Carrera: Licenciatura en Sistemas

Materia: Orientación a Objetos II

Equipo docente:

Titular: Prof. María Alejandra Vranić alejandravranic@gmail.com

Ayudantes: Prof. Leandro Ríos leandro.rios.unla@gmail.com

Prof. Gustavo Siciliano gussiciliano@gmail.com

Prof. Romina Mansilla romina.e.mansilla@gmail.com

Año: 2018

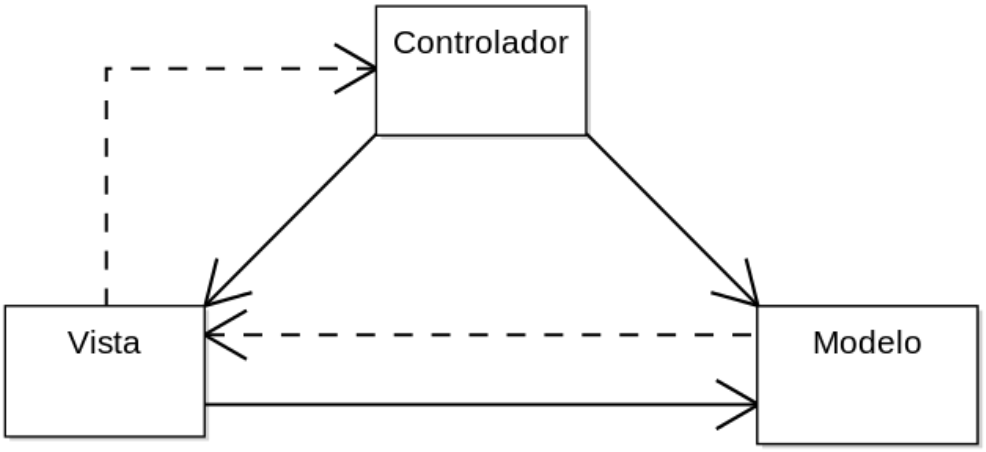
IDE de para proyectos web: https://eclipse.org/downloads/

MVC, Java Servlets y JSP

MVC (Model View Controller, Modelo Vista Controlador)

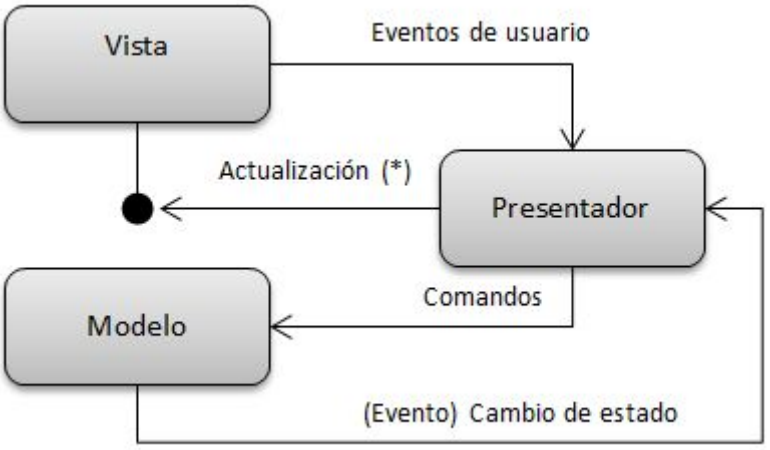
MVC es un patrón de diseño que nos permite separar la presentación (la forma en que los datos se muestran al usuario) de nuestro modelo, es decir, las clases que implementan los datos y reglas de negocio. Esto se hace para reducir el acoplamiento entre vista y modelo de negocios, y facilitar la migración de nuestros sistemas de un tipo de presentación o otro. Por ejemplo, una aplicación desarrollada para funcionar con una interfaz de usuario tipo GUI podría incorporar además vistas tipo web, para ser consultadas a través de Internet. Si nuestra aplicación tuviera el código de presentación muy acoplado con el modelo de negocio o con los datos, para agregar una vista distinta deberíamos modificarlos también.

