# Práctica 1: Programación de Servicio Web RESTful con EJB y ORM

Desarrollo y despliegue de componentes software - I

| | |
|---|---|
| Comienzo de la práctica | 10-10-2023 |
| Fecha de entrega recomendada | 24-10-2023 |

## 1 JSF introduction

Java Server Faces (JSF) is a framework that helps the development of Web applications following the MVC architectural *pseudo-pattern* ("Model View Controller"). Simplifies the construction of user interfaces (UIs) by using software components. One JSF-page provides facilities to connect *widgets* [1] a UI (which we call *componentIU* from now on) with the sources of data (or " resources "), according to the component model studied in lectures and with event handlers (or " services ") that are located on the server side.

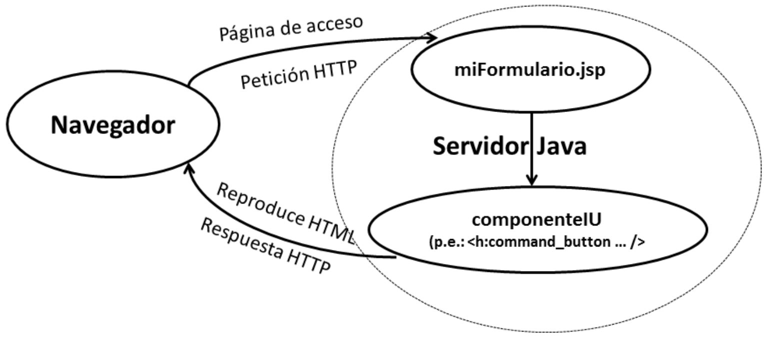


Figure 1: Architecture of an application that uses JSF technology

In figure 1, it can be seen the page `miFormulario.jsp`, which graphically displays the UI components by using custom tags defined by JSF technology. The *componenteIU* of the figure above is a component of the Web application that acts on the "objects" referenced by the `jsp` form.

The JSP page manages all the objects involved in the web application, such as the following ones:

- component-objects (labels of type `<h:command_button .../>`,
- *listeners* of actions that are launched form the component-objects, such as *validators* y *conversors*, that are usually included in component labels (e.g., `<f:validate _longrange minimum="0" maximum="10"/>` ),
- the '*objects*' (*beans*) of the model containing the Web application data,
- the rest of the functionality of the application components.

With JSF technology we can connect events generated in the client with the code of the Web application on the server side. Also map[2] the *componentesIU* with the objects in the server's side.

[1] widget is a component of an interface, which allows the user to perform a function or access a service

[2] i.e., that each one "understand" the other

We can build a *componenteIU* by using extensible and reusable software components. As well as saving and later restoring the state of the UI of our application after the requests to the server have been completed. Additionally, to use JSF technology offers us a series of important benefits that make programming Web applications easier, such as, for example, offering a clear separation between the behavior of software components and their representation within a Web page. It also improves some of the shortcomings of JSP technology, see figure 2, since with this technology we cannot make requests *http* to the server event handler, or handle UI-components as if they were objects.

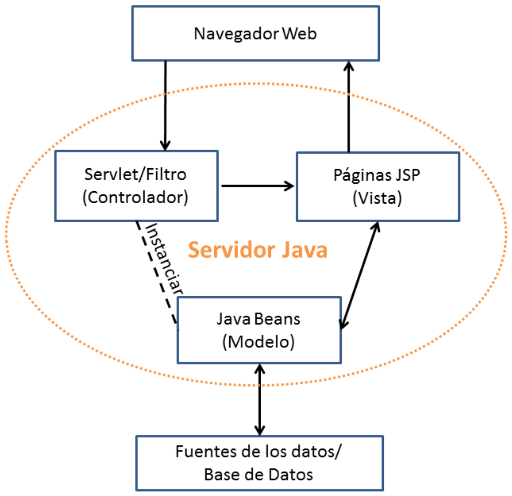


Figure 2: MVC model of one application that uses JSP pages

JSF includes a library of custom JSP tags for rendering components on a web page. And therefore, authors of Web pages without much programming experience can use these tags to include code in applications, which are usually located in the server part, and embed them in their Web pages, without having to do it by programming with a markup language (Javascript, or similar). JSF technology APIs (they are really several) have been created directly on top of the so-called *Servlet*-API of Java in such a way that if we use JSF, we can also use other Web page presentation technologies as well, in order to program the client part of our Web applications.

