Desarrollo y despliegue de componentes software–I

Comienzo de la práctica	15-10-2020
Fecha recomendada de entrega de la práctica	22-10-2020

1 Introducción al marco de trabajo JSF

Java Server Faces (JSF) es un marco de trabajo que ayuda al desarrollo de aplicaciones Web siguiendo el patrón arquitectónico MVC ("Model View Controller"). Simplifica la construcción de interfaces de usuario (IUs) utilizando componentes software. Una página JSF proporciona facilidades para conectar *widgets*[1] una IU (que llamamos *componenteIU* en adelante) con fuentes de datos (o "recursos", según el modelo de componentes estudiado en clase) y manejadores de eventos (o "servicios") que se ubican en el lado del servidor.

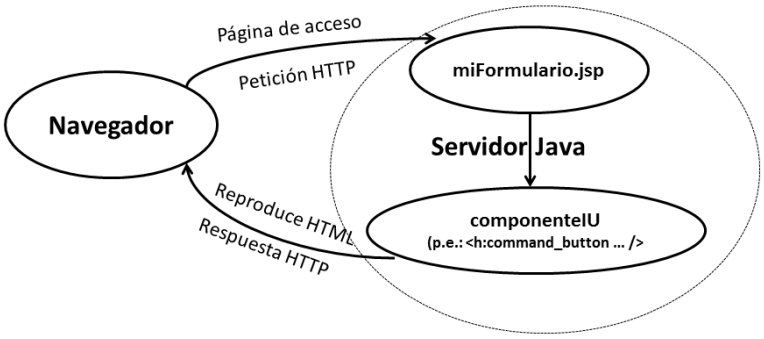

Figura 1: Arquitectura de una aplicación que utiliza la tecnología JSF

En la figura 1 se puede ver la página `miFormulario.jsp`, que muestra gráficamente los componentes del interfaz de usuario, utilizando para ello las etiquetas personalizadas que define la tecnología JSF. El *componenteIU* de la figura es un componente de la aplicación Web que actúa sobre los "objetos" referenciados por el formulario `jsp`.

La página JSP gestiona todos los objetos que intervienen en la aplicación Web, tales como:

- objetos componentes (las etiquetas del tipo `<h:command_button .../>`,
- los *escuchadores* de las acciones que se inician desde los objetos componentes, tales como *validadores* y *conversores*, que suelen estar incluidos en las etiquetas de los componentes (por ejemplo, `<f:validate _longrange minimum="0" maximum="10"/>`),
- los objetos del modelo (*beans*) que contienen los datos de la aplicación Web,
- el resto de la funcionalidad de los componentes de la aplicación.

Con la tecnología JSF podemos conectar eventos generados en el cliente con el código de la aplicación Web en el lado del servidor. También hacer corresponder[2] los *componentesIU* con los

[1] un componente de una interfaz, que permite a un usuario realizar una función o acceder a un servicio

[2] es decir, que "se entiendan"

objetos del lado del servidor. Podemos construir un *componenteIU* utilizando para ello componentes extensibles y reutilizables. Así como guardar y restaurar posteriormente el estado de la IU de nuestra aplicación después de que se hayan terminado las peticiones al servidor.

Adicionalmente, el utilizar la tecnología JSF nos ofrece una serie de importantes beneficios que hacen más fácil el programar aplicaciones Web, tales como, por ejemplo, ofrecer una clara separación entre el comportamiento de los componentes software y su representación dentro de una página Web. También mejora algunas de las deficiencias que presenta la tecnología JSP, ver figura 2, ya que con esta tecnología no se pueden hacer peticiones *http* al manejador de eventos del servidor, ni manejar componentes-IU como si fueran objetos.