En MVC, la vista se desacopla del modelo por medio de lo que se denomina un controlador. Éste módulo de software funciona como mediador entre el usuario que realiza peticiones y el modelo que debe satisfacerlas. El controlador recibe las peticiones del usuario y decide qué datos debe pedir al modelo para luego seleccionar la vista más adecuada, completarla con los datos devueltos por el modelo y presentarla al usuario. También recibe los cambios de estado que el usuario quiere provocar en el modelo e invoca las clases y métodos necesarios en el mismo para que esto ocurra.



En el patrón MVC puro, la vista observa al modelo permanentemente por si ocurren cambios que deba reflejar. Es decir, si algo cambia en el modelo, la vista se entera y refleja ese cambio sin que el usuario deba pedir que se refresque la vista. Cuando se desarrolló MVC, en los años 70, se hizo pensando en interfaces de usuario tipo GUI, donde esta dependencia entre la vista y el modelo es más simple de implementar. De todos modos, tiene el problema de acoplar la vista al modelo (el modelo debe conocer detalles de la vista para poder notificarla de cambios) y disminuye la cohesión, ya que el modelo, además de representar datos y reglas de negocio tiene que implementar funcionalidad para notificar a la vista.
En un entorno web, este acoplamiento entre la vista y el modelo es aún más complicado de resolver; de hecho, nos impide utilizar lo que se denomina "vista pasiva". Una vista pasiva es, por ejemplo, una página web generada dinámicamente utilizando sólo HTML. Si ocurre un cambio en el modelo, no hay forma de que la vista se entere, ya que la misma no tiene forma de interrogar al modelo por los cambios que pudieran haber ocurrido. En este caso, el usuario deberá refrescar el navegador periódicamente para ver reflejados los cambios en el modelo.

Para evitar estos problemas con el patrón MVC, en los años 90 se le introdujo una modificación: El controlador mediaría exclusivamente entre la vista y el modelo; es decir: la vista no tiene acceso directo al modelo ni éste a la vista. Este tipo de controlador que media absolutamente toda la interacción entre el modelo y la vista se denominó Presentador, y el patrón de diseño pasó a llamarse MVP, Model View Presenter o Modelo Vista Presentador.



Si bien en este gráfico puede observarse que el modelo envía eventos de cambio de estado al presentador, lo que significa que cambiamos la dependencia de la vista con la del controlador, ello puede evitarse haciendo que el presentador consulte periódicamente al modelo por cambios. Si los hubo, procede a actualizar la vista. Este esquema nos permite utilizar una vista pasiva que es renovada periódicamente por el presentador, o una vista activa que asume la responsabilidad de consultar periódicamente al presentador por cambios y éste a su vez consulta al modelo. De este modo terminamos con la dependencia del modelo con el presentador y la vista, aunque éstos últimos aún tengan dependencias con el modelo.

Lo que usualmente se denomina MVC en realidad es algún tipo de MVP (hay una multitud de variantes de MVC y MVP, un artículo interesante sobre la historia y variaciones en estos patrones puede encontrarse en un artículo de Martin Fowler: http://martinfowler.com/eaaDev/uiArchs.html, en inglés) pero la costumbre ha hecho que se los denomine MVC de manera genérica.

Nos encontramos una vez más ante una estrategia de dividir la funcionalidad en capas para disminuir la complejidad y fomentar la reutilización del software. Ante un escenario de cambio de vista o al incorporar una nueva, el programador no debe preocuparse por el modelo y puede enfocar su atención en agregar vistas y controladores. Del mismo modo, el mismo modelo servirá para aplicaciones con interfaces de usuario radicalmente distintas, ya sea web, GUI o de dispositivos móviles. Así como el patrón DAO independiza al modelo de la persistencia de datos, el patrón MVC lo independiza de la presentación de los mismos.

Java Servlets

Ya sabemos que el patrón MVC (o MVP, utilizaremos ambos nombres indistintamente, lo mismo que con controlador y presentador) nos permite desacoplar el modelo de la vista. Esto es particularmente útil para aplicaciones con interfaz web, ya que la interfaz de usuario suele encontrarse físicamente separada del modelo de negocios y sus datos. Un desacoplamiento total no sólo es beneficioso para reutilizar el modelo en otros escenarios, sino que nos facilita la implementación de la presentación de datos y resultados, ya que al no haber dependencias directas entre modelo y vistas, la separación física entre ambos no ofrece más problemas que los derivados de la implementación del modelo de comunicación entre ellos, lo que en el caso de las aplicaciones web se encuentra resuelto por el stack TCP/IP.