## 1.1 Maven: how to package “ components ” for deployment in a Web applications container

Maven is a Yiddish word that means “knowledge accumulator”. It is a standard way of building Java-based projects with capabilities to generate *software components* that can be deployed in specific containers of Web applications, such as Apache Tomcat; more information about Web Tools Platform (WTP) can be seen at`https://eclipse.org/webtools/`). With Maven we can develop *software components* for our Web applications as long as we guarantee the following conditions:

- We have a clear definition of what a project contains (programmed in Java, in this case)
- We will use an easy way to publish this information (by the file `web.xml` creation, for example)
- We can always share files in JAR format between several projects

## 1.2 Managed beans

A Managed Bean (MB) is a special class of the Java programming language (named *bean*) but registered within the JSF framework. In other words, **one MB is a *Java Bean* managed by JSF**.

Recall that the concept of *Java Bean* refers to a class written in Java that encapsulates many objects in a single entity, which is the *bean*, and that it is intended for deployment in a distributed execution environment. Therefore, Java *Beans* technology aims at creating a model of reusable software components by using the Java language as programming support. The *beans* are serializable and, therefore, have a constructor method without arguments and allow us to access to their *attributes* from our programs, according to the object-oriented terminology we are used to, by using the methods *getter()* y *setter()*.

Therefore, one MB contains the following elements:

- methods `getter()` and `setter()`
- The "business logic" of the component, that is, all the information, associated with a *bean*, and contained in application HTML forms

One MB acts as the "Model", according to the MVC pattern, applied to build Web applications, which uses JSF technology and does so for each *componenteUI* of our application. The corresponding files are Java-classes that will be located (for now) inside the package `prueba` in the subdirectory `src/main/java` of the main source of the component that we are going to program.

```java
package prueba;
import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean(name="holaMundo", eager=true)
@RequestScoped
public class HolaMundo implements Serializable {
  private static final long serialVersionUID = 1L;

  public HolaMundo() {
        System.out.println("HolaMundo␣ha␣comenzado!!!");
  }
  public String getMensaje() {
        return "Hola␣Mundo!!!";
  }
}
```

## 1.3 The controller

**FacesServlet**[3] initializes all the JSF components of the application we have developed, before it starts to run; that is, before the JSF framework is fully 'deployed' on our runtime platform. We can say that *FacesServlet* is the main *servlet* responsible for handling all requests from the client part of a Web application towards the application components located on the server.

*FacesServlet* acts as the central controller, according to the MVC model we are following when programming with this framework; first, it initializes all the JSF components, before the JSP page that includes the form is displayed (see figure 1) on the client Web-browser. *FacesServlet* is located in the folder  tt webapp/WEB_INF of the component's code that we are programming.

[3]It is like a Java *applet* that runs on the server instead of on the client machine that opens the browser

### 1.4 The view

Usually, the 'view' (from the MVC) becomes a page encoded with XHTML [4] in our programming scheme, which takes the filename `home.xhtml` or `index.xhtml` and is located in the subdirectory `DeployedResources/WebContent` from the component's Java project.

```
<!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>
<title>
Ejemplo JSF!
</title>
</head>
<body>
<p>#{holaMundo}</p>
#{holaMundo.mensaje}
</body>
</html>
```

## 2 Setting up the practial assignment context

Before even starting to program the *managed bean* in the IDE of your choice [5], double check that the Java installation includes a JDK $\geq$ 1.8, then we will download and configure Maven and finally we would install Apache Tomcat to try to run a "hello world" from the correctly configured IDE.
**Alternatively, we could do everything in the Eclipse IDE**, since if we have installed *Eclipse IDE for Enterprise Java Developers* we can select the type *Dynamic Web Project* to create the initial project *HelloWorld*(Target Runtime: Apache Tomcat v.x and the Dynamic Web Module Version: *latest version*), with the following folder configuration:

```
HolaMundo
  \DeploymentDescriptor: HolaMundo
  \JAX-WS Web Services
  \Java Resources
     \src
     \libraries
  \JavaScript Resources
  \WebContent
     \META-INF
        MANIFEST.MF
     \WEB-INF
        \lib
```