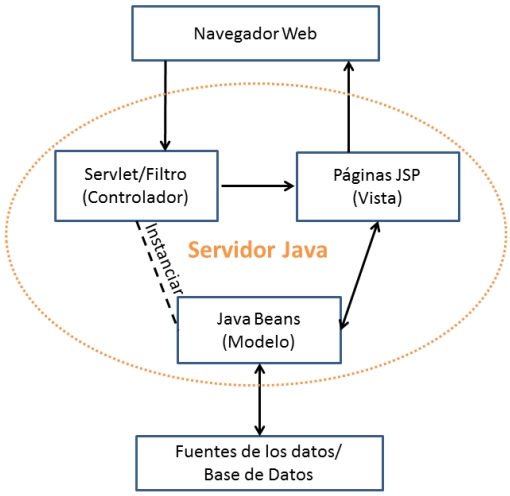

Figura 2: Modelo MVC de una aplicación que utiliza páginas JSP

JSF incluye una biblioteca de etiquetas JSP personalizadas para representar componentes en una página Web. Los autores de páginas Web sin mucha experiencia de programación pueden utilizar esas etiquetas para incluir código en las aplicaciones, que normalmente se ubican en la parte servidora, e incrustarlo en sus páginas Web, sin necesidad de tener que hacerlo programando con algún lenguaje de marcas (Javascript, etc.). Los API (son varios realmente) de la tecnología JSF se han creado directamente encima del denominado *Servlet* API de Java de tal forma que si utilizamos JSF, también podríamos utilizarlo con otras tecnologías de presentación de páginas Web para programar la parte del cliente de nuestras aplicaciones Web.

1.1 Maven: cómo empaquetar "componentes" para desplegar

Maven es una palabra en Yiddish que significa "acumulador de conocimiento". Se trata de una manera estándar de construir proyectos basados en Java con capacidades para generar *componentes software* que puedan ser desplegados en herramientas Web específicas, tales como Apache Tomcat, por ejemplo; se puede ver más información sobre WTP en `https://eclipse.org/webtools/`). Con Maven podemos conseguir desarrollar *componentes software* siempre que garanticemos lo siguiente:

- Tenemos una definición clara de lo que contiene un proyecto (programado en Java, en este caso)
- Utilizaremos una forma fácil de publicar dicha información (archivo `web.xml`, por ejemplo)
- Siempre podremos compartir archivos en formato JAR entre varios proyectos

1.2 Managed beans

Un Managed Bean (MB) es una clase especial de Java (denominada *bean*) pero registrada dentro del marco de trabajo JSF. Dicho de otra manera,**un MB es un *Java Bean* gestionado por JSF**.

Recordemos que el concepto de *Java Bean* se refiere a una clase escrita en Java que encapsula muchos objetos en una única entidad, que es el *bean*, y que está pensada para su despliegue en un entorno de ejecución distribuido. Por consiguiente, Java *Beans* tiene por objetivo crear un modelo de componentes software reutilizables utilizando para ello, como soporte de programación, el lenguaje Java. Los *beans* son serializables, poseen un método constructor sin argumentos y permiten acceder a sus propiedades (o *atributos*, según la terminología de orientación a objetos a la que estamos acostumbrados, mediante la utilización de los métodos *getter()* y *setter()*.

Por consiguiente, un MB contiene los siguientes elementos:

- métodos `getter()` y `setter()`
- La "lógica del negocio", es decir, toda la información, asociada a un *bean*, y contenida en formularios HTML de aplicación

Un MB actúa como el "Modelo", de acuerdo con el patrón MVC, aplicado para construir aplicaciones Web, que usa la tecnología JSF y lo hace para cada *componenteIU* de nuestra aplicación. Los archivos correspondientes son clases-Java que se ubicarán (por ahora) dentro del paquete `prueba` en el subdirectorio `src/main/java` del fuente principal del componente que vamos a programar.

```
1  package prueba;
2  import java.io.Serializable;
3  import javax.faces.bean.ManagedBean;
4  import javax.faces.bean.RequestScoped;
5
6  @ManagedBean(name="holaMundo", eager=true)
7  @RequestScoped
8  public class HolaMundo implements Serializable {
9    private static final long serialVersionUID = 1L;
10
11   public HolaMundo() {
12           System.out.println("HolaMundo ha comenzado!!!");
13   }
14   public String getMensaje() {
15           return "Hola Mundo!!!";
16   }
17 }
```

1.3 El controlador

Faces servlet[3] inicializa todos los componentes JSF de la aplicación que hemos desarrollado, antes de que comience su ejecución; es decir, antes de que el marco de trabajo JSF se "despliegue" completamente en nuestra plataforma de ejecución. Podemos decir que *FacesServlet* es el *servlet* principal responsable de gestionar todas las solicitudes desde la parte de cliente de la aplicación Web a los componentes de la aplicación ubicados en el servidor. *FacesServlet* actúa como el controlador central, de acuerdo con el modelo MVC que estamos siguiendo, inicializando todos los componentes de JSF, antes de que se muestre la página JSP que incluye el formulario (ver figura 1) del cliente-Web. *FacesServlet* se ubica en la carpeta `webapp/WEB_INF` del código del componente que estamos produciendo.

