstName" label="First Name"
        value="#{userBean.firstName}"></h:inputText>

      <h:outputLabel for="lastName">Last Name</h:outputLabel>

      <h:inputText id="lastName" label="Last Name"
        value="#{userBean.lastName}"></h:inputText>

      <h:outputLabel for="age">Age</h:outputLabel>

      <h:inputText id="age" label="age" value="#{userBean.age}">
</h:inputText>

      <h:outputLabel for="dateOfBirth">Date of Birth (dd-mm-yyyy)</h:outputLabel>

      <h:inputText id="dateOfBirth" label="Date of Birth"
        value="#{userBean.dateOfBirth}">
        <f:convertDateTime pattern="dd-mm-yyyy" />
</h:inputText>

      <h:outputLabel for="height">Height</h:outputLabel>

      <h:inputText id="height" label="Height" value="#{userBean.height}"></ ←
h:inputText>

      &nbsp;
      &nbsp;

      <h:commandButton value="Submit" action="viewuser"></h:commandButton>

    </h:form>
  </body>
</html>

```

Now modify the `viewuser.xhtml` to display the height with single digit in fraction part. We will use the `maxFractionDigits` attribute of the `f:convertNumber` tag to achieve this.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>View User</title>
</head>
<body>
  <h:form>
    <h2>View User</h2>
    <h4>
      <h:outputText value="#{userBean.firstName}"></h:outputText>
      <br />
      <h:outputText value="#{userBean.lastName}"></h:outputText>
      <br />
      <h:outputText value="#{userBean.age}"></h:outputText>
      <br />
      <h:outputText value="#{userBean.dateOfBirth}">
        <f:convertDateTime pattern="dd-MMM-yy"></f:convertDateTime>
      </h:outputText>
      <br />
      <h:outputText value="#{userBean.height}">
        <f:convertNumber maxFractionDigits="1" />
      </h:outputText>
    </h4>
  </h:form>
</body>
</html>
```

Now again package using maven and copy the war to the apache tomcat webapps folder. Open the following URL in the browser

<http://localhost:8080/jsfconverters/adduser.xhtml>



The screenshot shows a web browser window with the address bar containing `http://localhost:8080/jsfconverters/adduser.xhtml`. The page title is "Add User". The form contains the following fields and values:

Field Label	Value
First Name	Rob
Last Name	Martin
Age	35
Date of Birth (dd-mm-yyyy)	01-05-1980
Height	172.25

A "Submit" button is located at the bottom left of the form. In the bottom right corner, there is a logo for "Java Code Geeks" with the tagline "JAVA 2 JAVA DEVELOPERS RESOURCE CENTER".

Figure 17.12: Add User -3

Enter the values for first name, last name, age, date of birth and height (two decimal digits). Now, click on the submit button. The JSF will use the implicit navigation to forward the request to `viewuser.xhtml`. We will see the height being displayed with single fraction digit irrespective of the user input.



Figure 17.13: View User -3

## 17.9 Download the Eclipse Project

This was an example of how to use Java Server Faces Standard Converters. **Download** You can download the full source code of this example here: [JSF Standard Converters](#)

## Chapter 18

# Components Listeners Example

In this example of JSF Components Listeners, we will discuss about various component listeners provided by Java Server Faces and show you different ways of using the listeners.

In a web page when the user makes changes to the input component or performs an action on the UI component, the JSF fires an event. These events can be handled by application to take necessary action. JSF provides listeners to capture the event. We can implement the listeners as classes or use the backing bean method to capture the event. Depending upon how the listener is implemented, the page can either use listener tag or listener attribute of the UI component. We will show you both the approaches here. Let's begin with setting up a JSF project and do all the necessary configuration to run the application.

Our preferred environment is Eclipse. We are using Eclipse Luna SR1 with Maven Integration Plugin, JDK 8u25 (1.8.0\_25) and Tomcat 8 application server. Having said that, we have tested the code against JDK 1.7 and Tomcat 7 as well.