[4]Extensible Hypertext Markup Language (XHTML) belongs to the family of markup languages XML markup languages, it extends to the HTML language for writing pages; XHTML can be explored using XML parsers, unlike HTML that needs a specific browser for this code

[5]it is recommended to use Eclipse for this practical assignment

## 2.1 Download and configure Maven

One of the following compressed files can be downloaded from the Maven site at Apache Software Foundation (ASF), depending on the operating system on which will be used:

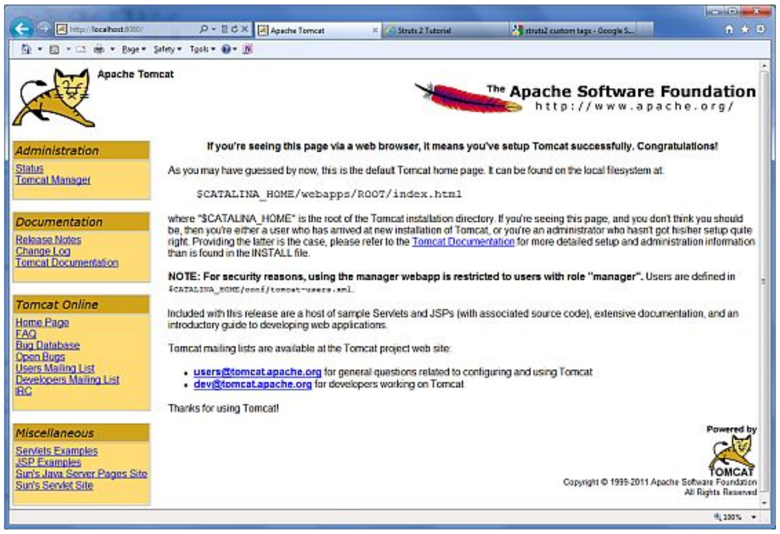
```
Windows: apache-maven-x.y.z-bin.zip
Linux: apache-maven-x.y.z-bin.tar.gz
Mac:  apache-maven-x.y.z-bin.tar.gz
```

Extract the file in the subdirectory where you want to locate this software.

The subdirectory: `apache-maven-x.y.z` will then be created, from the extracted file. The following environment variables must be added to the execution platform: `M2_HOME` and `MAVEN_OPTS (-Xms256m -Xmx512m)`. For example, on Linux, to adjust environment variables, using a window of *bash*, we will type the following commands:

```
export M2_HOME=/usr/local/apache-maven/apache-maven-3.2.3
export M2=%M2_HOME%/bin
export MAVEN_OPTS=-Xms256m -Xmx512m
```

Finally, add the environment variable `M2` to the `PATH` ( by typing `export PATH=`$M2$` : PATH`) and verify that the Maven installation was successful: `mvn -version`. **Alternatively, we could do it directly from the Eclipse IDE starting from a** *Dynamic Web Project*, right-clicking on the project name: *Configure ⇒ Convert to Maven Project*, which adds a small `pom.xml` in the *WebContent* subdirectory of the project.



**If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!**

As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:

```
$CATALINA_HOME/webapps/ROOT/index.html
```

where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the Tomcat Documentation for more detailed setup and administration information than is found in the INSTALL file.

**NOTE: For security reasons, using the manager webapp is restricted to users with role "manager".** Users are defined in `$CATALINA_HOME/conf/tomcat-users.xml`.

Included with this release are a host of sample Servlets and JSPs (with associated source code), extensive documentation, and an introductory guide to developing web applications.

Tomcat mailing lists are available at the Tomcat project web site:

- users@tomcat.apache.org for general questions related to configuring and using Tomcat
- dev@tomcat.apache.org for developers working on Tomcat

Thanks for using Tomcat!

