Objetivos de la sesión:

- Crear nuestro ***primer Servlet*** desde Eclipse
- Entender como funcionan los Servlets
- Crear nuestra ***primera página JSP*** desde Eclipse
- Entender como realizar una llamada **GET o POST** al servidor y como procesar esas peticiones en el servidor
- Entender que problemas tenemos cuando recibimos un código de **error 404 o 405**.

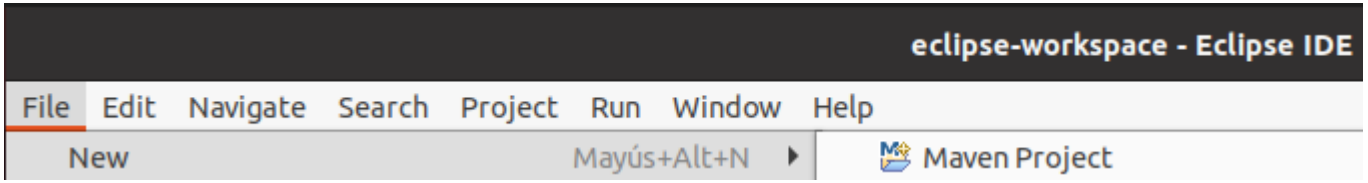
Se recomienda leer todo el pdf sin dejarse nada. En caso contrario se corre el peligro en las siguientes prácticas de no entender nada. NO SE TRATA DE MEMORIZAR, SE TRATA DE ENTENDER. El examen será con apuntes.

Creación de nuestro primer Servlet desde Eclipse:

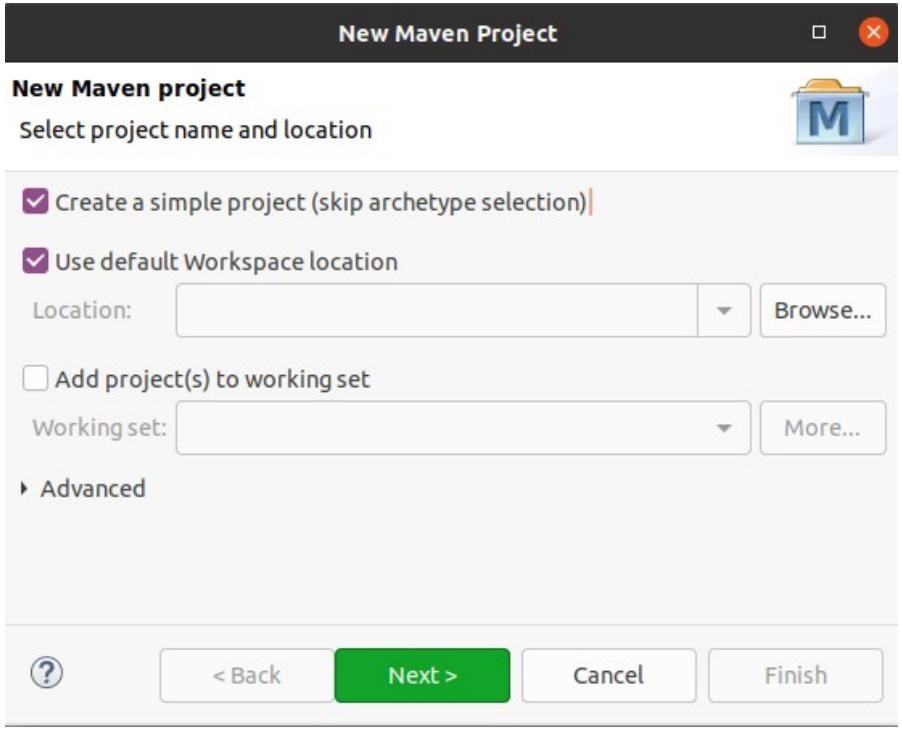
Para crear desde cero nuestro primer proyecto web que contenga un Servlet en las próximas diapositivas realizaremos los siguientes ***10 pasos***:

1º Crearemos un nuevo proyecto Maven
2º Configuraremos el proyecto sin ningún arquetipo
3º Configuraremos los datos de maven en nuestro proyecto
4º Analizaremos la estructura principal de la aplicación creada
5º Entenderemos porque necesitamos maven y su relación con el fichero pom.xml
6º Copiaremos 3 fichero a nuestro proyecto: pom.xml, web.xml y LoginServlet.java
7º Aprenderemos a actualizar los cambios de configuración desde Maven.
8º Configuraremos la ejecución de nuestro proyecto web con maven
9º Comprobaremos que tenemos nuestro primer Servlet en marcha.
10º Entenderemos la teoría que existe detrás de nuestra primera aplicación con Servlet.

1º Abrimos Eclipse y en el menu superior creamos un **nuevo proyecto Maven**:



2º **Creamos un proyecto sin arquetipo**: Marcamos Create a simple project para no tener que seleccionar un arquetipo en concreto:



3º Rellenamos los parámetros del proyecto Maven:

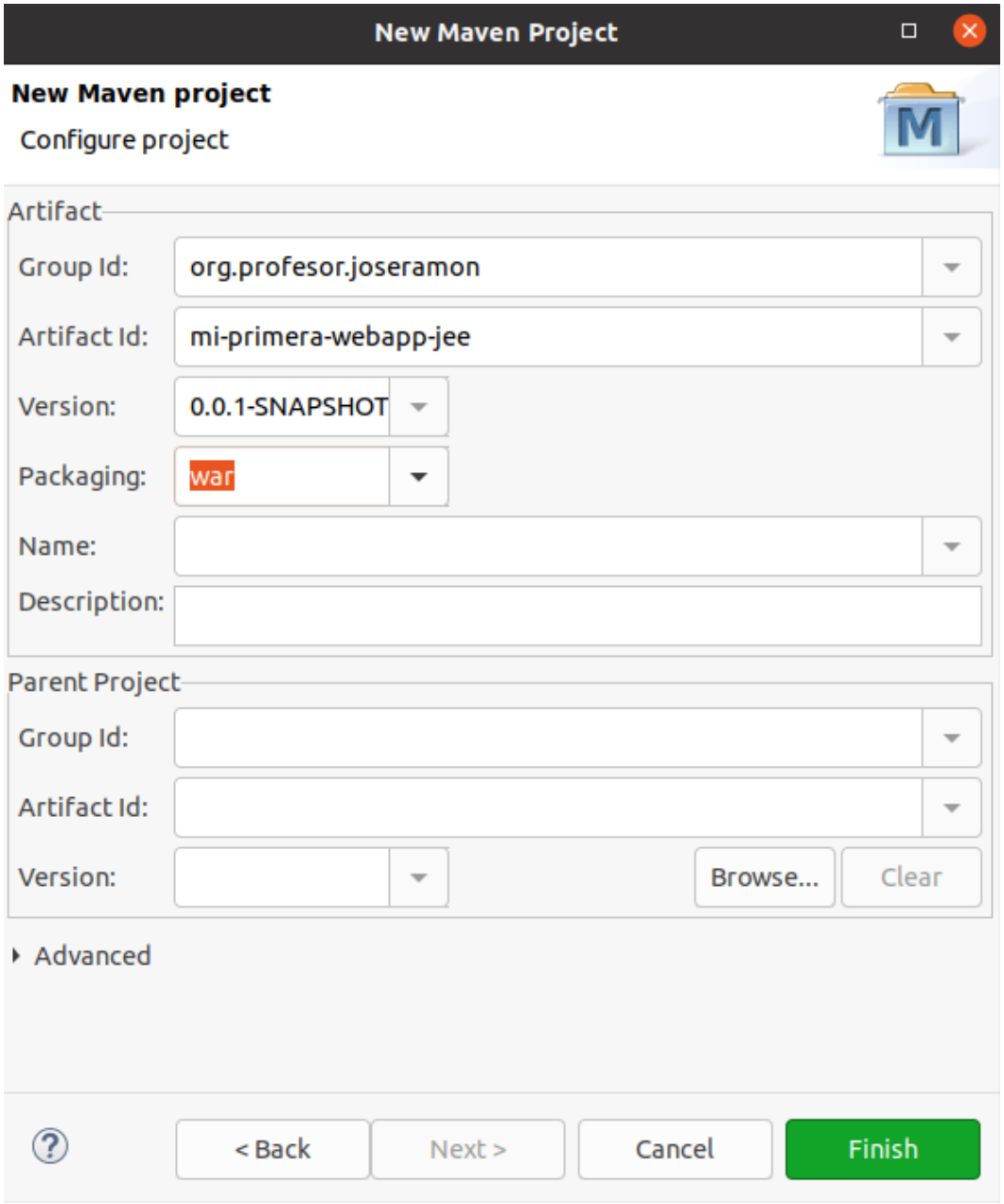
En el caso del alumno el ”Group Id” será org.alumno.NombreAlumno donde NombreAlumno es el nombre del alumno.

Es importante que en ”Packaging” seleccionemos war .

“War” significa Web Archive y es el tipo de formato que se utiliza para contener una web entera.

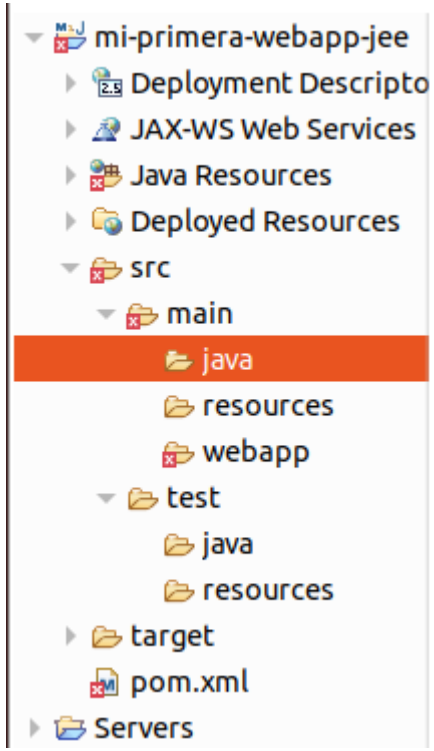
Si queremos añadir una aplicación a Tomcat debemos proporcionar un fichero con formato “.war”.