### 18.1 Create a new Maven Project

Go to File → New → Other → Maven Project

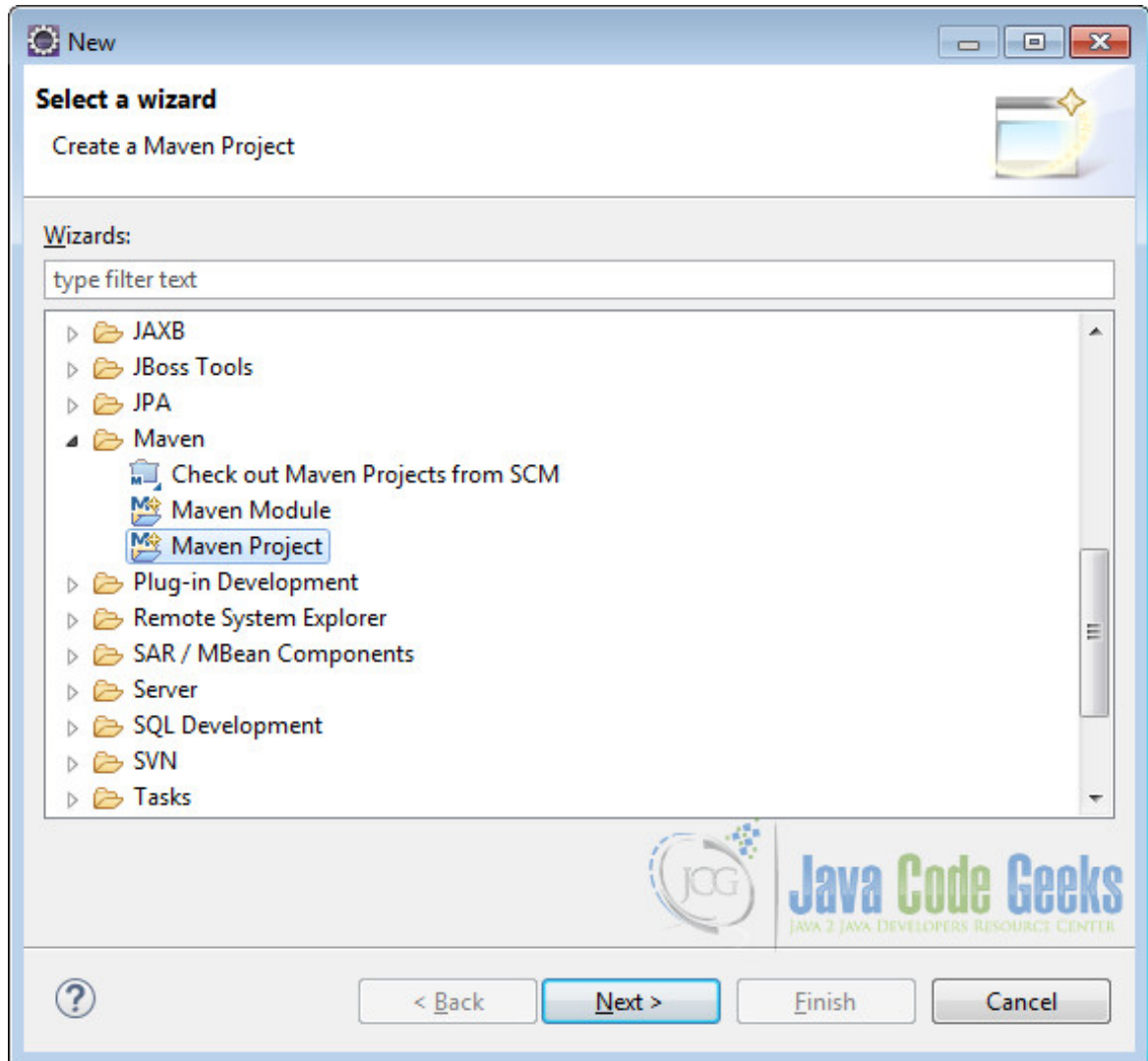


Figure 18.1: Maven setup - step 1

In the “Select project name and location” page of the wizard, make sure that “Create a simple project (skip archetype selection)” option is **unchecked**, hit “Next” to continue with default values.

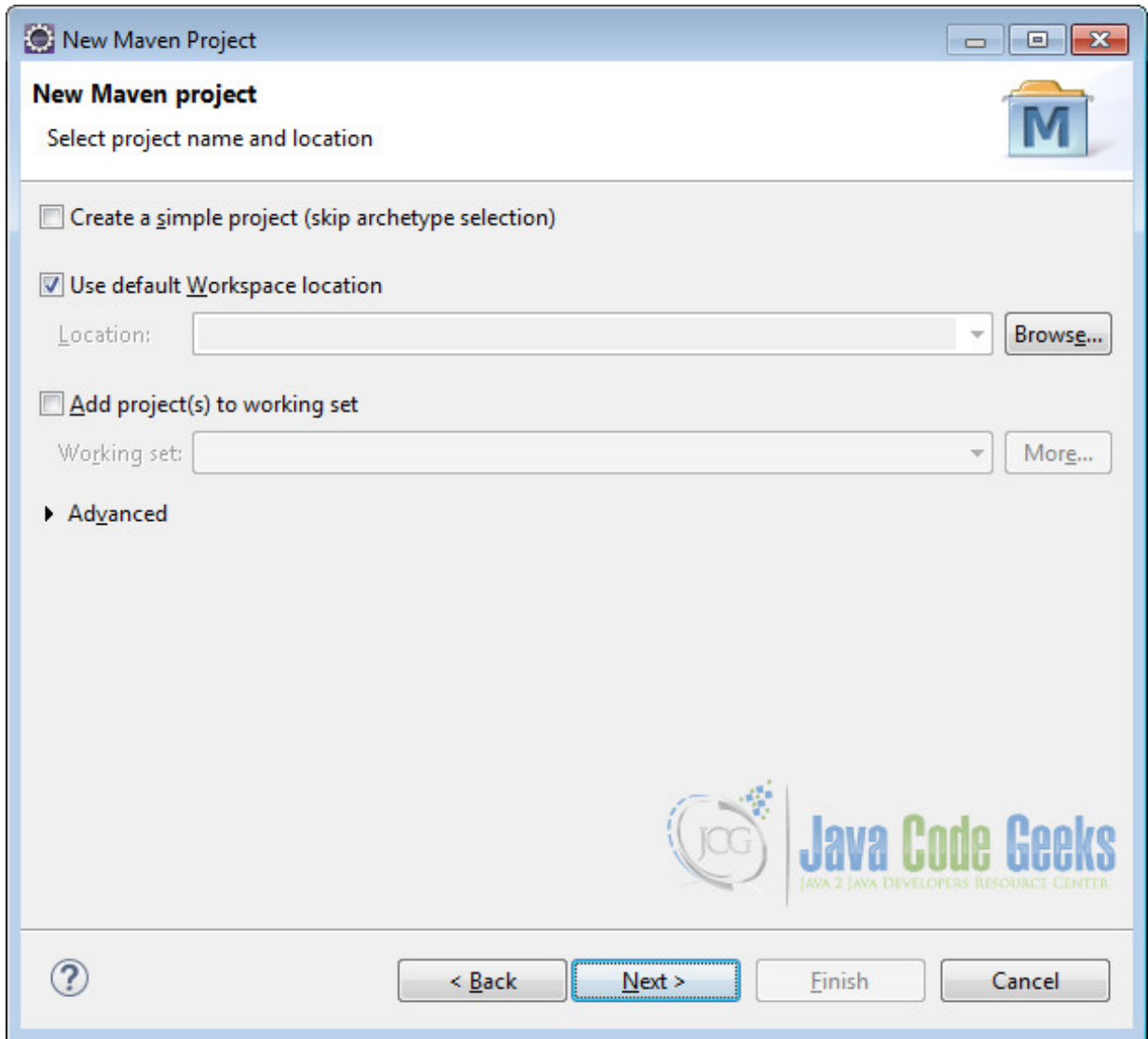


Figure 18.2: Maven setup - step 2

Here choose “maven-archetype-webapp” and click on Next.