Una aplicación web implica que el servidor que la aloje tiene que poder ejecutarla al recibir peticiones para la misma. Tendremos un punto de entrada a la aplicación, representado por un URL, y el servidor deberá saber cómo invocar al controlador correspondiente. Éste a su vez invocará a nuestro modelo y seleccionará la vista más adecuada para mostrar los datos al cliente. En general, las vistas corren en el cliente y los controladores y el modelo corren en el servidor. De todos modos, tanto los controladores como el modelo (como así también los datos) pueden estar en servidores distintos, conformando una arquitectura física multicapas.

En Java, la forma de ejecutar programas en el servidor web es a través de lo que se denomina Java servlets. El nombre servlet tiene su origen en los applets Java, que son aplicaciones java que corren en el navegador de internet; seguramente todos se han encontrado con uno de ellos alguna vez. Fue una de las primeras ideas para agregar dinamismo a las páginas web, usualmente estáticas (esto ocurría en los años 90), antes de Javascript y Flash. Cuando se vio la necesidad de que Java corriera también en el servidor para reemplazar a los scripts CGI, pareció lo más natural denominarlos servlets.

Un servlet es una clase Java que se carga y ejecuta en lo que llamamos un contenedor de servlets, que generalmente forma parte de un servidor web, aunque no necesariamente. Éste contenedor de servlets ofrece un medio para que los mismos puedan ejecutarse (una máquina virtual java principalmente); para recibir configuración y parámetros y para poder enviar respuestas al servidor. Cuando el servidor recibe una petición que sabe que está relacionada con un servlet, le pasa el pedido al contenedor de servlets, que instancia el servlet, le pasa el control y cuando este devuelve una respuesta (por lo general en forma de página web, pero no siempre), la pasa al servidor para que este la envíe al navegador del cliente. Vemos que el servlet media la comunicación entre el cliente y nuestro modelo; por ello es evidente que el papel principal que ocupa nuestro servlet es el del controlador de MVC.

Al ser el servlet una clase java, puede instanciar o comunicarse con otros objetos java (como los que conforman nuestro modelo) e invocar métodos de los mismos para satisfacer el requerimiento del cliente.

Esta descripción en realidad es la de un servlet especializado en el protocolo HTTP (de la clase HttpServlet). Un servlet genérico (clase Servlet,

superclase de HttpServlet) no necesariamente precisa un servidor web y puede emplearse para prestar cualquier servicio.

Ciclo de vida de un servlet

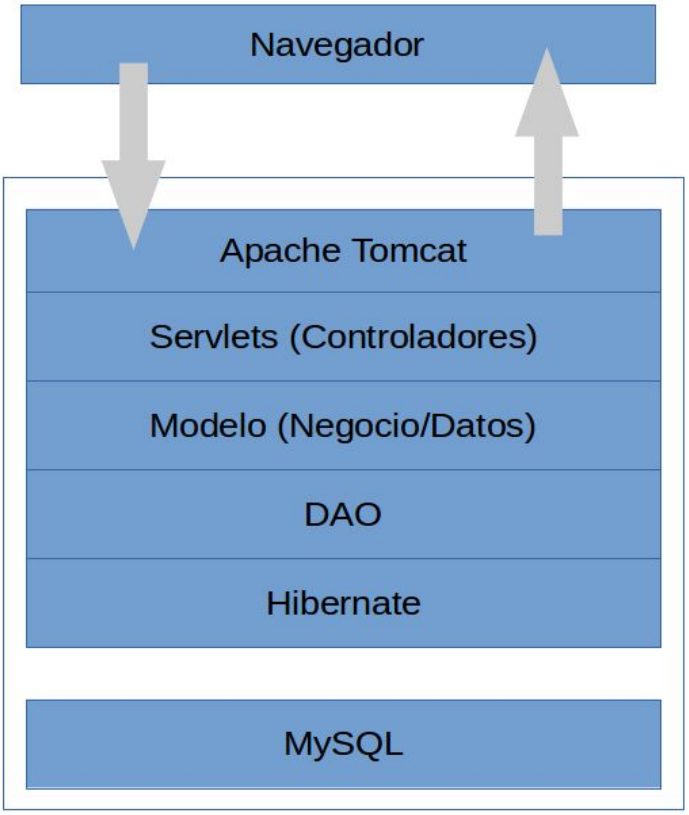
Un servlet debe implementar la interfaz javax.servlet.Servlet, que define su ciclo de vida. Este no se instanciará hasta que exista una petición de servicio para el mismo; cuando el servidor web recibe un pedido que debe trasladar a un servlet, primero lo instanciará llamando a su constructor y luego llamará al método init, con un parámetro de la clase ServletConfig. Esta clase encapsula varios parámetros de configuración necesarios para inicializar el servlet. El contenedor de servlets se encarga de instanciar y pasar ese parámetro y a menos que necesitemos cambiar la inicialización del servlet, no necesitamos tocarlo.