Pulsamos en “Finish”.



4º **Entendiendo la estructura de la aplicación web:**

Despliega “mi-primera-webapp-jee” para ver las secciones más Importantes que se han generado en nuestra webapp:



\src\main\java: Es la carpeta más importante del proyecto, porque es donde se dejan todos los ficheros java.

\src\main\resources: Es donde dejaríamos recursos de varios tipos, como por ejemplo ficheros de propiedades, xml,...

\src\test: Utilizado para realizar pruebas unitarias.

Fichero pom.xml:

mi-primera-webapp-jee/pom.xml

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0"
2    <modelVersion>4.0.0</modelVersion>
3    <groupId>org.profesor.joseramon</groupId>
4    <artifactId>mi-primera-webapp-jee</artifactId>
5    <version>0.0.1-SNAPSHOT</version>
6    <packaging>war</packaging>
7  </project>
```

Nos falla en la linea 6 al indicar que queremos empaquetar con un war porque no encuentra el fichero “web.xml” y la variable failOnMissingWebXml esta configurado a true. No tocaremos nada pero pronto lo arreglaremos.

5º **Mvn y el fichero pom.xml:**

· El fichero **pom.xml** es el componente más importante para Maven.

· **Maven** es una herramienta de gestión de dependencias de librerías de java (.jar) y un gestor de proyectos que nos permite crear proyectos con una estructura predefinida.

· Para ejecutar una aplicación web necesitamos muchas **librerías de java que vienen empaquetadas en fichero “.jar”** .

Si tenemos una aplicación que utiliza la tecnología JEE8 necesitaremos la librería javaee-web-api-8.0.jar

Si tenemos una aplicación que utiliza Hibernate para acceder a la base de datos necesitaremos Spring, y la librería Spring.jar.

· **Maven simplifica la vida del programador porque se encarga de gestionar las dependencias de las librerías que necesitamos en nuestro proyecto descargando los “.jar” que nos hacen falta dependiendo de la dependencia que le indiquemos en el fichero pom.xml.**

Si queremos utilizar la tecnología JEE8 necesitaremos añadir la dependencia en el fichero POM para instalar la librería javaee-web-api-8.0.jar en la webapp.

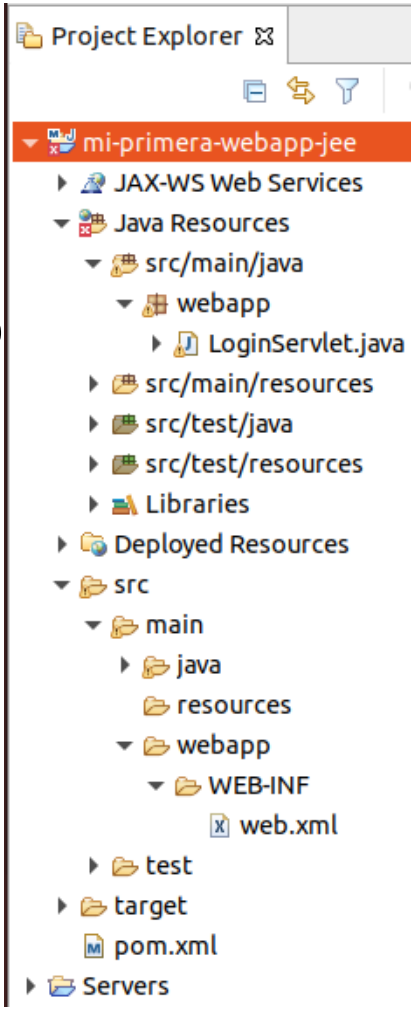
6º **Ficheros a copiar en nuestro proyecto:**

Aunque hemos creado nuestra primera aplicación desde cero, ahora vamos a copiar **3 ficheros** :

\pom.xml: Abrimos el fichero pom.xml y pegamos el contenido del fichero DWES_UD1_03_pom_xml.txt. Solo nos quedará cambiar en el groupId nombreAlumno por el nombre del alumno.

\src\main\java\webapp\LoginServlet.java: Creamos el fichero "LoginServlet.java" en \src\main\java y pegamos el contenido de DWES_UD1_03_LoginServlet.txt. Si nos situamos en la primera linea sobre la palabra "webapp" nos aparece "Move LoginServlet to package webapp" y si pulsamosEclipse creará automáticamente la carpeta webapp y colocará el fichero java dentro.

\src\main\webapp\WEB-INF\web.xml: Creamos la carpeta WEB-INF en \src\main\webapp. Creamos un nuevo fichero (New\Other\Xml file) llamado web.xml y para pegar el contenido de DWES_UD1_03_web_xml.txt no situamos primero en la subpestaña inferior **"Source"** del fichero web.xml.



7º **Actualizar los cambios con Maven:**

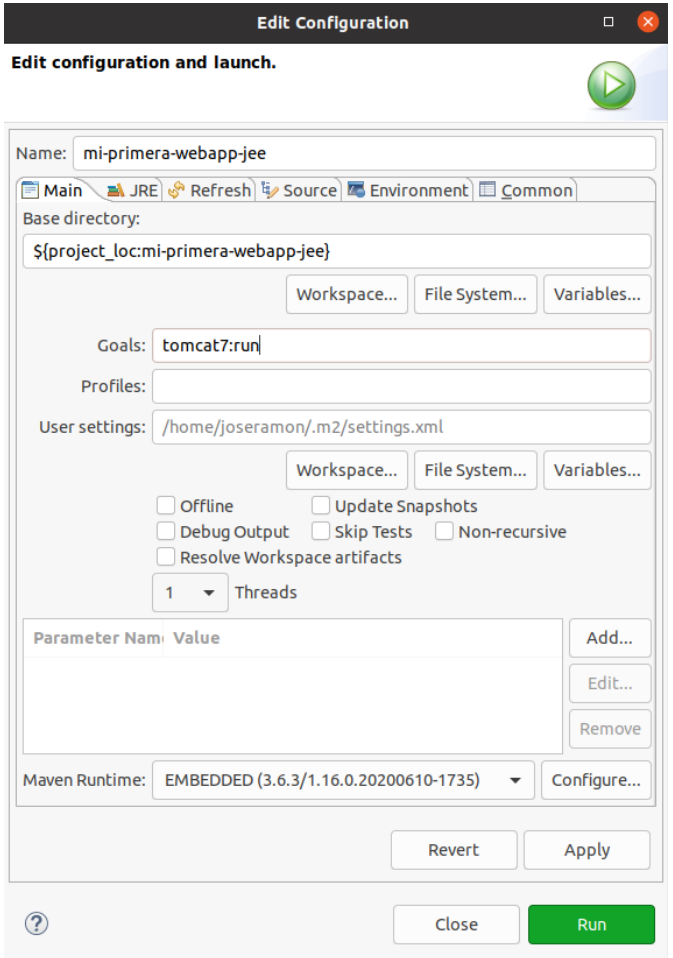
No situamos encima del nombre de nuestro proyecto y con el botón derecho pulsamos "Maven\Update project" y pulsamos OK. Podemos ver que todos los errores desaparecen y tenemos una **aplicación web correctamente configurada y sin errores**.

8º **Configurar la ejecución de la aplicación web:**

No situamos encima de nuestro proyecto y con el botón derecho pulsamos "Run as\ Maven build..." y vemos que nos aparece la siguiente pantalla:

Nos situamos en el campo "Goals" y escribimos "tomcat7:run" para ejecutar el plugin que hemos configurado en el fichero pom.xml

Nota: Fijemonos que es tomcat7 y no tomcat9 porque hasta la fecha no existe el plugin para tomcat9.

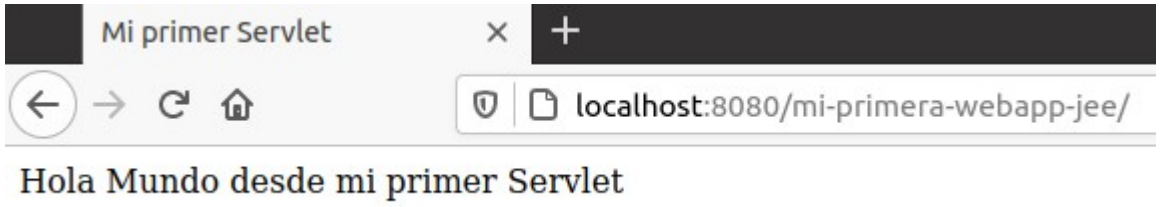


9º **Comprobar que nuestro primer Servlet esta en marcha:**

Si todo ha ido bien veremos que la **consola** nos muestra una información similar a la imagen:

```
[INFO] --- tomcat7-maven-plugin:2.2:run (default-cli) @ mi-primera-webapp-jee ---
[INFO] Running war on http://localhost:8080/
[INFO] Using existing Tomcat server configuration at /home/joseramon/eclipse-workspace/
[INFO] create webapp with contextPath:
sept. 18, 2020 3:43:59 P. M. org.apache.coyote.AbstractProtocol init
INFORMACIÓN: Initializing ProtocolHandler ["http-bio-8080"]
sept. 18, 2020 3:43:59 P. M. org.apache.catalina.core.StandardService startInternal
INFORMACIÓN: Starting service Tomcat
sept. 18, 2020 3:43:59 P. M. org.apache.catalina.core.StandardEngine startInternal
INFORMACIÓN: Starting Servlet Engine: Apache Tomcat/7.0.47
sept. 18, 2020 3:44:00 P. M. org.apache.coyote.AbstractProtocol start
INFORMACIÓN: Starting ProtocolHandler ["http-bio-8080"]
```

Y si nos vamos al **Firefox** podemos ver nuestro primera aplicación con Servlets en marcha:



Nota: Hay veces que Eclipse no funciona demasiado bien y el servidor se queda enganchado porque nos permite volver a lanzar el Tomcat con nuestro proyecto sin haber parado el Tomcat y por tanto cuando lanzamos la aplicación nos dice que alguién esta utilizando dicha url. La gran mayoria de veces se soluciona cerrando Eclipse para que se cierre el thread de Eclipse que ha lanzado el Tomcat. y volviendo a arrancar Eclipse.

10º **Entendiendo nuestra primera aplicación con Servlets:**

Fichero pom.xml:

Vamos a abrir el fichero pom.xml e intentar entender su contenido:

¿Que permite que nuestra aplicación corra en Tomcat?

... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

Fichero pom.xml:

```
<plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.2</version>
    <configuration>
        <path>/</path>
        <contextReloadable>true</contextReloadable>
    </configuration>
</plugin>
```

tomcat7-maven-plugin es el encargado de hacer que nuestra app pueda funcionar en tomcat. Como dijimos antes, a fecha de hoy todavía no hay versión 9 de este plugin y aunque utilicemos tomcat9 para hacerlo funcionar utilizamos el plugin versión 7.

... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

Fichero pom.xml:

¿Que permite que nuestra aplicación se pueda compilar con una versión específica u otra del compilador de java AUNQUE no tengamos dicha versión instalada?

... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

Fichero pom.xml:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
        <verbose>true</verbose>
        <source>1.8</source>
        <target>1.8</target>
        <showWarnings>true</showWarnings>
    </configuration>
</plugin>
```

maven-compiler-plugin es el encargado de compilar las clases, montar los ficheros .jar y los .war. Normalmente todas las aplicaciones suelen ser compatibles con la versión 1.5, peró si queremos utilizar alguna funcionalidad propia de otra versión o tenemos una versión diferente en nuestro equipo no hay problema porque el plugin de maven se encargará de compilar nuestra aplicación en la versión que deseemos. En nuestro caso le hemos dicho que compile para la versión 1.8 de Java.

... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

Fichero pom.xml:

Esto significa que podemos tener instalada una versión concreta de java en nuestro equipo local y si hace falta podemos configurar una versión diferente en la aplicación web. Esto puede pasar si el servidor tiene una versión de java diferente a la nuestra y no podemos cambiarla. Esto lo podemos conseguir con solo cambiar la versión de java en el fichero pom.xml.

Nuestra aplicación es una ***aplicación web basada en Java Enterprise Edition (JEE) en su versión 8***. Esto significa que utillizaremos clases propias del JEE8.

¿Como se le hemos dicho a maven que descargue el .jar del JEE8 para utilizar sus clases?

... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

Fichero pom.xml:

```
<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-web-api</artifactId>
        <version>8.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

javaee-web-api es una dependencia que permite indicarle a Maven que se descargue la versión 8 del JEE.

Sin esta dependencia , en nuestro proyecto fallaría cualquier referencia a clases javax.*.

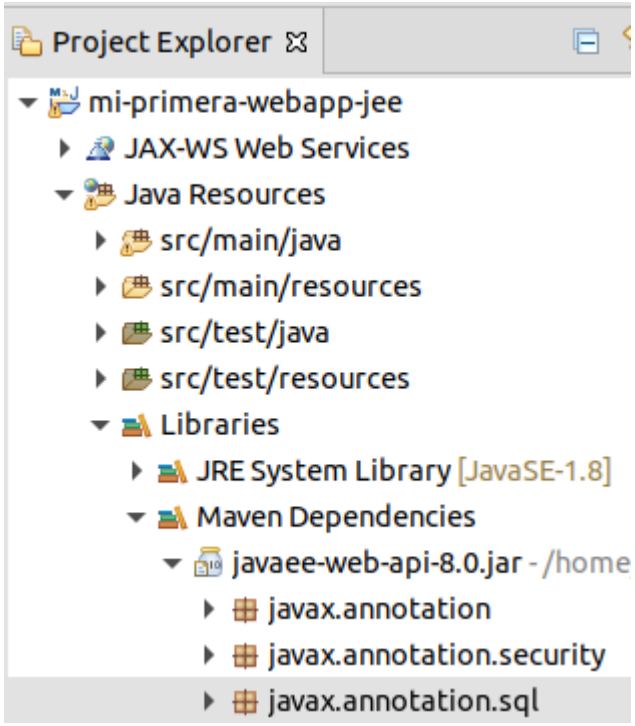
LoginServlet.java