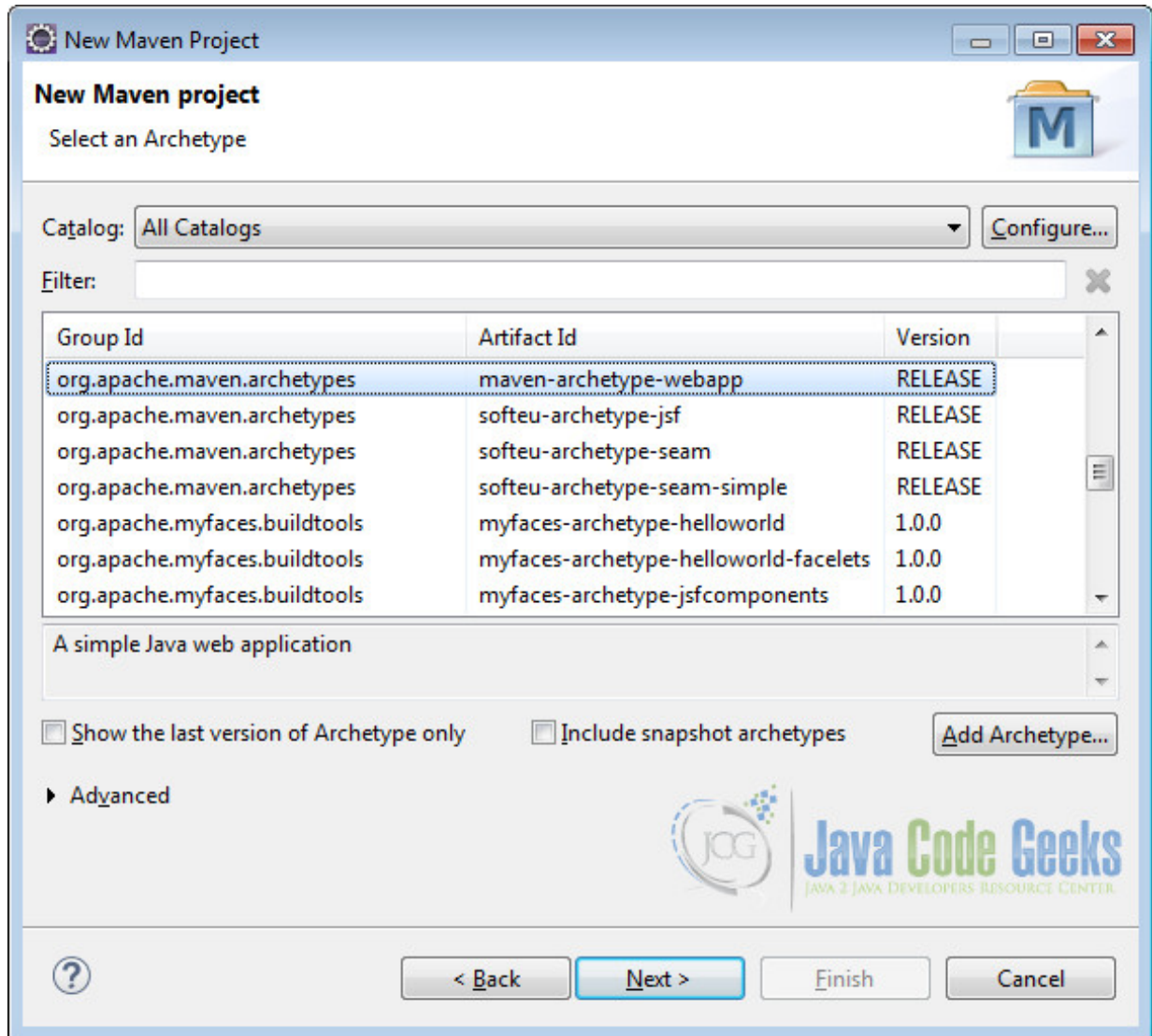


Figure 18.3: Maven setup - step 3

In the “Enter an artifact id” page of the wizard, you can define the name and main package of your project. Set the “Group Id” variable to "com.javacodegeeks.snippets.enterprise" and the “Artifact Id” variable to "jsfcompleters". For package enter "com.javacodegeeks.jsfcompleters" and hit “Finish” to exit the wizard and to create your project.

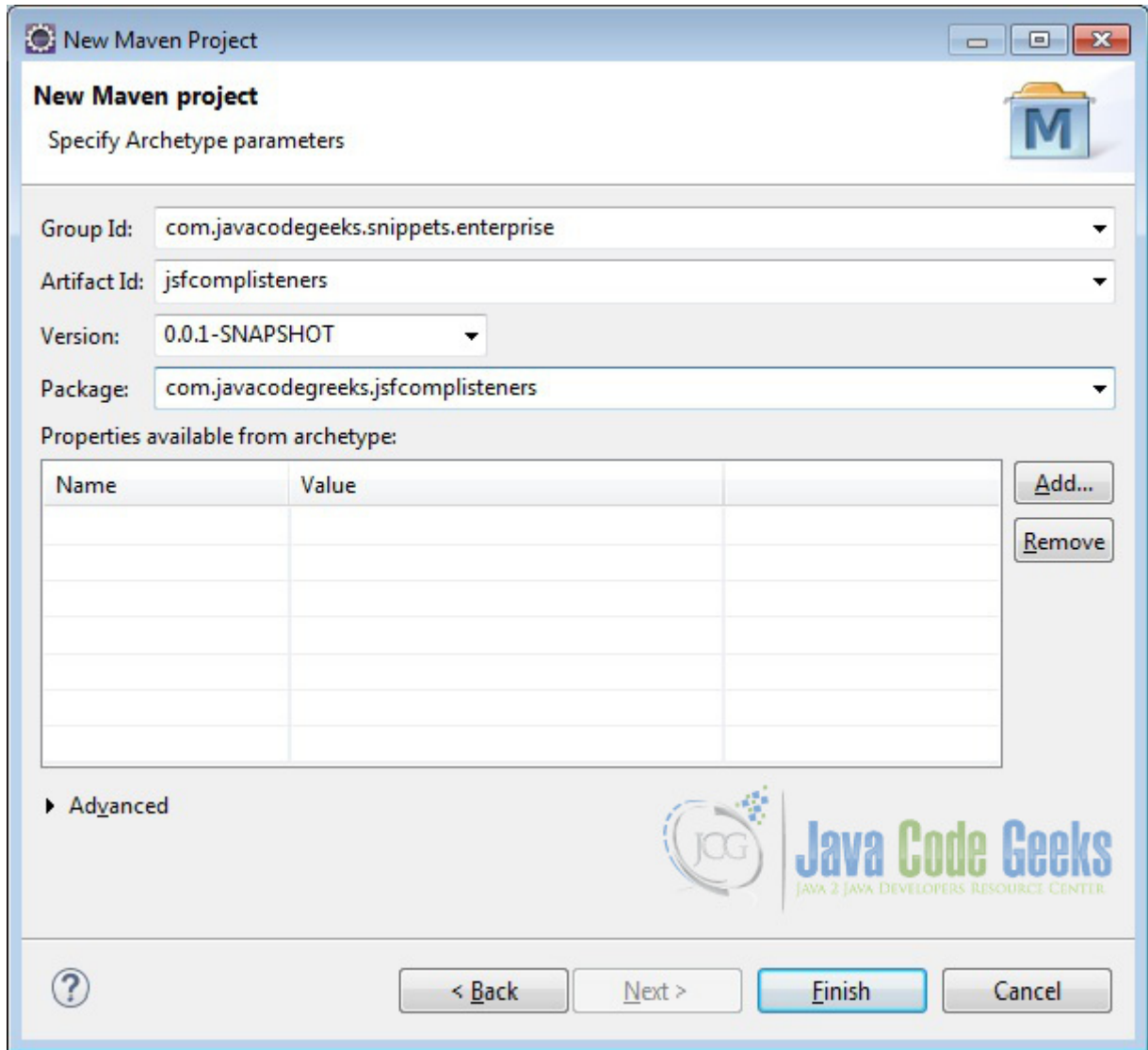


Figure 18.4: Maven setup - step 4

Now create a folder called `java` under `src/main`.