[3] un *servlet* es como un *applet* de Java que se ejecuta en el servidor en lugar de hacerlo en la máquina cliente que abre el navegador

1.4 La vista

Normalmente se trata de una página codificada con XHTML[4] que adquiere el nombre de archivo `home.xhtml` o `index.xhtml` y que se ubica en el subdirectorio `DeployedResources/WebContent` del proyecto Java del componente.

```
<!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 Configuración del marco de trabajo de la práctica

Antes de comenzar a programar el *managed bean* en el IDE de tu elección[5], hay que comprobar que la instalación de Java incluye un JDK > =1.5, después pasaríamos a descargar y configurar Maven y, por último, instalaríamos Apache Tomcat para intentar ejecutar un "hola mundo" desde el IDE correctamente configurado. **Alternativamente, podríamos hacerlo todo en el IDE Eclipse**, ya que si hemos instalado *Eclipse IDE for Enterprise Java Developers* podemos seleccionar el tipo *Dynamic Web Project* para crearnos el proyecto inicial *HolaMundo* (Target Runtime: Apache Tomcat v.x y Dynamic Web Module Version: *última versión*), con siguiente configuración de carpetas:

```
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) pertenece a la familia de lenguajes de marcado XML markup languages, extiende al lenguaje HTML para la escritura de páginas; XHTML puede ser explorado utilizando parsers XML, a diferencia de HTML que necesita un explorador específico para este código

[5]se recomienda utilizar Eclipse para esta práctica

2.1 Descargar y configurar Maven

Se puede descargar del sitio Maven en Apache Software Foundation (ASF), uno de los siguientes archivos comprimidos, dependiendo del sistema operativo en el que se vaya a utilizar:

```
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
```

Extraer el archivo en el subdirectorio donde se quiera ubicar este software.

El subdirectorio: `apache-maven-x.y.z` se creará entonces, a partir del archivo extraido. Hay que añadir las siguientes variables al entorno de ejecución de la máquina: `M2_HOME` y `MAVEN_OPTS (-Xms256m -Xmx512m)`. Por ejemplo, en Linux, para ajustar las variables de entorno, utilizando una ventana de *bash*, tecleraríamos lo siguiente:

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

Por último, añadir la variable de entorno `M2` al `PATH` (`export PATH=`$M2$ `:PATH`) y verificar que la instalación de Maven se ha realizado correctamente: `mvn -version`.

Alternativamente, podríamos hacerlo directamente desde el IDE Eclipse partiendo de un *Dynamic Web Project*, seleccionando con el botón derecho encima del nombre del proyecto: *Configure⇒Convert to Maven Project*, que añade un `pom.xml` en el subdirectorio *WebContent* del proyecto.

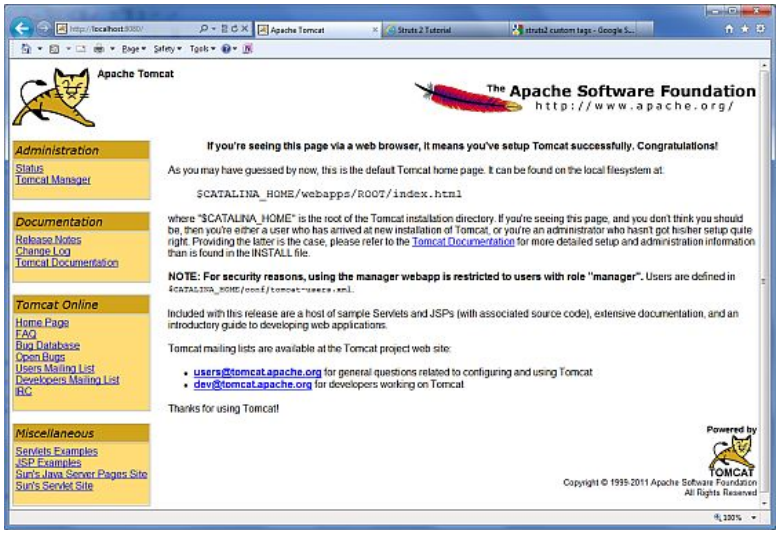

Figura 3: Página principal de Tomcat.