Copyright © 1999-2011 Apache Software Foundation
All Rights Reserved

Figure 3: Tomcat main page.

## 2.2 Installing Apache Tomcat

The latest version of Tomcat, an open source Web server for running servlets and other web applications, can be downloaded from `http://tomcat.apache.org/` and once the distribution has been unpacked in the chosen subdirectory, you must assign the environment variable: `CATALINA_HOME`.

Tomcat can be started or stopped with the command files `startup.bat` / `shutdown.bat` in the subdirectory `bin` of the folder where we have the local Tomcat installation. After performing a successful start of Tomcat, the Web applications that we include in the subdirectory `webapps` of Tomcat will be available to your clients by visiting the page: `http:/localhost:8080`.

If we have configured everything correctly, the Tomcat administration start window should appear (Figure 3).

## 2.3 JSF configuration

Notice: The following not to be done if Maven has been configured from the Eclipse IDE (see section 2.0) and it has become a *Maven Project*.

The first thing would be to create a Maven project, which can be done from the command line, `mvn archetype : generate -DgroupId = prueba -DartifactId = holamundo -DarchetypeArtifactId = maven-archetype - webapp`

Now we have to check that we have created a subdirectory structure that includes:

- `holamundo`: which contains the subdirectory `src` and the project object model (POM) file: `pom.xml`, which is an XML representation of the Maven project
- The subdirectory `src/main/webapp` that contains `WEB_INF` and the JSP index page
- The subdirectory `src/main/resources` where the resources (properties, images, etc.) associated with the emphUI component that we want to develop will be located

### 2.3.1 Prepare the project to import it into our IDE

Notice: The following not be done if Maven has been configured from the Eclipse IDE (see section 2.0) and it has become a emph Maven Project.

We have to check that the Maven project has been built well and that we have turned it into a project that can be included in our IDE.

If we use the Eclipse IDE, then we will use the command `mvn eclipse : eclipse -Dwtpversion = 2.0` (inside the subdirectory where `holaMundo` resides to find the configuration file `pom.xml` ) that prepares a project for this IDE and if the application `helloworld` is built correctly, something like the following should appear:

```
Downloading: http://repo.maven.apache.org/org/apache/maven/plugins/
maven-compiler-plugin/2.3.1/maven-compiler-plugin-2.3.1.pom
5K downloaded  (maven-compiler-plugin-2.3.1.pom)
Downloading: http://repo.maven.apache.org/org/apache/maven/plugins/
maven-compiler-plugin/2.3.1/maven-compiler-plugin-2.3.1.jar
29K downloaded  (maven-compiler-plugin-2.3.1.jar)
[INFO] Searching repository for plugin with prefix: 'eclipse'.
[INFO] ------------------------------------------------------------
[INFO] Building holamundo Maven Webapp
[INFO]    task-segment: [eclipse:eclipse]
[INFO] ------------------------------------------------------------
[INFO] Preparing eclipse:eclipse
[INFO] No goals needed for project - skipping
[INFO] [eclipse:eclipse {execution: default-cli}]
[INFO] Adding support for WTP version 2.0.
[INFO] Using Eclipse Workspace: null
[INFO] Adding default classpath container: org.eclipse.jdt.
launching.JRE_CONTAINER
Downloading: http://repo.maven.apache.org/
com/sun/faces/jsf-api/2.1.7/jsf-api-2.1.7.pom
12K downloaded  (jsf-api-2.1.7.pom)
Downloading: http://repo.maven.apache.org/
com/sun/faces/jsf-impl/2.1.7/jsf-impl-2.1.7.pom
10K downloaded  (jsf-impl-2.1.7.pom)
Downloading: http://repo.maven.apache.org/
com/sun/faces/jsf-api/2.1.7/jsf-api-2.1.7.jar
619K downloaded  (jsf-api-2.1.7.jar)
Downloading: http://repo.maven.apache.org/
com/sun/faces/jsf-impl/2.1.7/jsf-impl-2.1.7.jar
```

```
1916K downloaded  (jsf-impl-2.1.7.jar)
[INFO] Wrote settings to D:\Docencia\DSSBCS\wkspace-movil-eclipse\holamundo\.settings\
org.eclipse.jdt.core.prefs
[INFO] Wrote Eclipse project for "holamundo" to D:\Docencia\DSSBCS\wkspace-movil-eclipse\holamundo.
[INFO]
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------
[INFO] Total time: 6 minutes 7 seconds
[INFO] Finished at: Mon Nov 05 16:16:25 IST 2012
[INFO] Final Memory: 10M/89M
[INFO] ---------------
```

To finish, we only have to import the project thus generated in our IDE to be able to start doing this first parctical assignment.