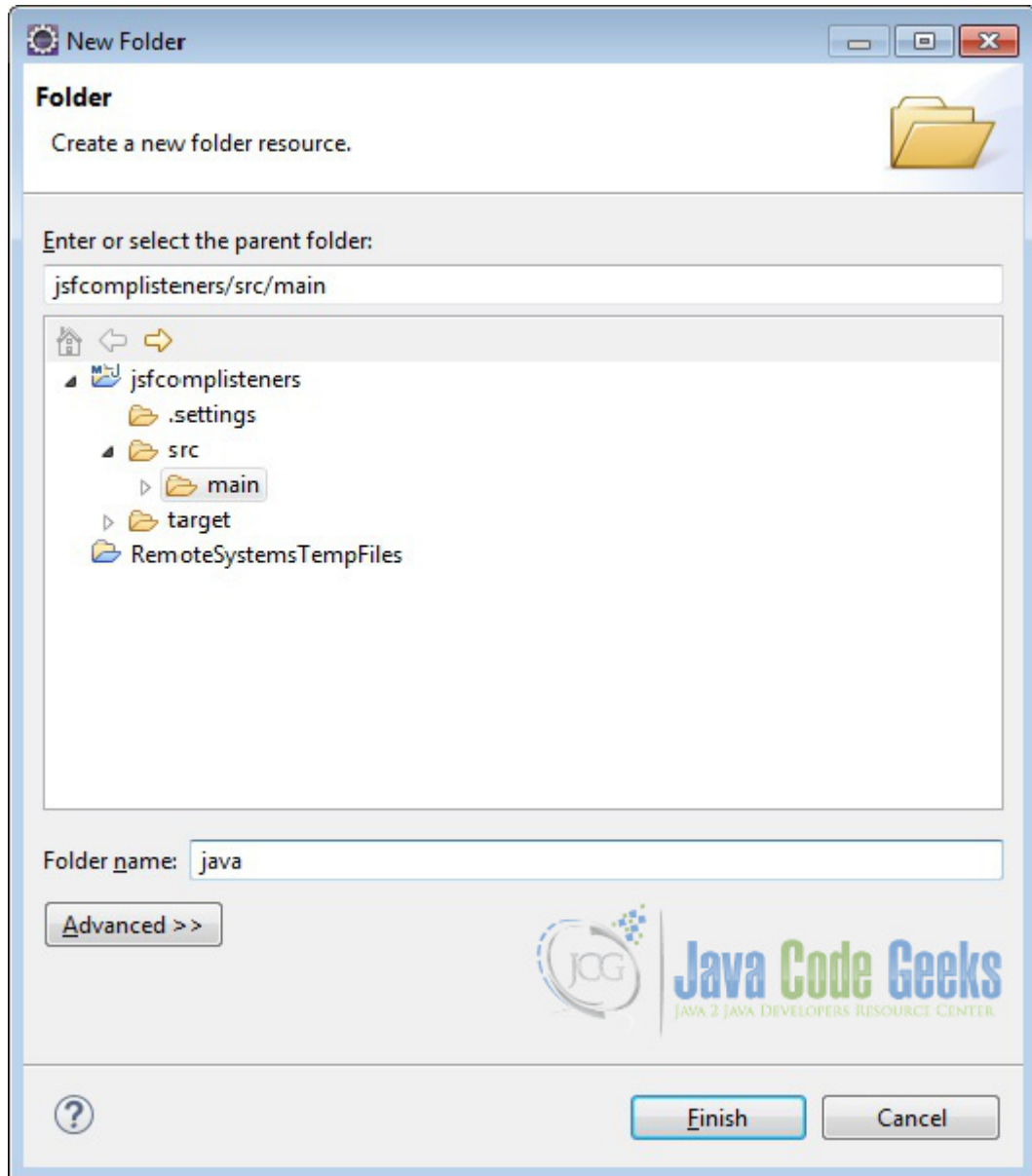


Figure 18.5: Maven setup - step 5

Refresh the project. Finally, the project structure will look like the below.