```
1  package webapp;
2
3  import java.io.IOException;
4  import java.io.PrintWriter;
5
6  import javax.servlet.annotation.WebServlet;
7  import javax.servlet.http.HttpServlet;
8  import javax.servlet.http.HttpServletRequest;
9  import javax.servlet.http.HttpServletResponse;
10
12  * Browser sends Http Request to Web Server
19
20 /*Java Platform, Enterprise Edition (Java EE) JEE8
32
33 @WebServlet(urlPatterns = "/login.do")
34 public class LoginServlet extends HttpServlet {
```

... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

Fichero pom.xml:

Si abrimos la carpeta ***"Java Resources"*** podemos observar que en ***"Libraries"*** tenemos la versión 1.8 de Java y en dependencias de maven podemos ver el jar ***javaee-web-api-8.0.jar*** que nos permitirá utilizar en nuestro proyecto clases como ***javax.servlet.http.HttpServlet***.



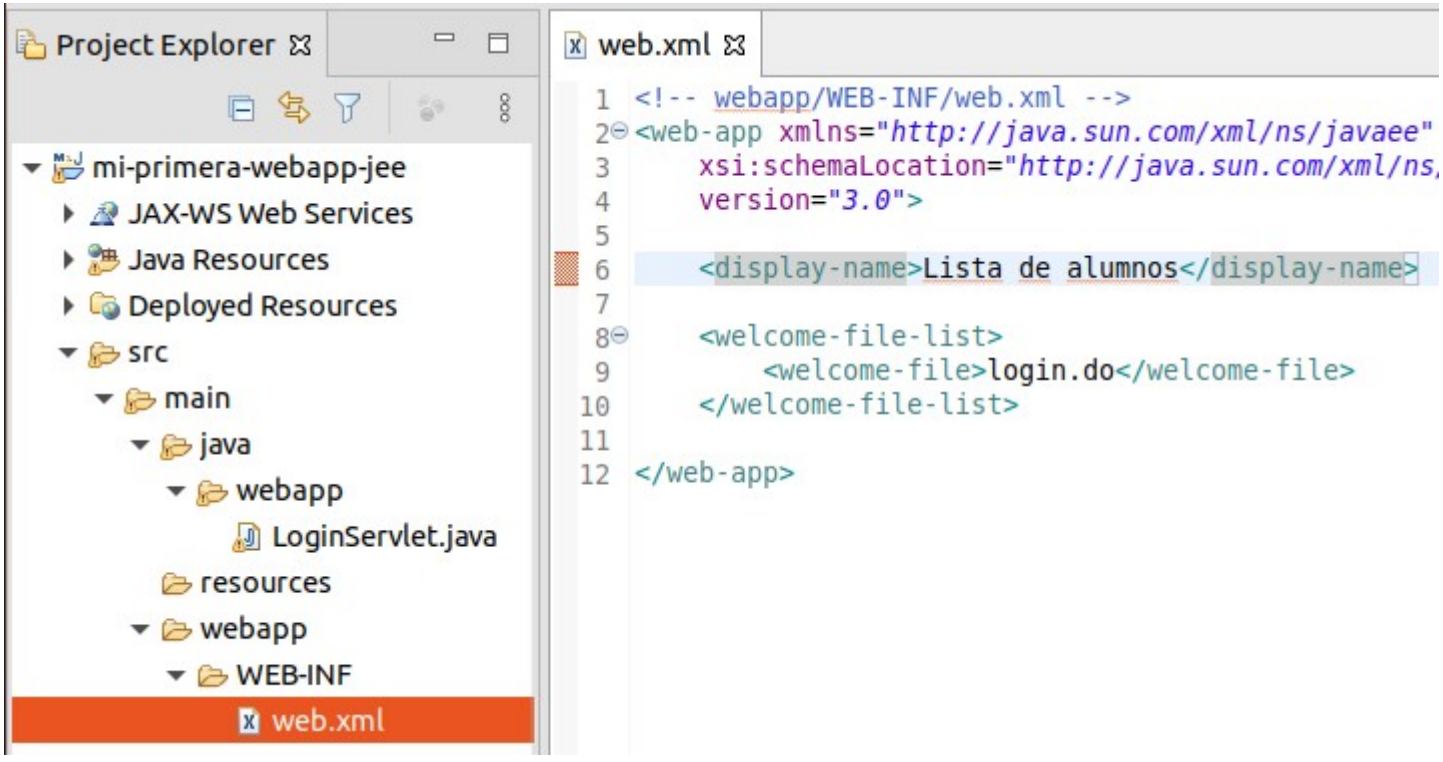
... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

¿Como le decimos a nuestra aplicación cual es la dirección de inicio a mostrar cuando arranque?

... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

Fichero web.xml:

Si abrimos la carpeta ***"src\main\webapp\WEB-INF"*** podemos ver el fichero ***"web.xml"***, que será el encargado de dar nombre a nuestra aplicación web (display-name) e indicar la url de inicio (welcome-file), en nuestro caso login.do.



... continuamos Entendiendo nuestra primera aplicación con Servlets:

Nuestra aplicación utilizará la tecnología JEE8 y más concretamente lo hará mediante Servlets.

¿Que es un Servlet?

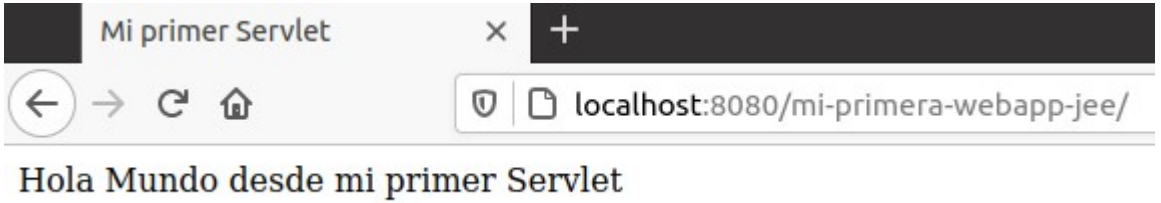
... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

Fichero LoginServlet.java:

Un ***Servlet*** es una clase Java muy sencilla. Su misión consiste en ***recepcionar peticiones (request) y contestar a dichas peticiones(response)***. Para ello, ***JEE8*** nos proporciona la clase ***javax.http.HttpServlet*** de la cual deberán extender todas nuestros Servlets (clases). ***Nuestro servlet LoginServlet hereda de HttpServlet***.

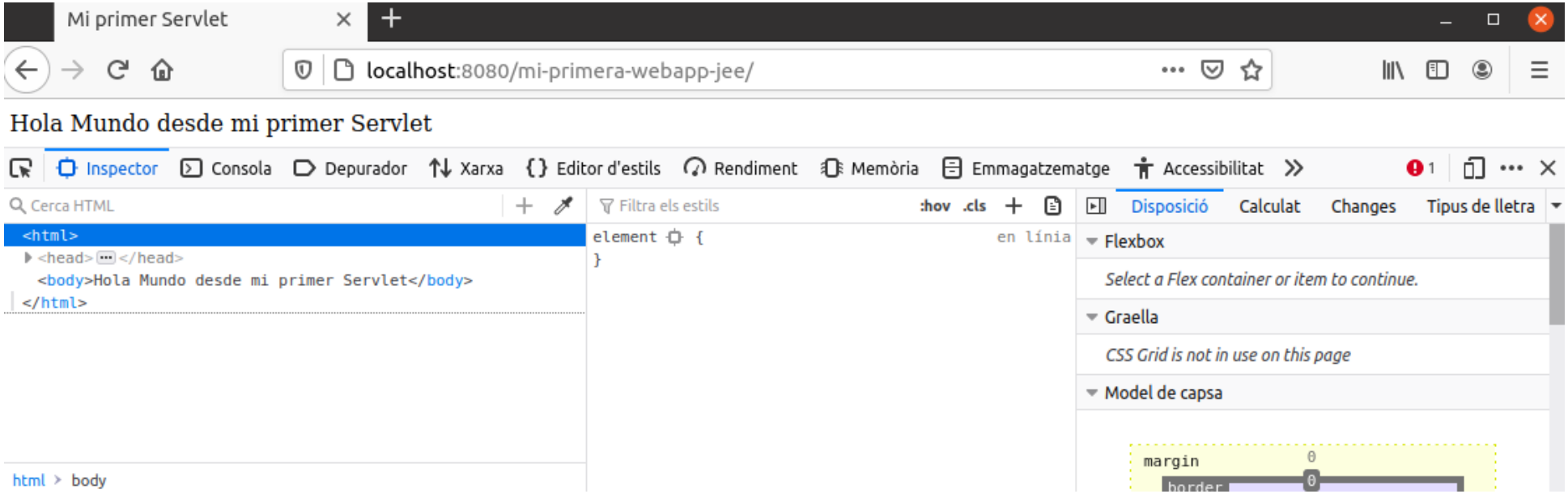
... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

¿Que pasos se realizan realmente cuando lanzamos nuestra aplicación en el Firefox?



... continuamos Entendiendo nuestra primera aplicación con Servlets:

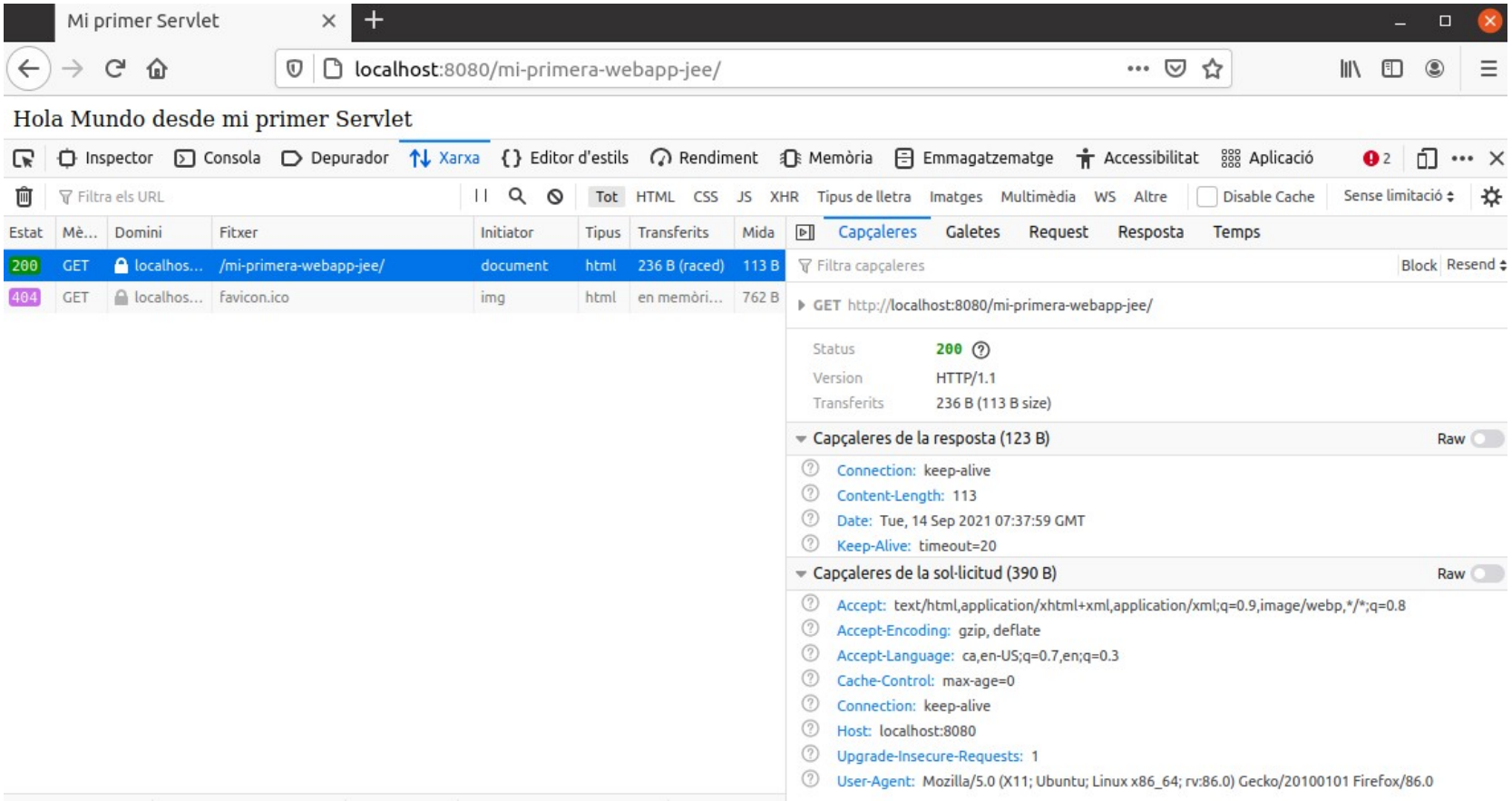
Vamos a abrir firefox y teclear *localhost:8080/mi-primera-webapp-jee/*. Si pulsamos el *botón derecho* y seleccionamos *"Inspeccionar elemento"* entramos en la sección de herramientas del desarrollador y nos muestra una ventana inferior con información:



Nos vamos a la pestaña "Red" y recargamos la página.

... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

Cuando tecleamos http://localhost:8080/mi-primera-webapp-jee/ el navegador realizar una petición ***GET sobre la url*** (nuestro ***HttpRequest***)

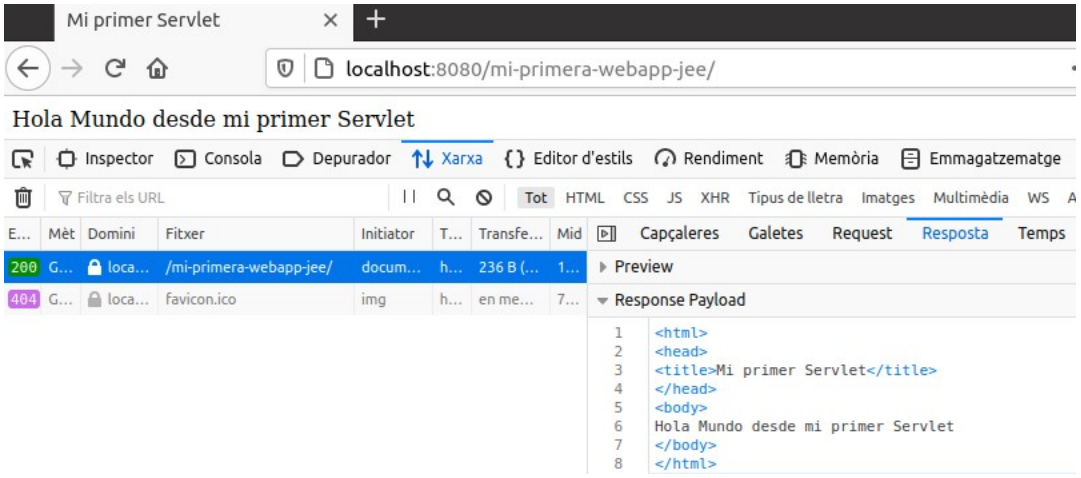


... continuamos **Entendiendo nuestra primera aplicación con Servlets:**

Y si nos vamos a la subpestaña *Respuesta\Contenido de la respuesta* se puede observar cual ha sido la respuesta del servidor. ***Nuestro servidor ha procesado la petición HttpRequest y ha respondido con un HttpResponse***.

Recordemos que el cliente (FrontEnd) solo vé lo que el server (BackEnd) le enviá y no vé como se genera esa información.

FrontEnd:



BackEnd:

LoginServlet.java

```
1  package webapp;
2
3  import java.io.IOException;
10
12  * Browser sends Http Request to Web Server
19
20 /*Java Platform, Enterprise Edition (Java EE) JEE8
32 @WebServlet(urlPatterns = "/login.do")
33 public class LoginServlet extends HttpServlet {
34     @Override
35     protected void doGet(HttpServletRequest request,
36             HttpServletResponse response) throws IOException {
37         PrintWriter out = response.getWriter();
38         out.println("<html>");
39         out.println("<head>");
40         out.println("<title>Mi primer Servlet</title>");
41         out.println("</head>");
42         out.println("<body>");
43         out.println("Hola Mundo desde mi primer Servlet");
44         out.println("</body>");
45         out.println("</html>");
46     }
47 }
```

... continuamos Entendiendo nuestra primera aplicación con Servlets:

En nuestro Servlet tenemos una ***anotación JEE8 (@WebServlet)*** que sirve para indicar la ***url de nuestro servlet (login.do)*** .

Adicionalmente, un servlet JEE8 tiene un método ***doGet*** que se debe sobreescribir y que es el encargado de recepcionar las peticiones GET en la url indicada. En nuestro ejemplo vemos que gracias ***PrintWriter*** conseguimos imprimir la salida, el fichero html que recibe el navegador web.