Una vez inicializado, el contenedor de servlets invocará al método service, que recibe dos parámetros: Uno de entrada, de la clase ServletRequest, que trae los parámetros con los que se realizó la petición de servicio y otro de salida, de la clase ServletResponse, donde devolveremos el resultado de la invocación del método. El método service se invocará tantas veces como peticiones existan. Finalmente, cuando se deba eliminar el servlet, se invocará al método destroy(), dentro del cual deberemos colocar las tareas de liberación de recursos que el servlet pudiera tener tomados y hacer una salida prolija.

La clase javax.servlet.http.HttpServlet, además de respetar el ciclo de vida del servlet a través de la implementación de la interfaz javax.servlet.Servlet, agrega otros métodos para facilitar el manejo del protocolo HTTP. Así, además de implementar el método service, agrega otros especializados para el manejo de este protocolo, como por ejemplo doGet que se invoca ante una petición HTTP GET, doPost para HTTP POST, etc.

Para que se nos aclaren un poco más estos conceptos, desarrollaremos una aplicación de ejemplo que demuestre la utilización de un servlet simple, utilizando Apache Tomcat como servidor web y contenedor de servlets, Eclipse como IDE y MySQL como servidor de base de datos. El modelo de negocio será el mismo en el que hemos trabajado durante la cursada: nuestro sistema de préstamos.

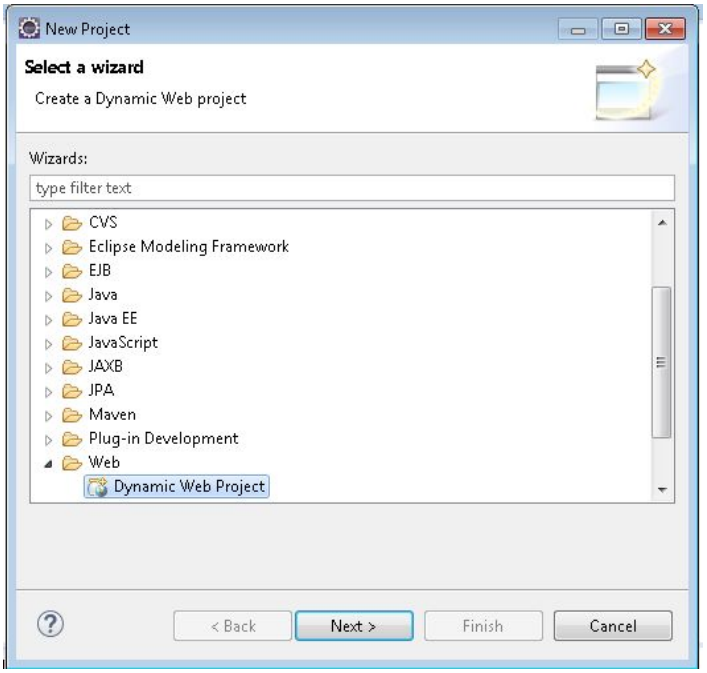
Nuestra aplicación irá creciendo a medida que incorporemos conceptos: Por lo pronto, la vista se generará completamente en el controlador; hay otras formas de generar la vista, que veremos más adelante. Mantendremos el diseño dividido en capas que hemos implementado, para ver con un ejemplo sencillo cómo se invocan los controladores desde el cliente y cómo interactúan con el modelo.