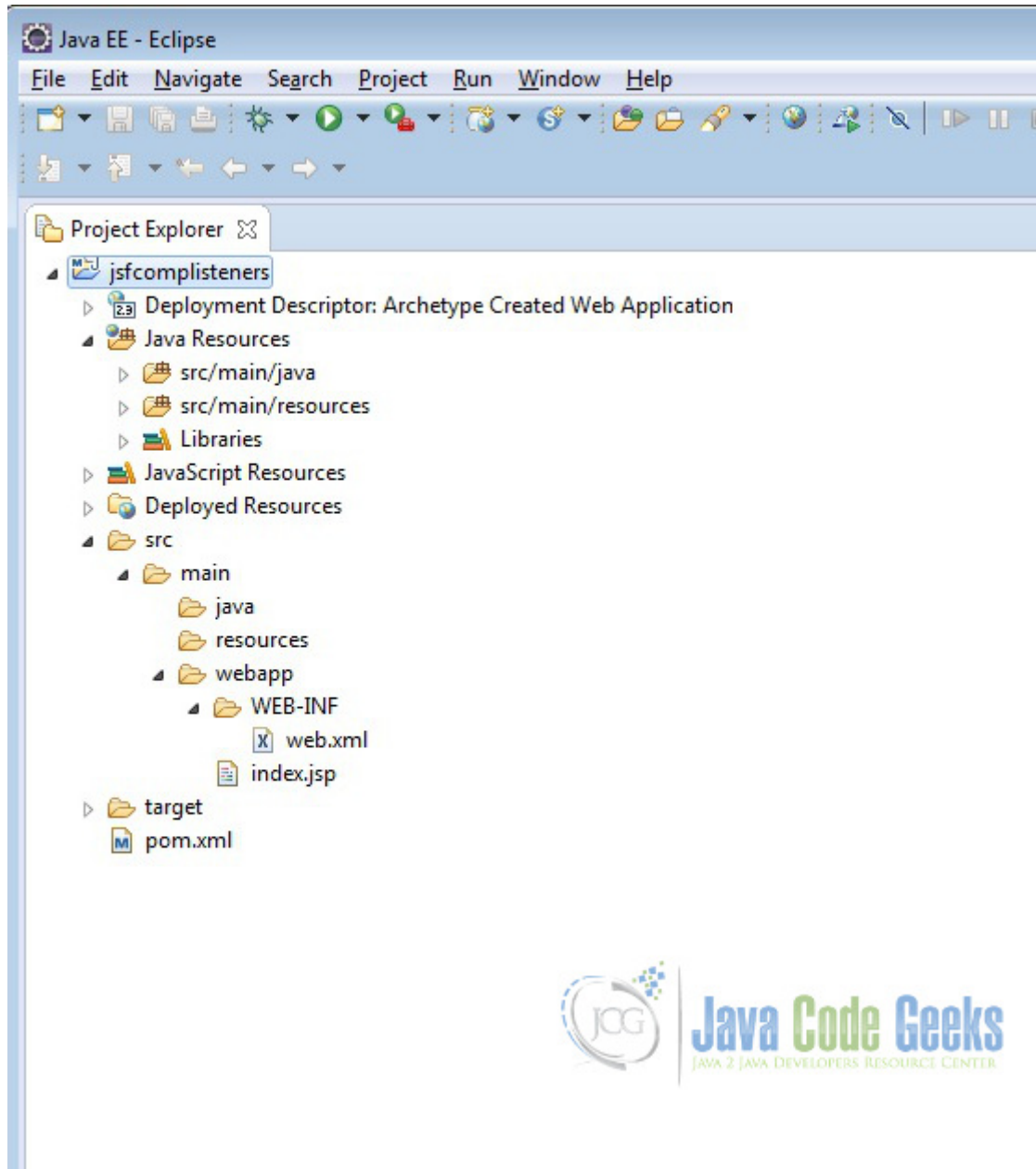


Figure 18.6: Maven setup - step 6

If you see any errors in the `index.jsp`, set target runtime for the project.

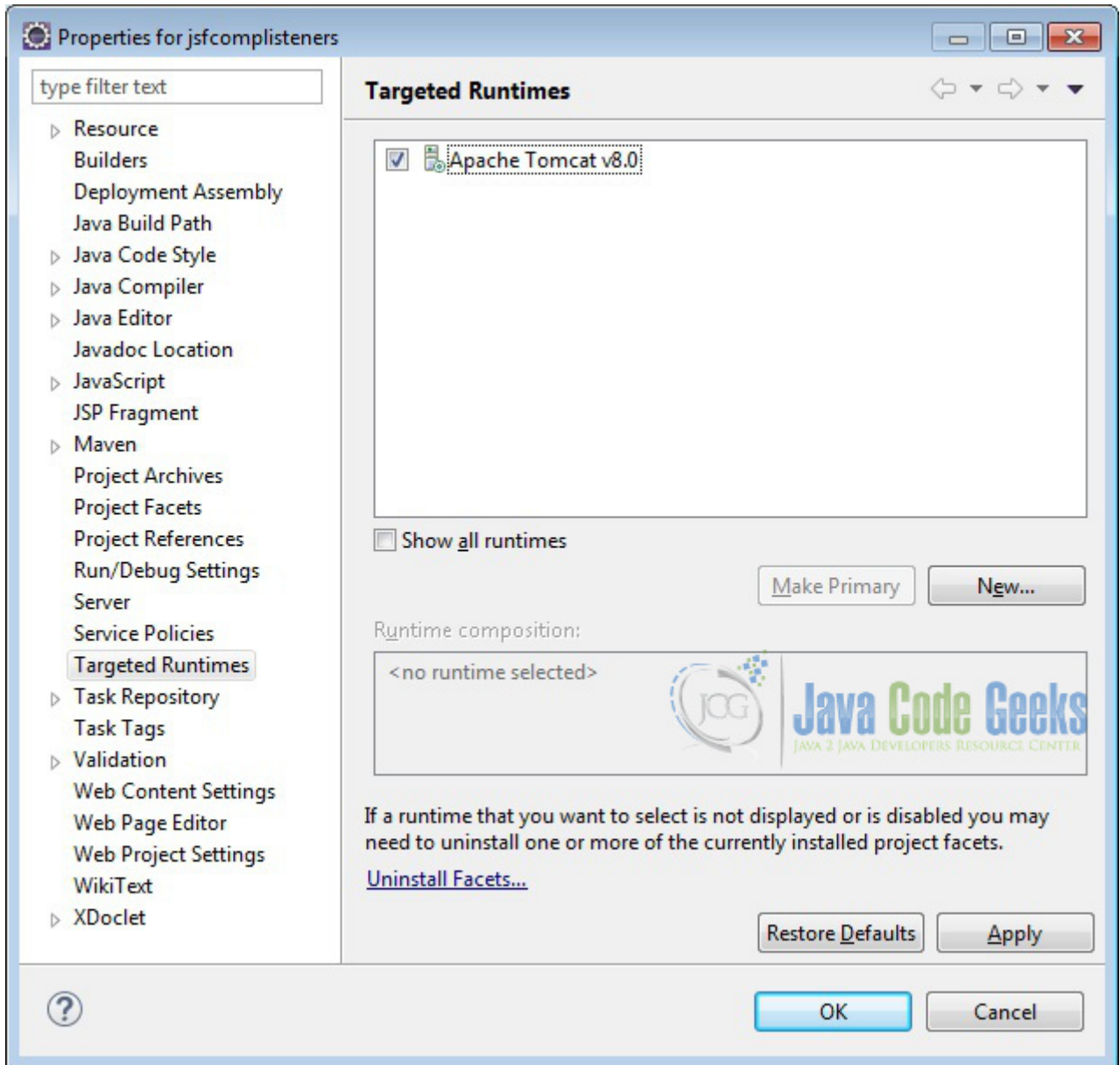


Figure 18.7: Maven setup - step 7

## 18.2 Modify POM to include JSF dependency

Add the dependencies in Maven's pom.xml file, by editing it at the "Pom.xml" page of the POM editor.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0
.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.javacodegreeks.snippets.enterprise</groupId>
<artifactId>jsfcomplisters</artifactId>
<packaging>war</packaging>
<version>0.0.1-SNAPSHOT</version>
<name>jsfcomplisters Maven Webapp</name>
```

```

<url>http://maven.apache.org</url>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.2.9</version>
  </dependency>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>2.2.9</version>
  </dependency>
</dependencies>
<build>
<finalName>jsfcomplisters</finalName>
</build>
</project>

```

### 18.3 Add Faces Servlet in web.xml

The web.xml file has to be modified to include the faces servlet configuration as below.