## 2.4 `pom.xml`definition

And now, we will add the capabilities that the JSF framework provides to the project, for which we have to include the following dependencies in the appropriate part of the `POM.xml` file,

```
<dependencies>
 <dependency>
 <groupId>com.sun.faces</groupId>
 <artifactId>jsf-api</artifactId>
 <version>2.1.7</version>
  </dependency>
  <dependency>
 <groupId>com.sun.faces</groupId>
 <artifactId>jsf-impl</artifactId>
 <version>2.1.7</version>
  </dependency>
  <dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.1</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
  </dependency>
</dependencies>
```

Obviously, the version numbers listed above must be replaced by the latest stable versions. You should also include the compiler version of Maven:

```
 <properties>
    <maven.compiler.source>1.6</maven.compiler.source>
    <maven.compiler.target>1.6</maven.compiler.target>
  </properties>
```

And the section to indicate the plugin necessary to generate the drop-down component in `war` format (if it is not done, it would generate it in the `jar` format by default):

```
<build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
          <release>11</release>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.2.3</version>
        <configuration>
          <warSourceDirectory>WebContent</warSourceDirectory>
        </configuration>
      </plugin>
    </plugins>
    <finalName>HolaMundo/finalName>
  </build>
```

## 2.5 servlet configuration

We access our project, correctly imported into the IDE that we are going to use, and we look for the file `web.xml` to check that it presents the following content:

```
<?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"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
id="WebApp_ID" version="2.5">
<welcome-file-list>
<welcome-file>faces/home.xhtml</welcome-file>
 <welcome-file>index.xhtml</welcome-file>
 <welcome-file>index.htm</welcome-file>
 <welcome-file>index.jsp</welcome-file>
 <welcome-file>default.html</welcome-file>
 <welcome-file>default.htm</welcome-file>
 <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
<!--
FacesServlet is a primary servlet responsible for handling all requests.
It acts as a central controller.
This servlet initializes the JSF components before the JSP is displayed.
-->
<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>
</web-app>
```

## 2.6 Managed beans

To carry out this part of the practical assignment we are going to create a test package with the subdirectory structure: `src/ main/ java`, which contains the following classes:

- `HolaMundo.java`
- `Mensaje.java`

One MB represents the controller part of the MVC pattern and must be programmed as 1 or more Java classes. Starting with version 2.0 of the JSF framework, MBs can be *registered*(included in the framework) by using the annotations that we are going to point out below.

The `@ManagedBean` annotation will mark the bean to be managed by the JSF framework with the name that appears in its `name` attribute. If no name is specified, then the default name will be the name of the class. If we assign the attribute `eager` the value `true` it will mean that the *bean* will be created before being requested; in case we are not convinced by that, we will use the opposite value of the attribute so that it has a *lazy* behavior (its creation is delayed until it is needed).

The other scope annotations that the JSF framework understands are as follows:

- `@RequestScoped`
- `@NoneScoped`
- `@ViewScoped`
- `@SessionScoped`
- `@ApplicationScoped`
- `@CustomScoped`

La anotación `@ManagedProperty` indica la inyección de una dependencia estática simple en un *managed bean* que entiende JSF.