2.2 Instalar Apache Tomcat

Se puede bajar la última versión de Tomcat, servidor web de código abierto para ejecutar servlets y otras aplicaciones Web, de `http://tomcat.apache.org/` y una vez que se haya desempaquetado la distribución en el subdirectorio elegido, hay que asignar la variable de entorno: `CATALINA_HOME`.

Tomcat puede ser lanzado o parado con los archivos de órdenes `startup.bat` / `shutdown.bat` en el subdirectorio `bin` de la carpeta donde tengamos la instalación local de Tomcat. Después de un arranque con éxito de Tomcat, las aplicaciones Web que vayamos incluyendo en el subdirectorio `webapps` de Tomcat estarán disponibles para sus clientes visitando la página: `http:/localhost:8080`. Si lo hemos configurado todo correctamente, nos ha de aparecer la ventana de inicio de administración de Tomcat (Figura 3).

2.3 Configurar JSF

No hay que hacerlo si se ha configurado Maven desde Eclipse IDE (ver sección 2.0) y se ha convertido en un *Maven Project*

Lo primero sería crearse un proyecto Maven, lo cual se puede hacer desde la línea de órdenes, tecleando: `mvn archetype : generate −DgroupId = prueba −DartifactId = holamundo −DarchetypeArtifactId = maven − archetype − webapp`

Hemos de comprobar ahora que se nos ha creado una estructura de subdirectorios que incluya:

- `holamundo`: que contenga el subdirectorio `src` y el archivo del modelo objetos del proyecto (POM): `pom.xml`, que es una representación en XML del proyecto Maven
- El subdirectorio `src/main/webapp` que contiene a `WEB_INF` y la página de índice JSP
- El subdirectorio `src/main/resources` donde se ubicarán los recursos (propiedades, imágenes, etc.) asociados al *componenteIU* que queremos desarrollar

2.3.1 Preparar el proyecto para importarlo en nuestro IDE

No hay que hacerlo si se ha configurado Maven desde Eclipse IDE (ver sección 2.0) y se ha convertido en un *Maven Project*

Tenemos que comprobar que el proyecto Maven se ha construido bien y que lo hemos convertido en un proyecto que pueda ser incluido en nuestro IDE.

Si utilizamos el IDE Eclipse, entonces utilizaremos la orden `mvn eclipse : eclipse −Dwtpversion = 2.0` (dentro del subdirectorio donde resifa `holamundo` para que encuentre al archivo de configuración `pom.xml`) que nos prepara un proyecto para este IDE y si se construye correctamente la aplicación `holamundo`, ha de aparecernos algo como lo siguiente:

```
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] ---------------
```

Para terminar, ya sólo hemos de importar el proyecto así generado en nuestro IDE para poder comenzar a realizar esta primera práctica.

2.4 Definición del `pom.xml`

Ahora, le añadiremos las capacidades que proporciona el marco de trabajo JSF al proyecto, para lo cual hemos de incluir las siguientes dependencias en la parte apropiada del archivo `POM.xml`,

```
<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>
```

Obviamente, los números de versión que aparecen anteriormente se han de sustituir por la últimas versiones estables. Es conveniente incluir también la versión de compilador de Maven:

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

Y la sección para indicar el plugin necesario para generar el componente desplegable en formato `war` (si no se hace, lo generaría en el formato `jar` por defecto):

```
<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 Configuración del servlet

Accedemos a nuestro proyecto, correctamente importado en el IDE que vayamos a utilizar, y buscamos el archivo `web.xml` para comprobar que presenta el siguiente contenido:

```
<?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 es un servlet principal responsable de gestionar todas las solicitudes.
Actua como un controlador central.
Este servlet inicializa los componentes de JSF antes de que se muestre JSP.
-->
<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

Para realizar esta parte de la práctica vamos a crearnos un paquete de prueba con la estructura de subdirectorios: `src/main/java`, que contenga las siguientes clases:

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

Un MB representa la parte del controlador del patrón MVC y se ha de programar como 1 ó más clases de Java. A partir de la versión 2.0 del marco de trabajo JSF, los MB se pueden *registrar* (incluir en el marco de trabajo) utilizando las anotaciones que a continuación vamos a indicar.