```
@WebServlet(urlPatterns = "/login.do")
public class LoginServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws IOException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Mi primer Servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("Hola Mundo desde mi primer Servlet");
        out.println("</body>");
        out.println("</html>");
    }
```

UD 1: Introducción a los lenguajes de servidor
4.- JEE: Servlets, JSP y JSTL

Desarrollo Web en Entorno Servidor - joseramon.profesor@gmail.com

... continuamos Entendiendo nuestra primera aplicación con Servlets:

En ***PHP*** , era el programador el que se encargaba de comprobar que tipo de petición se realizaba para saber que tarea realizar en función del tipo de petición ***($_SERVER['REQUEST_METHOD'])***.

En Java está más estructurado gracias a los Servlets:

```php
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Formulario: Nombre y apellidos</title>
    <style>label,input
        {
            display:block;
        }
    </style>
</head>
<body>
<?php
    if ($_SERVER['REQUEST_METHOD']==='POST'){
        print "<p>Hola ".$_POST['nombre']. " " . $_POST['apellidos']."</p>";
        print "<p>Si desea cambiar sus datos vuelva a rellenar el formulario</p>";
    }
?>
<form action="<?=$_SERVER['PHP_SELF'] ?>" method="POST">
    <label for="nombre">Nombre</label>
    <input type="text" name="nombre" value="">
    <label for="apellido">Apellidos</label>
    <input type="text" name="apellidos" value="">
    <input type="submit" value="Enviar">
</form>
</body>
</html>
```

```java
@WebServlet(urlPatterns = "/login.do")
public class LoginServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws IOException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Mi primer Servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("Hola Mundo desde mi primer Servlet");
        out.println("</body>");
        out.println("</html>");
    }
```

... continuamos Entendiendo nuestra primera aplicación con Servlets:

Si solicitamos http://localhost:8080/mi-primera-webapp-jee/login.do vemos que nos muestra la página inicial. Pero ahora la pregunta es:

¿Por que cuando tecleamos http://localhost:8080/mi-primera-webapp-jee acabamos realizando una petición GET a “login.do” ?

```java
@WebServlet(urlPatterns = "/login.do")
public class LoginServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws IOException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Mi primer Servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("Hola Mundo desde mi primer Servlet");
        out.println("</body>");
        out.println("</html>");
    }
```

... continuamos Entendiendo nuestra primera aplicación con Servlets:

Realmente esta pregunta ya la habíamos contestado antes. En el fichero web.xml le indicábamos que la página de bienvenida,o sea la página inicial, debía ser login.do.

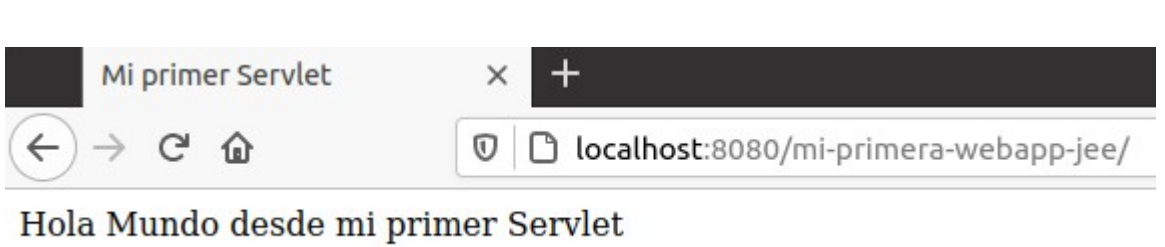
web.xml

```
1  <!-- webapp/WEB-INF/web.xml -->
2  <web-app xmlns="http://java.sun.com/xml/ns/javaee"
3      xsi:schemaLocation="http://java.sun.com/xml/ns/
4      version="3.0">
5
6      <display-name>Lista de alumnos</display-name>
7
8      <welcome-file-list>
9          <welcome-file>login.do</welcome-file>
10     </welcome-file-list>
11
12 </web-app>
```

En nuestro ejemplo la página HTML devuelta no es muy compleja, pero en aplicaciones reales pueden tener un tamaño considerable. Por tanto la pregunta ahora es:

¿No hay un sistema más sencillo o cómodo de escribir páginas HTML en Java que no sea creando los correspondientes “out.println”?