We will use the first one (`@ RequestScoped`) in both classes that we have to program in this part of the practice: `HelloWorld.java` and `Message.java`. The `@ManagedProperty` annotation means to inject a simple static dependency into a *managed bean* that understands JSF. In this way, a property of a package `test` with the MB (`Message.java`), would be as follows:

```java
package prueba;
import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;


@ManagedBean(name = "mensaje", eager = true)
@RequestScoped
public class Mensaje implements Serializable{
        private static final long serialVersionUID = 1L;
        private String mensaje;

        public Mensaje() {
                this.mensaje = "Constructor␣mensaje" ;
        }
        public String getMensaje() {
                return mensaje;
        }
        public void setMensaje (String mensaje) {
                this.mensaje =mensaje;
        }
}
```

The previous MB can be injected into another MB (for example in `HelloWorld.java`) flexibly and quickly:

```java
package prueba;
import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;
import javax.faces.bean.RequestScoped;

@ManagedBean(name = "holaMundo", eager = true)
@RequestScoped
public class HolaMundo implements Serializable{
  private static final long serialVersionUID = 1L;
  @ManagedProperty(value="#{mensaje}")

  private Mensaje mensajeBean;
  private String mensaje= "Nada aun";//no confundir con el MB 'mensaje'

  public HolaMundo() {
            System.out.println("HolaMundo ha comenzado!");
  }
  public String getMensajeP() {
                return this.mensaje;
  }
  public String getMensaje() {
                mensaje = mensajeBean.getMensaje() ;
                return mensaje;
  }
  public void setMensajeBean(Mensaje mensaje) {
                mensajeBean = mensaje;
  }
}
```

## 2.7 Build the Maven project and deploy the component

We go to the `holamundo` project in our IDE explorer and in the fold-down that is activated with the right mouse button we select *"Convert to Maven Project"*. Later, in the same contextual menu, we will select *" Run as "* in the option *Maven Test* and then *Maven Install* to generate the deployment file `helloworld.war` (see in `http://www.yolinux.com/TUTORIALS/Java-WAR-files.html` to know what a `.war` is).

Before being able to execute the component that we have developed in our project, we have to prepare the application for its correct start-up on the Tomcat server. This last activity is known by the English term "to deploy" (*deploy*) and is typical of component-based software development. Therefore, now the software development cycle will generally contain more phases: *analyze-design-implement-deploy* ..., than those of traditional software development.

To perform the *deployment* correctly, you must copy the file `helloworld.war` in the subdirectory `webapps`. Then, we start Tomcat by executing `startup.bat` (you have to stop it first if it is already running). If everything has gone well, we will verify that a subdirectory `helloworld` has been created, within the `webapps` of the Tomcat installation that we have made on our disk, which contains the following structure:

```
META-INF
  \maven
     \prueba
          \holamundo
               \pom.properties  \pom.xml
   MANIFEST.MF
```

```
WEB-INF
  \classes
     \prueba
       \HolaMundo.class \Mensaje.class
   \lib
      \jsf-api-2.1.7
      \jsf-imp-2.1.7
   web.xml
home.xhtml
index.jsp
```

If the entire subdirectory and file structure above is found, then we have successfully deployed our first *UI component* at the root of the Tomcat Web Server.

## 2.8 View creation

Now we have to create an XHTML file, which we will call `home.xhtml` within the subdirectory `webapp`, which will represent the “view” component within the MVC architectural pattern. It basically consists of filling up the template given in **??**.

```
...
<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>
                <title>
                        Ejemplo JSF - 2!
                </title>
        </head>
        <body>
        <p>#{holaMundo}</p>
        <p>#{holaMundo.getMensajeP()}</p><!--returns the local value of HolaMundo-->
        <p>#{holaMundo.mensaje}</p><!--calls the constructor method of Mensaje-->
        <p>#{mensaje.setMensaje("Hola␣a␣todo␣el␣mundo!")}</p>
        <p>#{mensaje.getMensaje()}</p><!--returns the new value of "mensaje" -->
        <p>#{holaMundo.getMensajeP()}</p><!-- the local copy of "mensaje" in HolaMundo hasn't␣changed␣-->
␣␣␣␣␣␣␣␣<p>#{holaMundo.getMensaje()}</p><!--now␣changes␣the␣"mensaje"␣in␣HolaMundo␣-->
␣␣␣␣␣␣␣␣<p>#{mensaje.setMensaje("Hi␣Folks!")}</p>
␣␣␣␣␣␣␣␣<p>#{holaMundo.getMensaje()}</p><!--now␣changes␣the␣"mensaje"␣in␣HolaMundo␣-->
␣␣␣␣␣␣␣␣<p>#{holaMundo.getMensajeP()}</p><!--the␣local␣copy␣of␣"mensaje"␣in␣HolaMundo␣has␣changed␣-->
␣␣␣␣␣␣␣␣<p>#{mensaje.setMensaje("Hi␣Folks␣all␣over␣the␣World!")}</p>
␣␣␣␣␣␣␣␣<p>#{holaMUndo.setMensajeBean(mensaje)}</p><!--really␣this␣method␣is␣useless-->
␣␣␣␣␣␣␣␣<p>#{holaMundo.getMensajeP()}</p><!--the␣local␣copy␣of␣"mensaje"␣in␣HolaMundo␣hasn't changed -->
        <p>#{holaMundo.getMensaje()}</p><!--now changes the "mensaje" in HolaMundo-->
        <p>#{holaMundo.getMensajeP()}</p><!--the local copy of "mensaje" in HolaMundo has changed -->
        </body>
</html>
```