La anotación `@ManagedBean` marcará al bean para ser gestionado por el marco JSF con el nombre que aparezca en su atributo `name`. Si no se indica ningún nombre, entonces el nombre por defecto será el nombre de la clase. Si asignamos al atributo `eager` el valor `true` significará que el *bean* se creará antes de ser solicitado; en caso que no nos convenza eso, utilizaremos el valor contrario del atributo para que tenga un comportamiento *perezoso* (se retrasa su creación hasta que se necesita).

El resto de anotaciones de ámbito que entiende el marco de trabajo JSF son las siguientes:

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

Utilizaremos la primera (`@RequestScoped`) en ambas clases que hemos de programar en esta parte de la práctica: `HolaMundo.java` y `Mensaje.java`. La anotación `@ManagedProperty` indica la inyección de una dependencia estática simple en un *managed bean* que entiende JSF. De esta forma una propiedad de un paquete `prueba` con el MB (`Mensaje.java`), quedaría como sigue:

```
1  package prueba;
2  import java.io.Serializable;
3  import javax.faces.bean.ManagedBean;
4  import javax.faces.bean.RequestScoped;
5
6
7  @ManagedBean(name = "mensaje", eager = true)
8  @RequestScoped
9  public class Mensaje implements Serializable{
10         private static final long serialVersionUID = 1L;
11         private String mensaje;
12
13         public Mensaje() {
14                 this.mensaje = "Constructor mensaje" ;
15         }
16         public String getMensaje() {
17                 return mensaje;
18         }
19         public void setMensaje (String mensaje) {
20                 this.mensaje =mensaje;
21         }
22 }
```

El MB anterior se puede inyectar en otro MB (por ejemplo en `HolaMundo.java`) de forma flexible y rápida:

```
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 Construir el proyecto Maven y desplegar el componente

Nos situamos sobre el proyecto `holamundo` dentro del explorador de nuestro IDE y en el desplegable que se activa con el botón derecho del ratón seleccionamos *"Convert to Maven Project"*. Posteriormente, en el mismo menú contextual, seleccionaremos *"Run as"* en la opción *Maven Test* y después *Maven Install* para generar el archivo de despliegue `holamundo.war` (ver en `http://www.yolinux.com/TUTORIALS/Java-WAR-files.html` para saber qué es un `.war`).

Antes de poder ejecutar el componente que hemos desarrollado en nuestro proyecto hemos de preparar la aplicación para su correcta puesta en funcionamiento en el servidor Tomcat. Esta última actividad se conoce con el término en inglés "to deploy" (*desplegar*) y es típico del desarrollo de software basado en componentes. Por tanto, ahora el ciclo de desarrollo de software contendrá en general más fases: *analizar-diseñar-implementar-desplegar...*, que las que posee el desarrollo de software tradicional.

Para realizar correctamente el *despliegue* hay que copiar el archivo `holamundo.war` en el subdirectorio `webapps`. Después arrancamos Tomcat ejecutando `startup.bat` (hay que pararlo primero si ya estuviera en ejecución). Si todo ha ido bien, comprobaremos que se ha creado un subdirectorio `holamundo`, dentro del `webapps` de la instalación de Tomcat que hayamos realizado en nuestro disco, que contiene la siguiente estructura:

```
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
```

Si se encuentra toda la estructura de subdirectorios y archivos anterior, entonces hemos conseguido desplegar correctamente nuestro primer *componenteIU* en el raíz del Servidor Web Tomcat.

2.8 Crear la vista

Ahora tenemos que crear un archivo XHTML, que llamaremos `home.xhtml` dentro del subdirectorio `webapp`, que representará el componente "vista" dentro de patrón arquitectónico MVC. Básicamente consiste en completar la plantilla dada en 1.4.