```java
@WebServlet(urlPatterns = "/login.do")
public class LoginServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws IOException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Mi primer Servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("Hola Mundo desde mi primer Servlet");
        out.println("</body>");
        out.println("</html>");
    }
```



Páginas JSP:

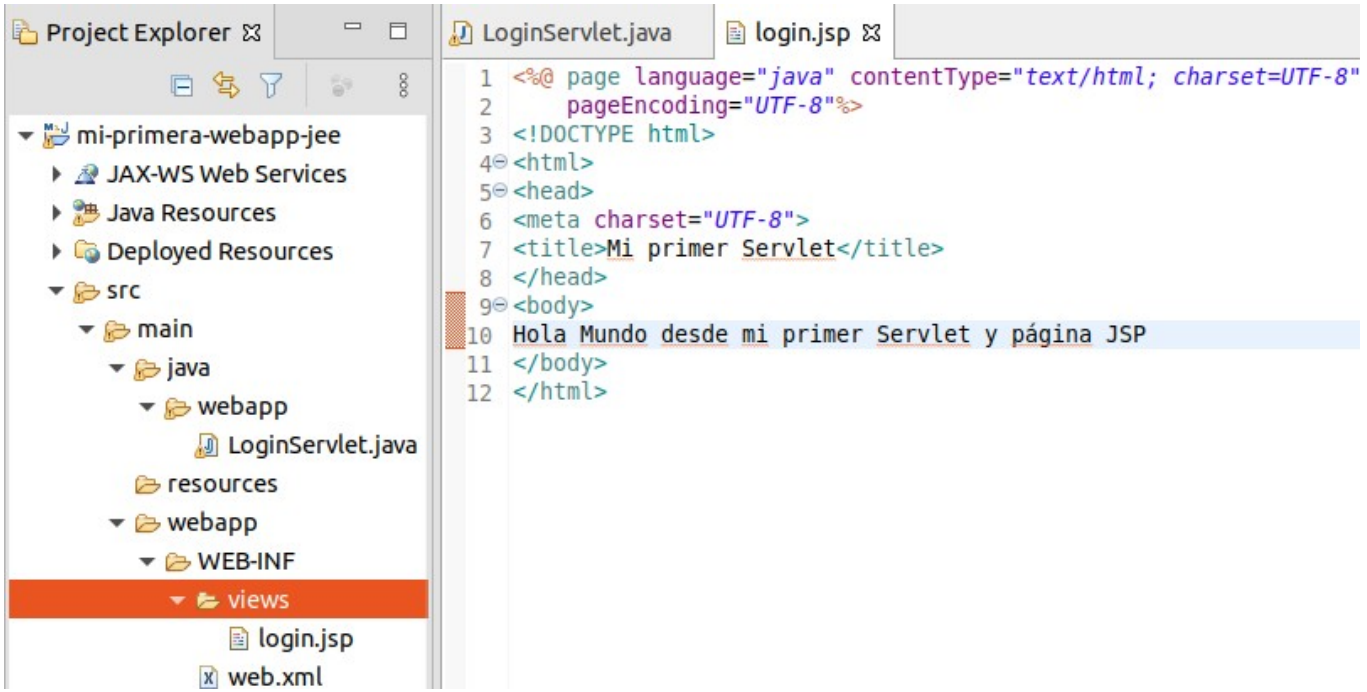
· ***JEE8*** cuando tiene que responder al navegador con una página HTML proporciona una ***solución muy elegante y sencilla, las páginas JSP (Java Server Pages)***.

· Las ***páginas JSP*** podriamos decir que son el ***equivalente a las páginas php*** que contienen HTML con trozos de código php embebido.

· La ***gran diferencia de las páginas JSP*** con respecto a las páginas php es que cuando las páginas JSP se compilan, ***se compilan internamente como Servlets.*** Y lo mejor de todo es que este ***proceso de compilación es transparente*** para el programador, no tiene que hacer nada.

... continuación **Páginas JSP:**

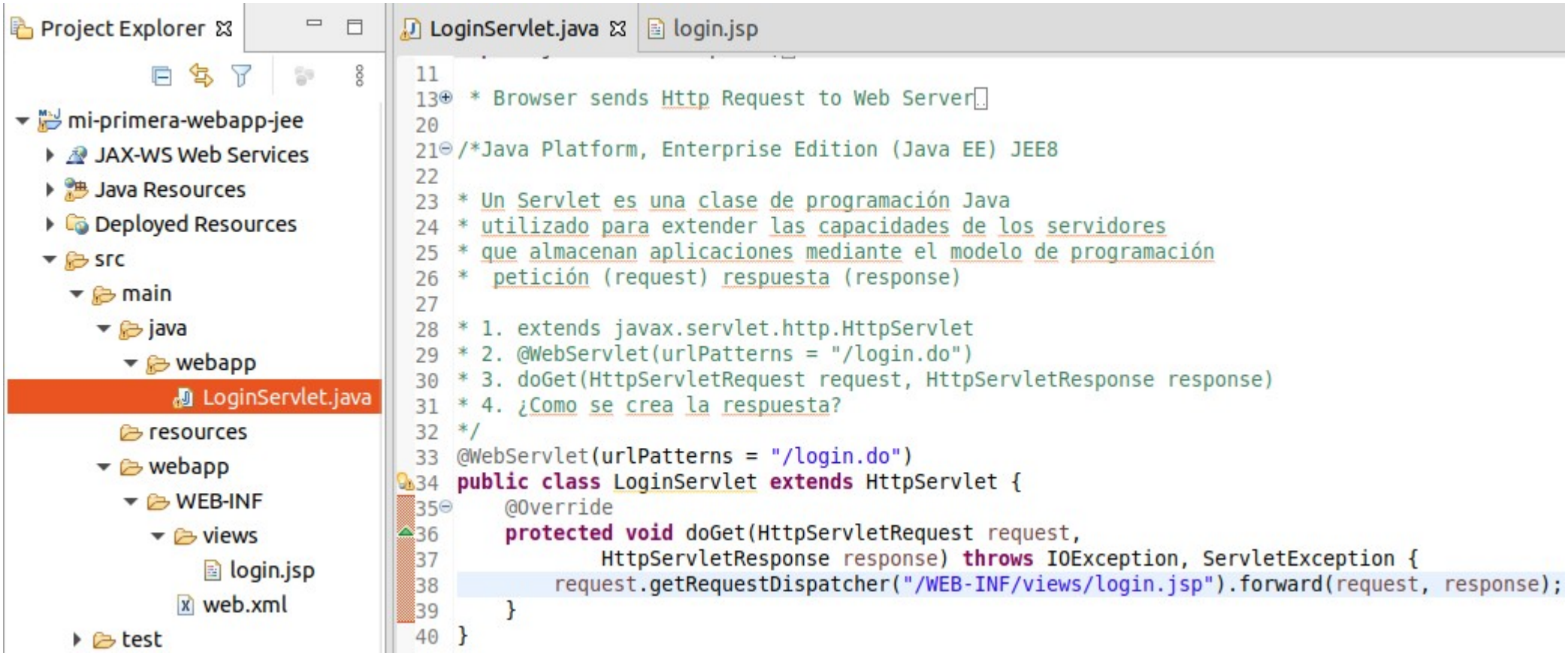
· Creemos nuestra primera página JSP (login.jsp). Para ello crearemos la carpeta views dentro de src\main\webapp\WEB-INF. Dentro copiaremos el HTML que teniamos en LoginServlet y le añadiremos la coletilla “y página JSP”.



¿Como le decimos a LoginServlet.java que ejecute la página JSP login.jsp?

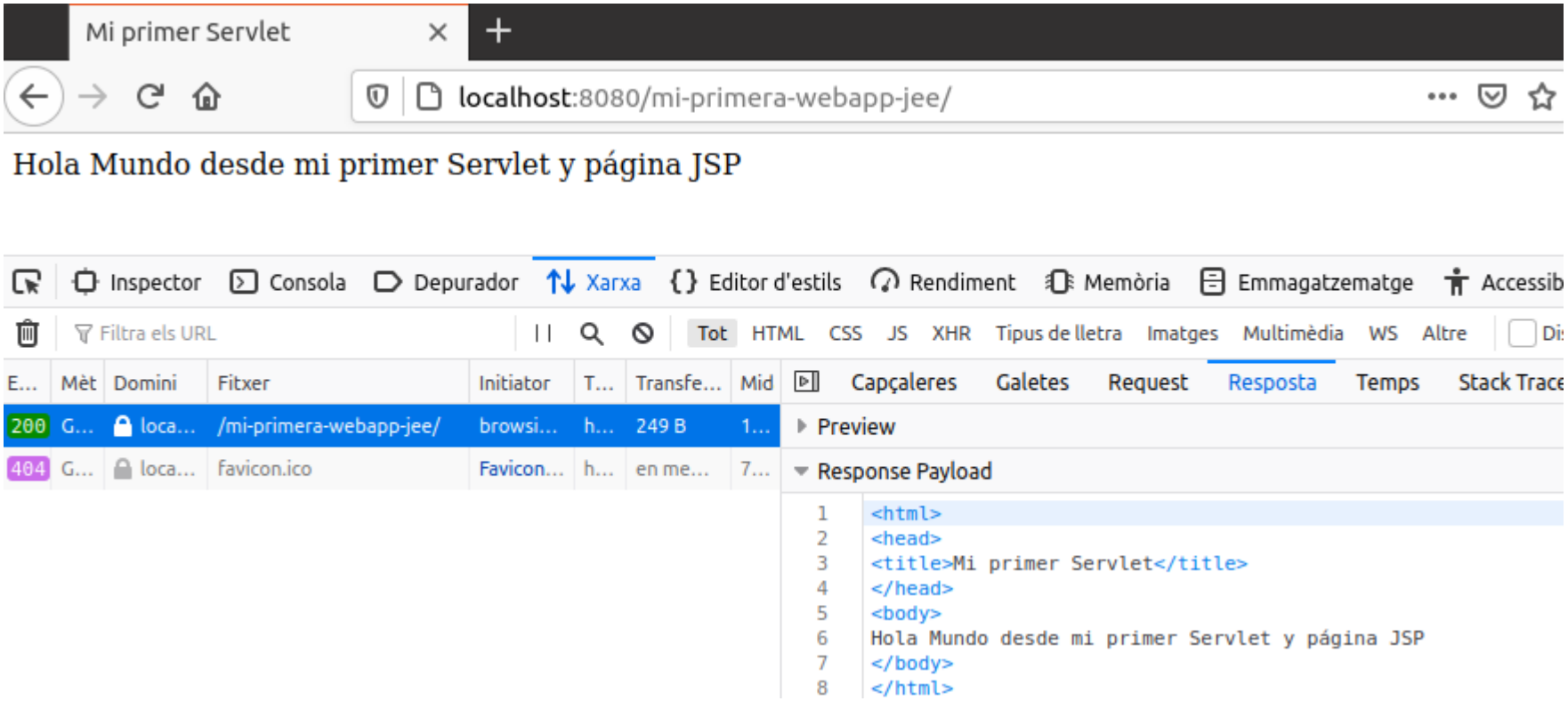
... continuación **Páginas JSP:**

· Para indicarle al Servlet que debe lanzar una página JSP utilizamos request.getRequestDispatcher("dirección página jsp").forward(request,response). Reescribamos el código de LoginServlet.java para que quede de la siguiente manera:



... continuación **Páginas JSP:**

· No hace falta volver a darle a “Run as”. Si guardamos y nos esperamos unos segundos a que Eclipse recargue el contexto del servidor de Tomcat podremos recargar la aplicación web en nuestro navegador y veremos el resultado:



... continuación **Páginas JSP:**

· Modifiquemos LoginServlet.java para que vaya a una página que no existe. Si nos esperamos unos segundos vemos que se recarga el contexto. Si intentamos acceder de nuevo a nuestra web ...

¿Que pasa? ¿Que devuelve Tomcat?