## 2.9 Run the application

Now we only have to open the page in a browser to start the application. The name of the server (`localhost`) and the port (`8080`) may vary depending on the Tomcat configuration that we have defined on our machine.

So that it can work (choose “Run as"–“run on server" if we are using the Eclipse IDE). Finally, the output that the browser will produce on the Web-client screen of the previous view, when entering in the browser: `http://localhost:8080/HolaMundo/`, will be:

```
Prueba.HolaMundo@2aeae06b
Nada aún
Constructor mensaje
Hola a todo el mundo!
Constructor mensaje
Hola a todo el mundo!
Hi Folks!
Hi Folks!
Hi Folks!
Hi Folks all over the World!
Hi Folks all over the World!
```

If it does not work (see the list of errors shown in the browser to check it out), it will possibly be fixed by including the appropriate library in “Maven dependencies"; select with the right button on the name of the Eclipse project: *Properties →Build Path→Configure BuildPath* and in *Libraries* check if it is under *Maven Dependencies*:
`M2_REPO/com/sun/faces/jsf − api/2.1.7/` *library contents*.

“Maven dependencies" library contents:

- `jsf − api − 2.1.7.jar`
- `jsf − impl − 2.1.7.jar`

## 3 Event management in JSF

When a user clicks on a link or clicks a button-component or changes any value in a text field, the JSF *UI component* issues an *event* that will be captured and managed by the Web application code.

An *event handler* has to be already registered within the application code or the MB to be able to handle such events.

When a *UI component* checks that the event caused by an action from the user has occurred, then it creates an instance of the corresponding class and adds it to the event list.

The component will then retransmit the event, that is, it checks the list of *listeners* for that specific event and calls the notification method on each of the *listeners* or *handlers* included in that list.

JSF also provides us *handlers* for system-level events, which can be used to perform some specific tasks at a lower level than that of applications, for example, during an application startup or shutdown.

Below we can find more information about the most important*event handlers* currently provided by JSF 2.0 and a demonstration example following a tutorial style of each of them:

- The change of value events (see `http://lsi2.ugr.es/dsbcs/Documentos/Practica/valueChangeL` `htm`) that are fired when the user makes changes to the input components.

- Action events (see `http://lsi.ugr.es/dsbcs/Documentos/Practica/actionListener.htm`) that are fired when the user presses a button or clicks on a link.
- The application events which are the events that are fired during the life cycle of the JSF framework, for example, those that are fired when the application starts, the application is about to close, and prior to displaying a new one page (`http://lsi.ugr.es/dsbcs/Documentos/Practica/applicationEvents.htm`):
  `PostConstructApplicationEvent`, `PreDestroyApplicationEvent`
  and `PreRenderViewEvent`.

## Credits

- `http://www.tutorialspoint.com/jsf/`
- `http://programacion.net/articulo/introduccion_a_la_tecnologia_javaserver_faces_233`
- J. Murach, M. Urban. “Java Servlets and JSP”. Murach, 2014
- `https://www.vogella.com/tutorials/JavaServerFaces/article.html`