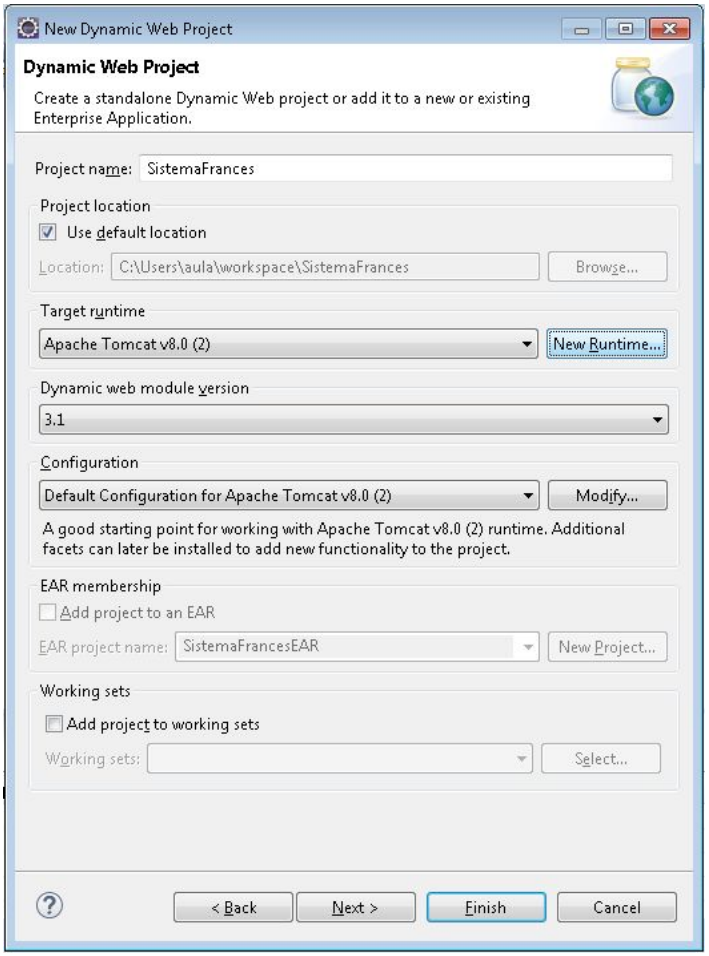
Desarrollo con Eclipse y Apache Tomcat.

Vamos a generar nuestro proyecto web. En la guía que sigue a continuación, hemos utilizado Eclipse Luna para desarrolladores Java EE, Java 8 y Apache Tomcat 8.0. Pueden utilizarse otras versiones, pero los menúes y opciones pueden variar ligeramente.

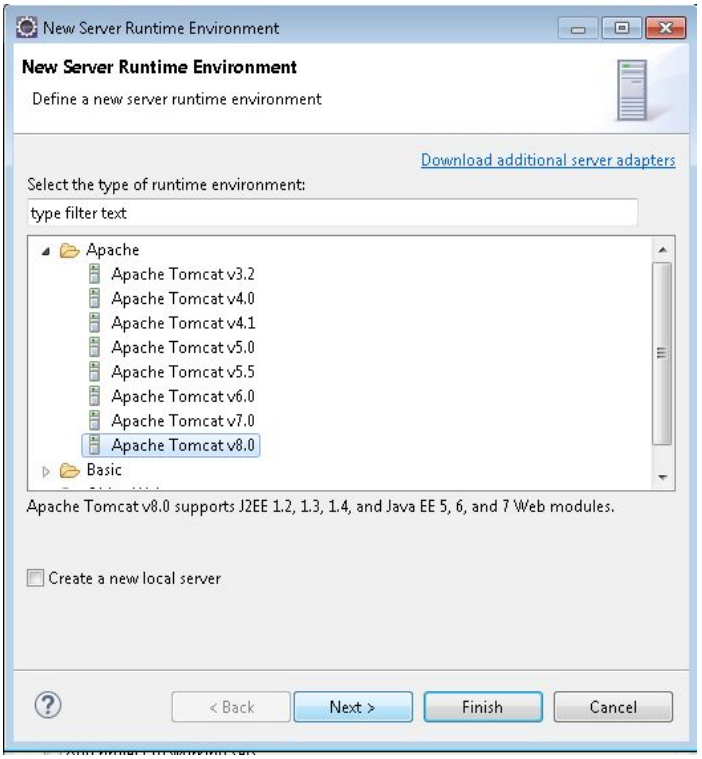
Comenzamos generando un nuevo proyecto del tipo "Dynamic Web Project" (Proyecto Web Dinámico)