```
33 @WebServlet(urlPatterns = "/login.do")
34 public class LoginServlet extends HttpServlet {
35     @Override
36     protected void doGet(HttpServletRequest request,
37             HttpServletResponse response) throws IOException, ServletException {
38         request.getRequestDispatcher("/WEB-INF/views/loginNO_Existe.jsp").forward(request, response);
39     }
40 }
```

... continuación Páginas JSP:

· Cuando Tomcat no encuentra una página devuelve un ***Error 404*** que significa que no ha encontrado el recurso solicitado y si vamos a la pestaña con la respuesta en el navegador podemos ver que ha devuelto exactamente Tomcat:

¡¡La próxima vez que tengamos un error 404 ya sabemos porque!!

... continuación Páginas JSP:

· Si corregimos el error y esperamos a que se actualice en Tomcat podemos ver que el código devuelto es el ***200 cuando todo ha ido bien***:

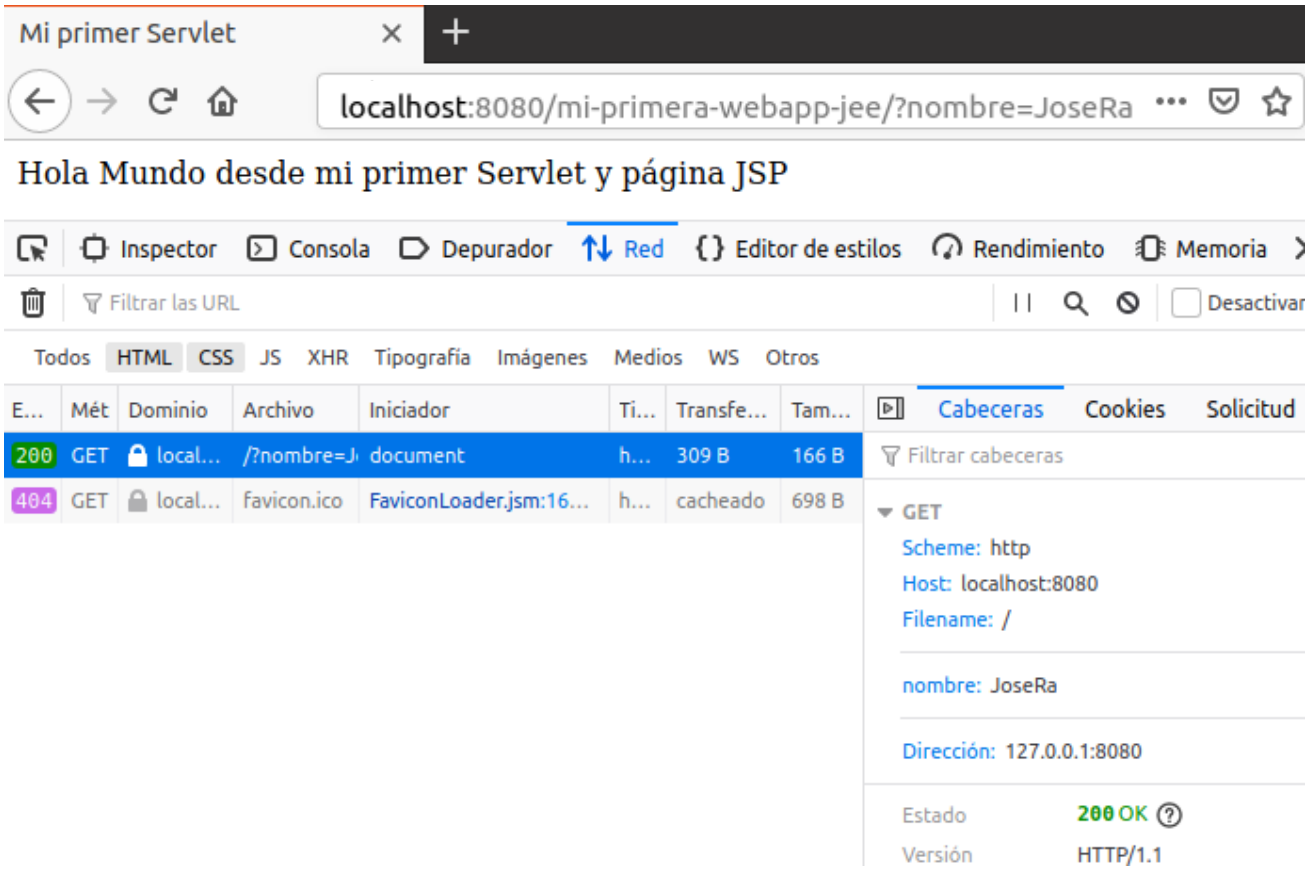
... continuación **Páginas JSP:**

· Hasta ahora hemos creado una aplicación que siempre devuelve la misma página web, pero :

¿ Como podemos pasarle parámetros a la página JSP para que utilice esos datos en la creación de la página HTML a devolver (Html dinámico)?

... continuación **Páginas JSP:**

· **1º Opción:** La primera opción es ***pasarle a la página JSP el parámetro al estilo php***, mediante la url. Sin cambiar nada en Eclipse añadimos en el navegador el siguiente parámetro a la dirección url "?nombre=nomAlumno". Si nos vamos a la subpestaña "Cabeceras" podemos ver que se ha pasado dicho parámetro en la llamada:



... continuación **Páginas JSP:**

· ***2º* Opción:** La segunda opción es ***pasarle a la página JSP el parámetro desde el Servlet***, para ello debemos leer el parámetro (request.getParameter()) y después añadirselo a los atributos de la petición (request.setAttribute()) :

```
33 @WebServlet(urlPatterns = "/login.do")
34 public class LoginServlet extends HttpServlet {
35     @Override
36     protected void doGet(HttpServletRequest request,
37             HttpServletResponse response) throws IOException, ServletException {
38         //Leemos el parámetro
39         String nombre= request.getParameter("nombre");
40         //Insertamos el parámetro en los atributos para que
41         //lo pueda leer la página JSP
42         request.setAttribute("nombre", nombre);
43         request.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(request, response);
44     }
45 }
```

¿ Como puede utilizar la página JSP los atributos recibidos?

... continuación **Páginas JSP:**

· La página JSP puede leer los parámetros mediante las ***expresiones del lenguaje (${nombreParametro})*** que permite Java. Modifiquemos el contenido de login.jsp para que quede como sigue:

login.jsp

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="UTF-8">
7  <title>Mi primer Servlet</title>
8  </head>
9  <body>
10 Bienvenido ${nombre} a mi primera página JSP
11 </body>
12 </html>
```

¿ Podemos introducir más código en las páginas JSP?

... continuación Páginas JSP:

· La página JSP puede contener más código en su interior mediante los llamados ***scriptlets que van entre los simbolos <% %> de forma similar a los trozos de código introducidos en las páginas php***.Modifiquemos el contenido de login.jsp para que quede como sigue y recarguemos la página web:

Recordemos que la dirección url de momento contendrá "?nombre=nomAlumno"

· ***No vamos a profundizar más en los scriptlets porque realmente son una mala práctica*** como veremos en el siguiente tema dedicado al modelo vista controlador (MVC). Solo comentar que en los scriptlets podríamos escribir en Java todo el código que quisiéramos.

login.jsp

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="UTF-8">
7  <title>Mi primer Servlet</title>
8  </head>
9  <%
10 System.out.println(request.getParameter("nombre")+
11         ": Esto se escribirá por consola");
12 for(int i=1;i<=3;i++){
13     System.out.println(i);
14 }
15 %>
16 <body>
17 Bienvenido ${nombre} a mi primera página JSP
18 </body>
19 </html>
```

Markers | Properties | Servers | Data Source Explorer | Snippets

```
mi-primera-webapp-jee [Maven Build] /usr/lib/jvm/java-13-openjdk-amd64/bin/java
JoseRa: Esto se escribirá por consola
1
2
3
```

... continuación **Páginas JSP:**

De momento, en el navegador hemos introducido el parámetro en la url para que lo lea el Servlet al recibir la petición (request), para que a su vez se lo reenvíe a la página jsp.

Ahora vamos a pedir el usuario y el password antes de entrar a nuestra página web. Pero …

¿ Porque es una mala práctica enviar los parámetros en la url de petición GET?

... continuación **Páginas JSP:**

Existe un grán ***problema de privacidad*** si los datos viajan en la url de la petición GET del navegador. Todos los routers por los que pasa la petición desde el navegador hasta el servidor podrían leer la url y utilizar dicha información para atacar nuestro sistema.

¿ Como podemos pedirle al usuario que introduzca el parámetro de otra manera?
¿Como podemos hacer que el Servlet lo lea para que a su vez se lo pase al fichero JSP?

... continuación **Páginas JSP:**

A continuación veremos que utilizando formularios y leyendo apropiadamente los parámetros podemos solucionar nuestro problema.

Empecemos modificando login.jsp para crear un formulario que pida el usuario y la contraseña en login.jsp (podemos copiar el texto de DWES_UD1_03_loginJSP) :

login.jsp

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="UTF-8">
7  <title>Login</title>
8  </head>
9  <body>
10 <form action=" login.do" method="post">
11 Introduzca su nombre: <input type="text" name="nombre"/>
12 Introduzca su contraseña:<input name="password" type="password" />
13 <input type="submit" value="Login">
14 </form>
15 </body>
16 </html>
```

... continuación **Páginas JSP:**

A continuación modificamos LoginServlet.java para leer los parámetros:

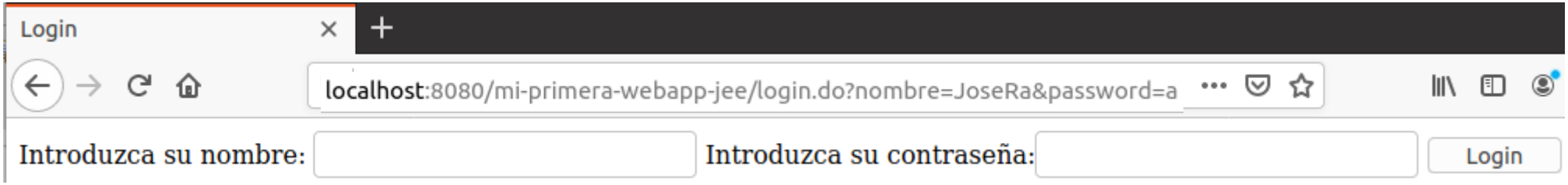
LoginServlet.java

```
25 * que almacenan aplicaciones mediante el modelo de programación
26 *  petición (request) respuesta (response)
27
28 * 1. extends javax.servlet.http.HttpServlet
29 * 2. @WebServlet(urlPatterns = "/login.do")
30 * 3. doGet(HttpServletRequest request, HttpServletResponse response)
31 * 4. ¿Como se crea la respuesta?
32 */
33 @WebServlet(urlPatterns = "/login.do")
34 public class LoginServlet extends HttpServlet {
35     @Override
36     protected void doGet(HttpServletRequest request,
37             HttpServletResponse response) throws IOException, ServletException {
38         request.setAttribute("nombre",request.getParameter("nombre"));
39         request.setAttribute("password",request.getParameter("password"));
40         request.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(request, response);
41     }
42 }
```

¿ Que pasaría si en login.jsp no le hubiéramos indicado que el método era POST? Prueba a pulsar en el botón “Login” en login.jsp habiendo quitado “method=POST”? ¿Que pasa?