```

<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
<display-name>Archetype Created Web Application</display-name>
<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>
</web-app>

```

### 18.4 Value Change Listener

JSF input components fires value-change-event when the user interacts with them. The input components such as `h:inputText` or `h:selectOneMenu` fire the value-change-event event on modifying the component values. JSF provides two mechanism for implementing the listeners. We will show you how to implement value change listener on `h:selectOneMenu` by using both the techniques.

First, lets create a package called `com.javacodegeeks.jsfcomplisters` under the folder `src/main/java`. Now we create a managed bean called `JavaVersion`. The `@ManagedBean` annotation makes the POJO as managed bean. The `@SessionScoped` annotation on the bean makes the bean available to the entire user session. We will use `java.util.LinkedHashMap` to store the Java versions released along with the date of release.

JavaVersion.java

```
package com.javacodegeeks.jsfcomplisters;

import java.util.LinkedHashMap;
import java.util.Map;

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

@ManagedBean(name = "javaVersion", eager = true)
@SessionScoped
public class JavaVersion {

    private static Map<String, String> versionData;
    private String releaseDate = "January 23, 1996";

    static {
        versionData = new LinkedHashMap<String, String>();
        versionData.put("JDK 1.0", "January 23, 1996");
        versionData.put("JDK 1.1", "February 19, 1997");
        versionData.put("J2SE 1.2", "December 8, 1998");
        versionData.put("J2SE 1.3", "May 8, 2000");
        versionData.put("J2SE 1.4", "February 6, 2002");
        versionData.put("J2SE 5.0", "September 30, 2004");
        versionData.put("Java SE 6", "December 11, 2006");
        versionData.put("Java SE 7", "July 28, 2011");
        versionData.put("Java SE 8", "March 18, 2014");
    }

    public Map<String, String> getVersionData() {
        return versionData;
    }

    public void setVersionData(Map<String, String> versionData) {
        JavaVersion.versionData = versionData;
    }

    public String getReleaseDate() {
        return releaseDate;
    }

    public void setReleaseDate(String releaseDate) {
        this.releaseDate = releaseDate;
    }
}
```

### 18.4.1 4.1 Using valueChangeListener attribute

To use the UI component valueChangeListener attribute technique, we need to first create a bean method. Let's modify the JavaVersion backing bean to include the listener method.

JavaVersion.java

```
package com.javacodegeeks.jsfcomplisters;

import java.util.LinkedHashMap;
import java.util.Map;

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



```

import javax.faces.event.ValueChangeEvent;

@ManagedBean(name = "javaVersion", eager = true)
@SessionScoped
public class JavaVersion {

    private static Map<String, String> versionData;
    private String releaseDate = "January 23, 1996";

    static {
        versionData = new LinkedHashMap<String, String>();
        versionData.put("JDK 1.0", "January 23, 1996");
        versionData.put("JDK 1.1", "February 19, 1997");
        versionData.put("J2SE 1.2", "December 8, 1998");
        versionData.put("J2SE 1.3", "May 8, 2000");
        versionData.put("J2SE 1.4", "February 6, 2002");
        versionData.put("J2SE 5.0", "September 30, 2004");
        versionData.put("Java SE 6", "December 11, 2006");
        versionData.put("Java SE 7", "July 28, 2011");
        versionData.put("Java SE 8", "March 18, 2014");
    }

    public Map<String, String> getVersionData() {
        return versionData;
    }

    public void setVersionData(Map<String, String> versionData) {
        JavaVersion.versionData = versionData;
    }

    public String getReleaseDate() {
        return releaseDate;
    }

    public void setReleaseDate(String releaseDate) {
        this.releaseDate = releaseDate;
    }

    public void versionChanged(ValueChangeEvent event) {
        releaseDate = event.getNewValue().toString();
    }
}

```

Now we will create a view called `attrlistener.xhtml` under `/src/main/webapp`. We have used `h:selectOneMenu` to display various Java releases and used `h:outputText` to display the release date. We will use the `valueChangeListener` attribute of `h:selectOneMenu` to invoke the bean method.

#### attrlistener.xhtml

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Listener Attribute</title>
</head>
<body>
<h:form>

```



```
<h3>Using valueChangeListener attribute</h3>
<div>
<strong>Selected Version : </strong>
<h:selectOneMenu value="#{javaVersion.releaseDate}"
onchange="submit()"
valueChangeListener="#{javaVersion.versionChanged}">
<f:selectItems value="#{javaVersion.versionData}" />
</h:selectOneMenu>

<br />

<strong>Release Date : </strong>
<h:outputText value="#{javaVersion.releaseDate}" />

</div>
</h:form>
</body>
</html>
```

Now we can create the deployment package using Run as → Maven clean and then Run as → Maven install. This will create a war file in the target folder. The war file produced must be placed in webapps folder of tomcat. Now we can start the server.

Open the following URL in the browser.

<http://localhost:8080/jsfcomplisters/attrlistener.xhtml>



Figure 18.8: valueChangeListener - Attribute

Modify the Java version using the drop down. The release date will get changed accordingly.

#### 18.4.2 4.2 Using valueChangeListener Tag

To use the valueChangeListener Tag, we need to first create a class implementing the ValueChangeListener interface. Lets first create a class called VersionChangeListener in the package com.javacodegeeks.jsfcomplisters and

implement the `processValueChange` method of the interface. We will use the `FacesContext` to get the `JavaVersion` object and set the release date using the `ValueChangeEvent`.

#### VersionChangeListener.java

```
package com.javacodegeeks.jsfcomplisters;

import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ValueChangeEvent;
import javax.faces.event.ValueChangeListener;

public class VersionChangeListener implements ValueChangeListener{

    public void processValueChange(ValueChangeEvent event)
        throws AbortProcessingException {
        JavaVersion javaVersion= (JavaVersion) FacesContext.getCurrentInstance().
            getExternalContext().getSessionMap().get("javaVersion");
        javaVersion.setReleaseDate(event.getNewValue().toString());

    }

}
```

We will create a view called `taglistener.xhtml` under `/src/main/webapp`. We have used `h:selectOneMenu` to display various Java releases and used `h:outputText` to display release date. But instead of using the `valueChangeListener` attribute, we are using the tag `f:valueChangeListener` this time. The type attribute of the tag reference to the fully qualified name of the listener which is `"com.javacodegeeks.jsfcomplisters.VersionChangeListener"` in our case.

#### taglistener.xhtml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Listener Tag</title>
</head>
<body>
    <h:form>

        <h3>Using valueChangeListener Tag</h3>
        <div>
            <strong>Selected Version : </strong>
            <h:selectOneMenu value="#{javaVersion.releaseDate}"
                onchange="submit()" >
                <f:valueChangeListener
                    type="com.javacodegeeks.jsfcomplisters.VersionChangeListener" />
                <f:selectItems value="#{javaVersion.versionData}" />
            </h:selectOneMenu>

            <br />

            <strong>Release Date : </strong>
            <h:outputText value="#{javaVersion.releaseDate}" />

        </div>
    </h:form>
</body>
```

```
</html>
```

Now again package using maven and copy the war to the apache tomcat webapps folder. Open the following URL in the browser.

```
http://localhost:8080/jsfcomplisters/taglistener.xhtml
```



Figure 18.9: valueChangeListener - Tag

Modify the Java version using the drop down. The release date will get changed accordingly.