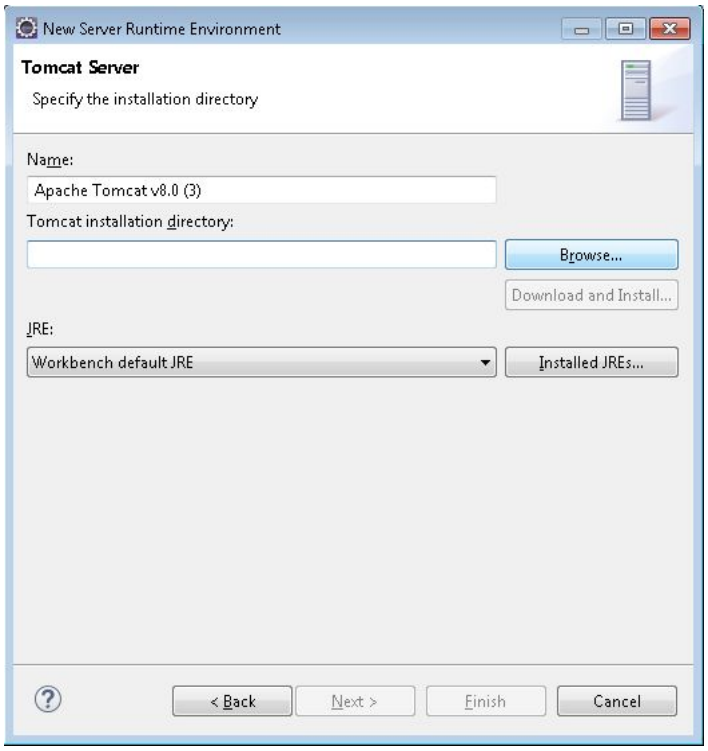


Y completamos los datos necesarios para crear nuestro proyecto:

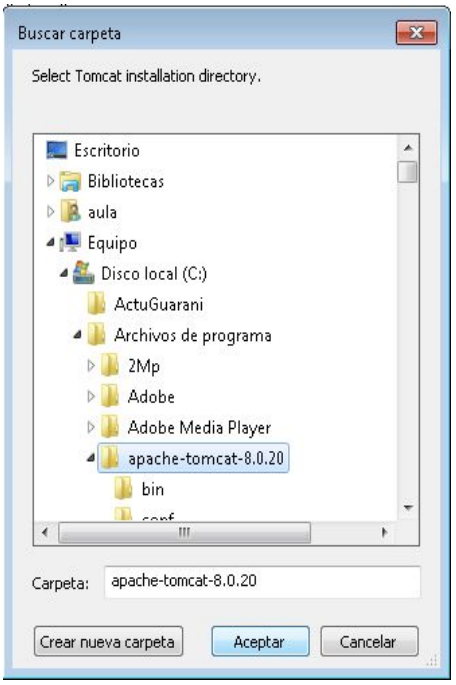


Si no existe aún una instancia de Apache configurada (lo más probable), la configuramos haciendo click en "New Runtime" y completando los datos necesarios:





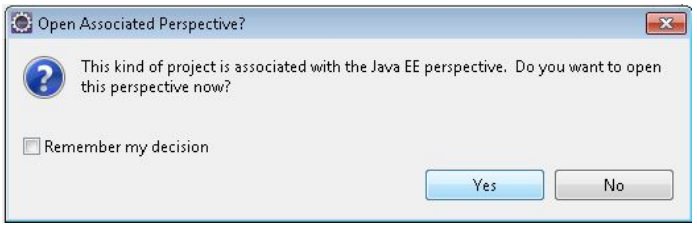
Hacemos click en el botón "Browse" para indicar la ruta a la instalación de Apache:



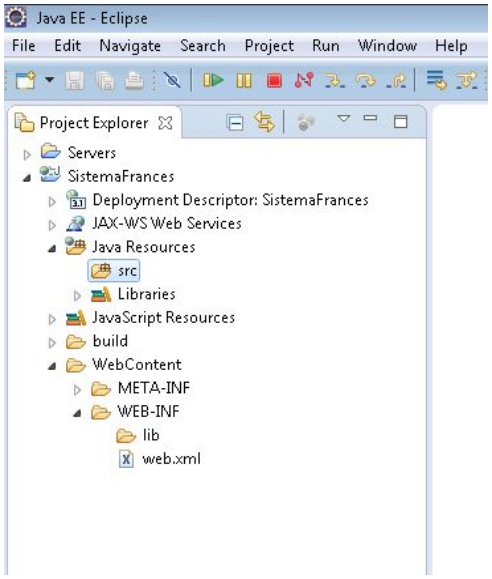
Aceptamos, y en la ventana anterior damos click al botón "Finish"

En el diálogo "New Dynamic Web Project" introducimos el nombre del proyecto (en este caso "SistemaFrances"), hacemos click en Next dos veces y marcamos la opción "Generate web.xml deployment descriptor". Finalmente, hacemos click en el botón "Finish" para crear nuestro proyecto. Si Eclipse nos

avisa que este tipo de proyecto está asociado con la perspectiva Java EE y nos pregunta si queremos abrirla, le contestamos que sí.



Eclipse debe haber creado una estructura de proyecto como la siguiente:



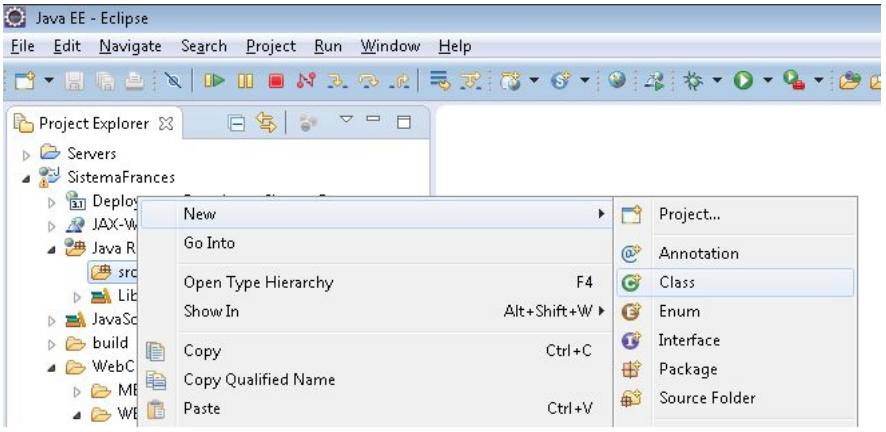
Cómo agregar nuestra lógica de negocios preexistente:

A continuación agregaremos las clases y librerías que necesita nuestro sistema de préstamos para funcionar:
hasta ahora, para agregar las librerías de Hibernate creábamos una carpeta llamada lib en la carpeta del proyecto, a la misma altura que src y copiábamos las librerías de Hibernate allí. Luego íbamos a las propiedades del proyecto, Java Build Path, elegíamos la pestaña Libraries y agregábamos las liberías al proyecto con el botón "Add External JARs". En este caso el proceso es similar, lo único que cambia es que la carpeta "lib" con las librerías debe estar ubicada dentro de la carpeta WEB-INF del proyecto, como se ve en la imagen superior.