```
...
<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><!--devuelve el valor local de HolaMundo-->
        <p>#{holaMundo.mensaje}</p><!--se llama al constructor de Mensaje-->
        <p>#{mensaje.setMensaje("Hola a todo el mundo!")}</p>
        <p>#{mensaje.getMensaje()}</p><!--devuelve el nuevo valor del mensaje -->
        <p>#{holaMundo.getMensajeP()}</p><!--la copia local de mensaje en HolaMundo no ha cambiado -->
        <p>#{holaMundo.getMensaje()}</p><!--ahora cambia mensaje en HolaMundo -->
        <p>#{mensaje.setMensaje("Hi Folks!")}</p>
        <p>#{holaMundo.getMensaje()}</p><!--ahora cambia mensaje en HolaMundo -->
        <p>#{holaMundo.getMensajeP()}</p><!--la copia local de mensaje en HolaMundo ha cambiado -->
        <p>#{mensaje.setMensaje("Hi Folks all over the World!")}</p>
        <p>#{holaMUndo.setMensajeBean(mensaje)}</p><!--realmente este metodo no sirve para nada-->
        <p>#{holaMundo.getMensajeP()}</p><!--la copia local de mensaje en HolaMundo no ha cambiado -->
        <p>#{holaMundo.getMensaje()}</p><!--ahora cambia mensaje en HolaMundo -->
        <p>#{holaMundo.getMensajeP()}</p><!--la copia local de mensaje en HolaMundo ha cambiado -->
        </body>
</html>
```

2.9 Ejecutar la aplicación

Ahora sólo nos queda abrir la página en un navegador para arrancar la aplicación. El nombre del servidor (`localhost`) y del puerto (`8080`) pueden variar dependiendo de la configuración de Tomcat que tengamos definida en nuestra máquina.

Para que pueda funcionar (elegir “Run as" –“run on server" si estamos utilizando el IDE de Eclipse). Finalmente, la salida que producirá el navegador en la pantalla del cliente-Web de la vista anterior, al introducir en el buscador: `http://localhost:8080/HolaMundo/`, será:

```
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!
```

Si no funcionara (lista de errores mostrada en el navegador), posiblemente se arreglará incluyendo la librería adecuada en “Maven dependencies"; seleccionar con el botón derecho en el nombre del proyecto Eclipse: Propiedades→Build Path→Configure BuildPath y en *Libraries* comprobar si está debajo de *Maven Dependencies*:
`M2_REPO/com/sun/faces/jsf − api/2.1.7/` *contenidos de la librería*.

Contenidos de la librería “Maven dependencies":

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

3 Gestión de eventos en JSF

Cuando un usuario pica en un enlace o pulsa un componente-botón o cambia cualquier valor de un campo de texto, el *componenteIU* de JSF emite un *evento* que será capturado y gestionado por el código de la aplicación Web.

Un *manejador* de eventos (o *“event handler”*) tiene que estar ya registrado dentro del código de la aplicación o del MB para poder gestionar tales eventos.

Cuando un *componenteIU* comprueba que ha ocurrido el evento provocado por una acción desde el usuario, entonces crea una instancia de la clase correspondiente y la añade a la lista de eventos. Después, el componente retransmitirá el evento, es decir, comprueba la lista de *escuchadores* de ese evento concreto y llamará al método de notificación en cada uno de los *escuchadores* o *manejadores* incluidos en dicha lista.

JSF también nos proporciona *manejadores* de eventos a nivel del sistema, que pueden ser utilizados para realizar algunas tareas específicas a más bajo nivel que el de las aplicaciones, por ejemplo, durante el arranque o parada de una aplicación.

A continuación podemos encontrar más información sobre los *manejadores* de eventos más importantes que actualmente nos proporciona JSF 2.0 y un ejemplo de demostración siguiendo un estilo tutorial de cada uno de ellos:

- Los eventos de cambio de valor (ver `http://lsi.ugr.es/dsbcs/Documentos/Practica/valueChangeListener.htm`) que se disparan cuando el usuario hace cambios en los componentes de la entrada.
- Los eventos de acción (ver `http://lsi.ugr.es/dsbcs/Documentos/Practica/actionListener.htm`) que se disparan cuando el usuario pulsa un botón o pica en un enlace.
- Los eventos de aplicaciones que son los eventos que se disparan durante el ciclo de vida del marco de trabajo JSF, por ejemplo, los que se disparan cuando comienza la aplicación, la aplicación está a punto de cerrarse, y previo a mostrar en pantalla una nueva página (`http://lsi.ugr.es/dsbcs/Documentos/Practica/applicationEvents.htm`): `PostConstructApplicationEvent`, `PreDestroyApplicationEvent` y `PreRenderViewEvent`.

Créditos

- `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`