... continuación **Páginas JSP:**

Como podemos observar al pulsar el botón login sin indicarle que es POST se realiza una llamada al Servlet al ***método GET con los parámetros en la url (mala práctica)***. Adicionalmente hemos vuelto a la página jsp donde se pide el usuario:

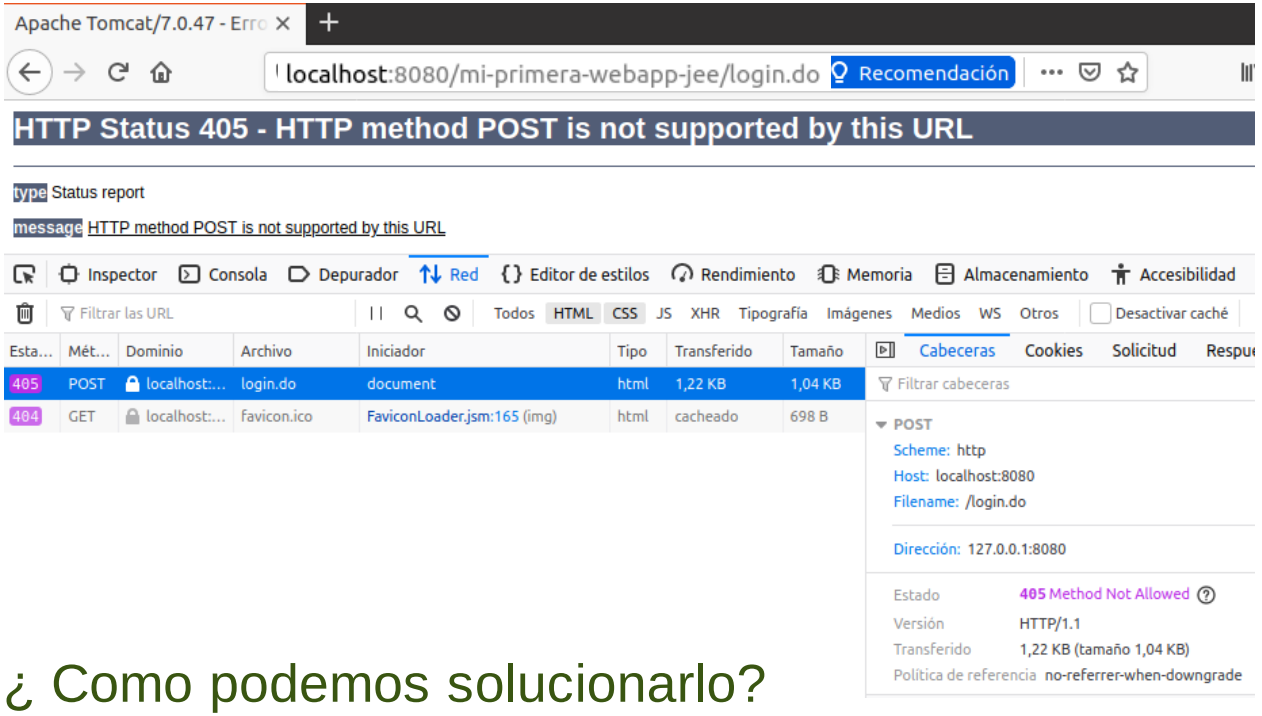


¿ Como podemos arreglarlo?
Modifica el formulario para que realice una petición POST como inicialmente. Ejecutalo y comprueba que pasa

... continuación Páginas JSP:

Si solo le indicamos al formulario que utilice el método POST lo que pasa es que Tomcat devolverá una respuesta indicando que ***no se esta utilizando un método permitido (405 - HTTP method POST is not supported by this URL***).

Este error ocurre porque ***tenemos un Servlet que acepta peticiones GET*** en login.do, pero es el único tipo de petición que hemos configurado. ***No hemos configurado que hacer si se recibe una petición POST.***



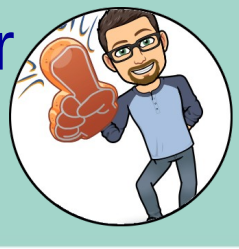
¿ Como podemos solucionarlo?

... continuación **Páginas JSP:**

Para solucionarlo debemos realizar **2 pasos:**

1ºcrear un método doPost

2º Crear una página distinta a login.jsp donde enviar al usuario después de logearse (bienvenida.jsp).



... continuación **Páginas JSP:**

Para solucionarlo debemos realizar **2 pasos:**

1ºcrear un método doPost

2º Crear una página distinta a login.jsp donde enviar al usuario después de logearse (bienvenida.jsp).

Empecemos por crear doPost en LoginServlet.java y modificar doGet:

LoginServlet.java

```
 1 package webapp;
 2
 3 import java.io.IOException;
11
13  * Browser sends Http Request to Web Server
20
21 /*Java Platform, Enterprise Edition (Java EE) JEE8
33 @WebServlet(urlPatterns = "/login.do")
34 public class LoginServlet extends HttpServlet {
35     @Override
36     protected void doGet(HttpServletRequest request,
37             HttpServletResponse response) throws IOException, ServletException {
38         request.getRequestDispatcher("/WEB-INF/views/login.jsp").forward(request, response);
39     }
40     @Override
41     protected void doPost(HttpServletRequest request,
42             HttpServletResponse response) throws IOException, ServletException {
43         request.setAttribute("nombre",request.getParameter("nombre"));
44         request.setAttribute("password",request.getParameter("password"));
45         request.getRequestDispatcher("/WEB-INF/views/bienvenida.jsp").forward(request, response);
46     }
47 }
```

... continuación **Páginas JSP:**

Para solucionarlo debemos realizar **2 pasos:**

1ºcrear un método doPost

2º Crear una página distinta a login.jsp donde enviar al usuario después de logearse (bienvenida.jsp).

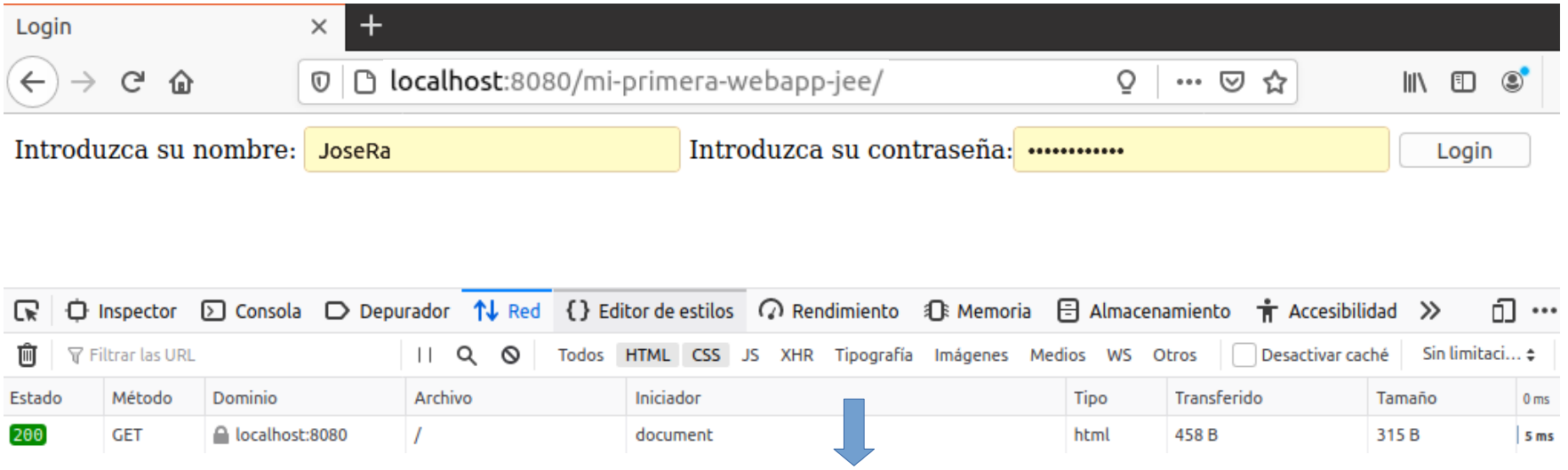
Ahora debemos crear bienvenida.jsp en el mismo directorio donde esta login.jsp:

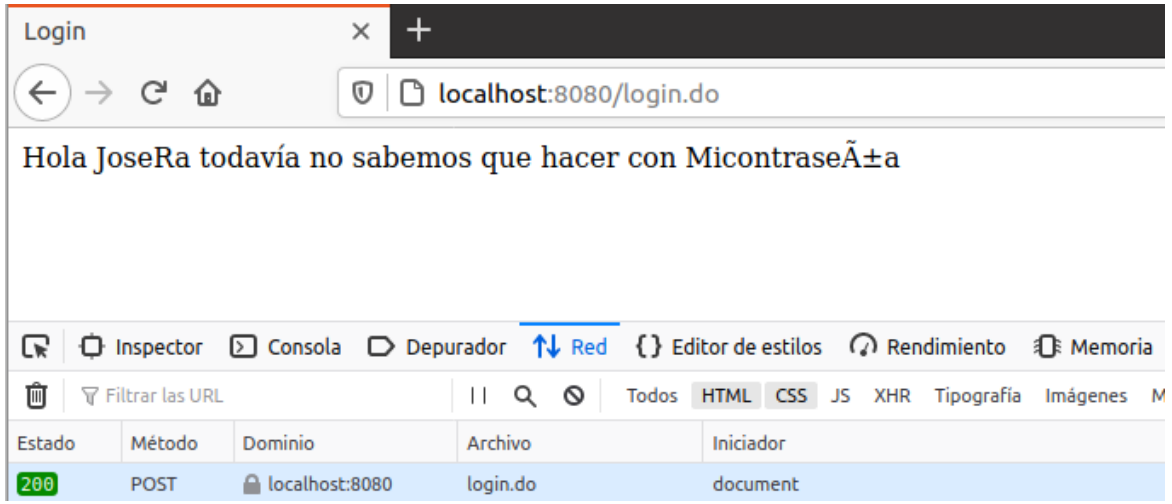
bienvenida.jsp

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="UTF-8">
7  <title>Login</title>
8  </head>
9  <body>
10 Hola ${nombre} todavía no sabemos que hacer con ${password}
11 </body>
12 </html>
```

... continuación **Páginas JSP:**

Como podemos observar si recargamos http://localhost:8080 el funcionamiento de nuestra primera aplicación web es la deseada:





EJERCICIO:

Sigue todos los pasos de los PDF y sube la aplicación final al moodle.

Para ello:

1º Haz un "Run As \Maven Clean" para dejar solo los fichero fuentes y quitar momentaneamente los necesarios para ejecutar la aplicación (dependencias).

2º Comprime la carpeta de tu aplicación y ponle como nombre al fichero comprimido UD1_practica3_nombreAlumno.tar.gz donde nombreAlumno es el nombre del alumno que entrega la práctica.

3º Súbela al moodle.

IMPORTANTE: No comprimir en RAR, porque Ubuntu no lo lee bien y en clase tenemos Ubuntu. Si tuviesemos Windows, podemos comprimir en ZIP.