## 18.5 Action Listener

JSF UI components fire the action-event when the user clicks on them. The components such as `h:commandButton` fire the event on click of it. Similar to `valueChangeListener`, `actionListener` can also be implemented in two techniques. We will show you how to implement action listener on `h:commandButton` using both the techniques.

Now, we create a managed bean called `UserProfile` under the package `com.javacodegeeks.jsfcomplisters`. The `@ManagedBean` annotation makes the POJO as managed bean. The `@SessionScoped` annotation on the bean makes the bean available to the entire user session. We will use the method `updateGreeting` to modify the greeting with current day. A utility method called `getDayUtil` is also provided to convert the day into user readable value.

`UserProfile.java`

```
package com.javacodegeeks.jsfcomplisters;

import java.util.Calendar;

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

@ManagedBean(name = "userProfile", eager = true)
@SessionScoped
```

```
public class UserProfile {

    private String label = "Click here for Today's Greeting ";
    private String greeting = "Hello, have a nice";

    public String getGreeting() {
        return greeting;
    }

    public String getLabel() {
        return label;
    }

    public void setLabel(String label) {
        this.label = label;
    }

    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }

    public void updateGreeting(ActionEvent event) {
        greeting = greeting + " "
            + getDayUtil(Calendar.getInstance().get(Calendar.DAY_OF_WEEK))
            + "!!";
    }

    private String getDayUtil(int day) {
        String dayOfWeek = "Sunday";
        switch (day) {
            case 1:
                dayOfWeek = "Sunday";
                break;
            case 2:
                dayOfWeek = "Monday";
                break;
            case 3:
                dayOfWeek = "Tuesday";
                break;
            case 4:
                dayOfWeek = "Wednesday";
                break;
            case 5:
                dayOfWeek = "Thursday";
                break;
            case 6:
                dayOfWeek = "Friday";
                break;
            case 7:
                dayOfWeek = "Saturday";
                break;
        }
        return dayOfWeek;
    }
}
```

### 18.5.1 5.1 Using ActionListener attribute

To use the UI component `actionListener` attribute technique, we will use the backing bean method `updateGreeting` of `UserProfile`.

Now lets modify the view `attrlistener.xhtml` to include `h:outputText` for creating label and `h:commandButton` for command button. The `actionListener` attribute of `h:commandButton` is used to invoke the backing bean method.

`attrlistener.xhtml`

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Listener Attribute</title>
</head>
<body>
<h:form>

<h3>Using valueChangeListener attribute</h3>
<div>
<strong>Selected Version : </strong>
<h:selectOneMenu value="#{javaVersion.releaseDate}"
onchange="submit()"
valueChangeListener="#{javaVersion.versionChanged}">
<f:selectItems value="#{javaVersion.versionData}" />
</h:selectOneMenu>

<br />

<strong>Release Date : </strong>
<h:outputText value="#{javaVersion.releaseDate}" />

</div>
<hr></hr>

<h3>Using ActionListener attribute</h3>
<div>
<h:outputText value="#{userProfile.label}"></h:outputText>
<h:commandButton id="submit" value="Submit" action="greeting"
actionListener="#{userProfile.updateGreeting}" />

</div>
</h:form>
</body>
</html>
```

Now create another view called `greeting.xhtml` under `/src/main/webapp`. By means of Implicit navigation the action value of the `commandButton` in `attrlistener.xhtml` will get resolved to `greeting.xhtml`. This page is a simple page to display the updated greeting.

`greeting.xhtml`

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core">
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Result</title>
</head>
<body>
  <h:form>

    <h3>Action Listener Result Page</h3>
    <div>
      <h:outputText value="#{userProfile.greeting}" />

    </div>
  </h:form>
</body>
</html>
```

Now again package using maven and copy the war to the apache tomcat webapps folder. Open the following URL in the browser.

<http://localhost:8080/jsfcomplisters/attrlistener.xhtml>



Figure 18.10: actionListener - Attribute

Now click on the submit button.

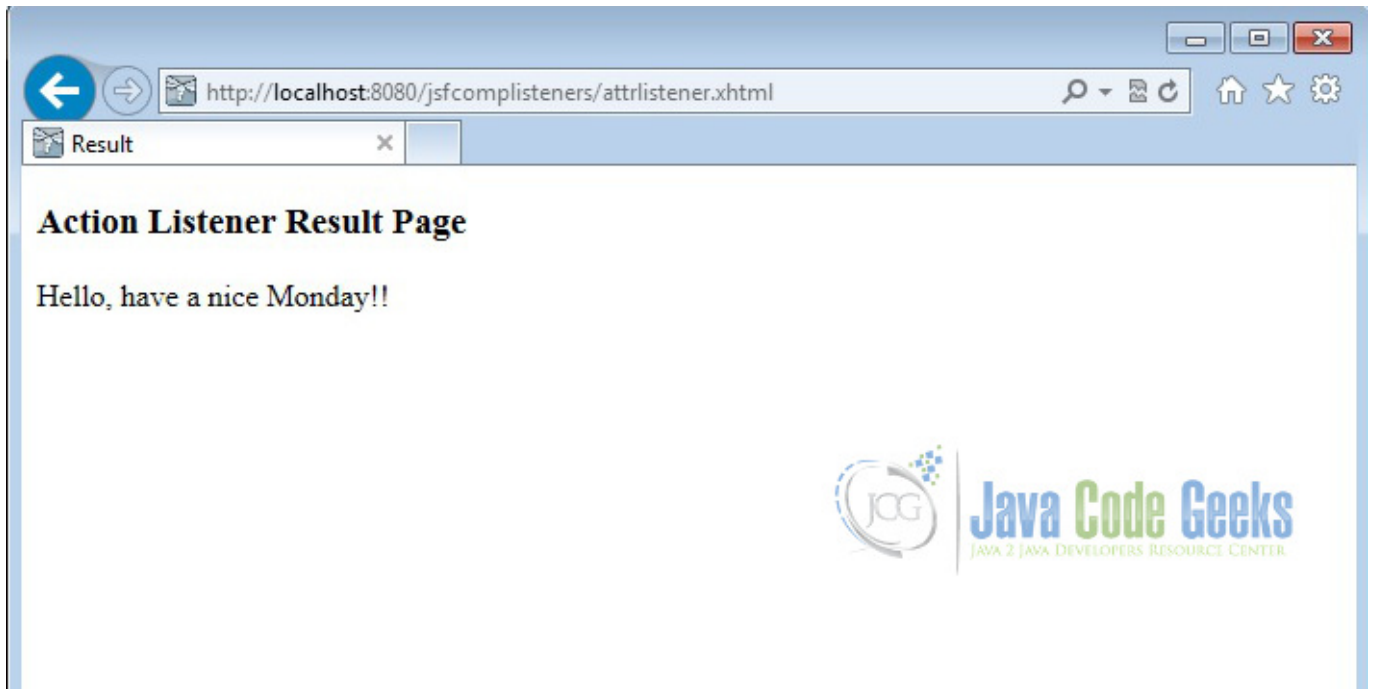


Figure 18.11: actionListener - Attribute Result

### 18.5.2 5.2 Using actionListener Tag

To use the `actionListener` Tag, we need to first create a class implementing the `ActionListener` interface. Lets first create a class called `UserProfileActionListener` in the package `com.javacodegeeks.jsfcomplisters` which implement the method `processAction` of the interface. We will use the `FacesContext` to get the `UserProfile` object and update the greeting using `ActionEvent`.

`UserProfileActionListener.java`

```
package com.javacodegeeks.jsfcomplisters;

import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;

public class UserProfileActionListener implements ActionListener {

    public void processAction(ActionEvent event)
        throws AbortProcessingException {
        UserProfile userProfile = (UserProfile) FacesContext
            .getCurrentInstance().getExternalContext().getSessionMap()
            .get("userProfile");
        userProfile.updateGreeting(event);
    }
}
```

Now let us modify the view `taglistener.xhtml` to include `h:outputText` for creating label and `h:commandButton` for command button. We will use the `f:actionListener` tag and reference the `type` attribute to the fully qualified name of the class `com.javacodegeeks.jsfcomplisters.UserProfileActionListener`.

`taglistener.xhtml`

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!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://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Listener Tag</title>
</head>
<body>
  <h:form>

    <h3>Using valueChangeListener Tag</h3>
    <div>
      <strong>Selected Version : </strong>
      <h:selectOneMenu value="#{javaVersion.releaseDate}"
        onchange="submit()" >
      <f:valueChangeListener
        type="com.javacodegeeks.jsfcomplisters.VersionChangeListener" />
      <f:selectItems value="#{javaVersion.versionData}" />
      </h:selectOneMenu>

      <br />

      <strong>Release Date : </strong>
      <h:outputText value="#{javaVersion.releaseDate}" />

    </div>
    <hr></hr>

    <h3>Using actionListener Tag</h3>
    <div>
      <h:outputText value="#{userProfile.label}"></h:outputText>
      <h:commandButton id="submit" value="Submit" action="greeting">
      <f:actionListener
        type="com.javacodegeeks.jsfcomplisters.UserProfileActionListener" />
      </h:commandButton>

    </div>
  </h:form>
</body>
</html>
```

Now again package using maven and copy the war to the apache tomcat webapps folder. Open the following URL in the browser

<http://localhost:8080/jsfcomplisters/taglistener.xhtml>





Figure 18.12: actionListener - Tag

Now Click on the Submit button.

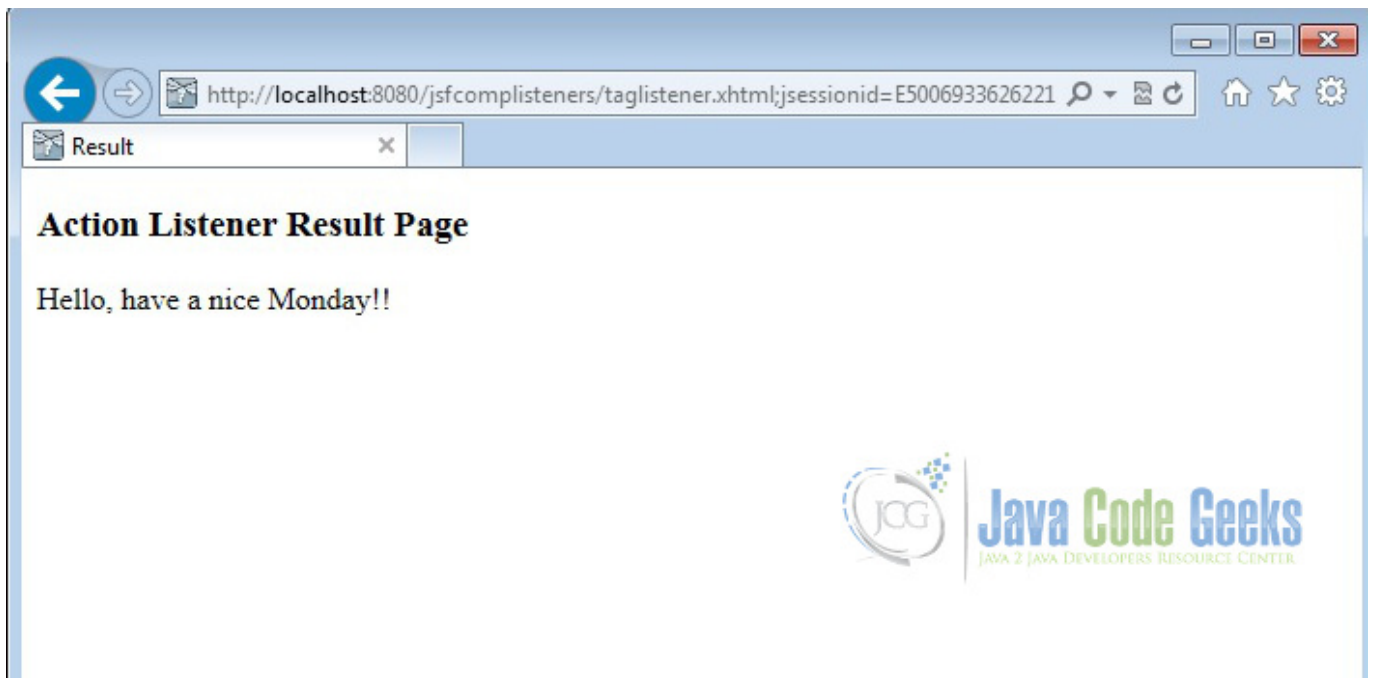


Figure 18.13: actionListener - Tag Result

## 18.6 Download the Eclipse Project

This was an example of how to use Java Server Faces Component Listeners.

**Download** You can download the full source code of this example here : [JSF Components Listeners](#)

---