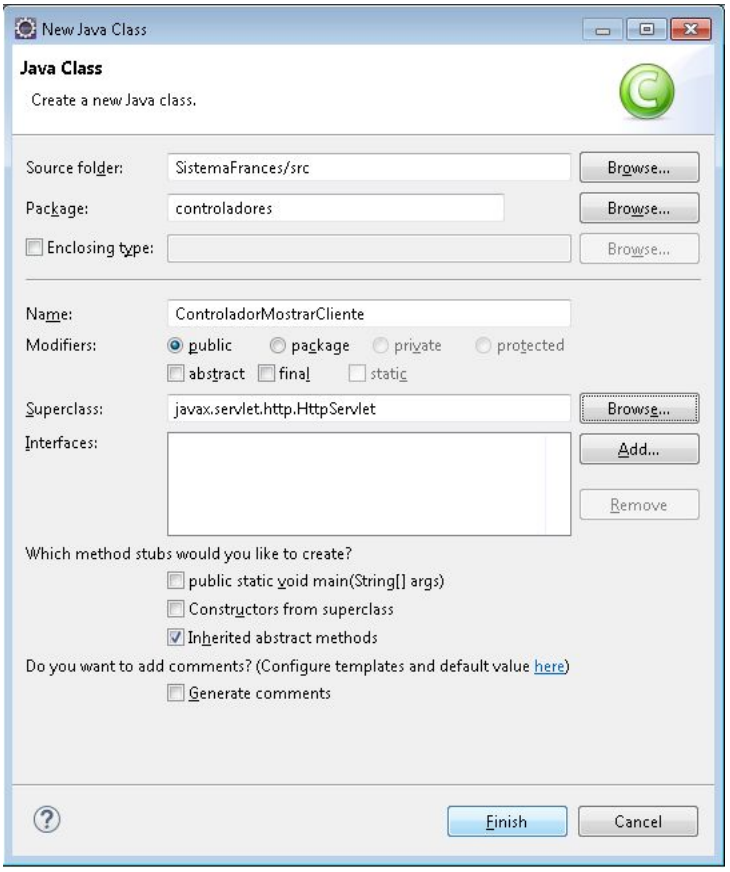
Para agregar las clases que conforman nuestra lógica de negocios, copiamos las carpetas representando los paquetes necesarios del proyecto anterior en la carpeta src del proyecto. No olvidar el archivo de configuración hibernate.cfg.xml ni los mapeos. Una vez que copiamos todo, hacemos un Refresh del proyecto para que refleje los cambios.

Cómo agregar y configurar un controlador:

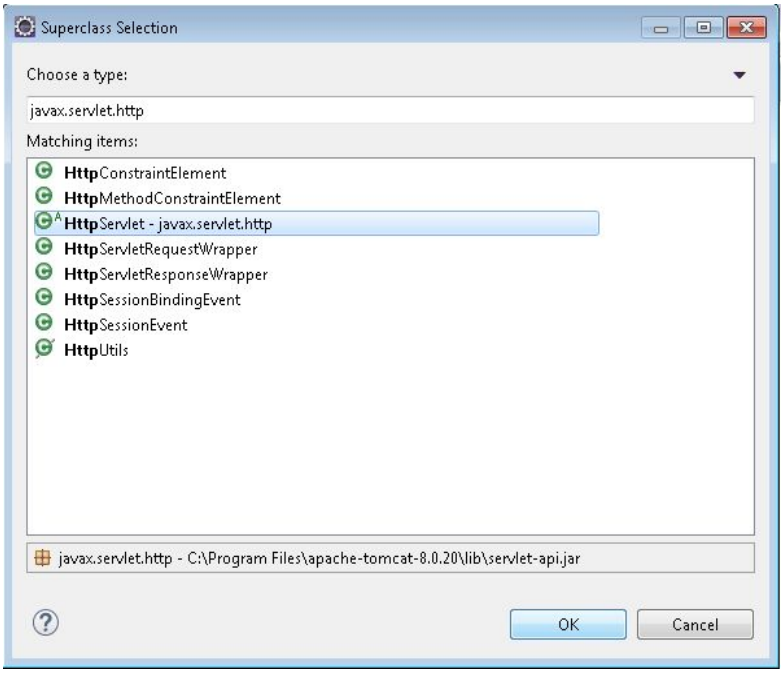
Para agregar un controlador a nuestro proyecto, hacemos click derecho sobre el paquete src y elegimos New/Class



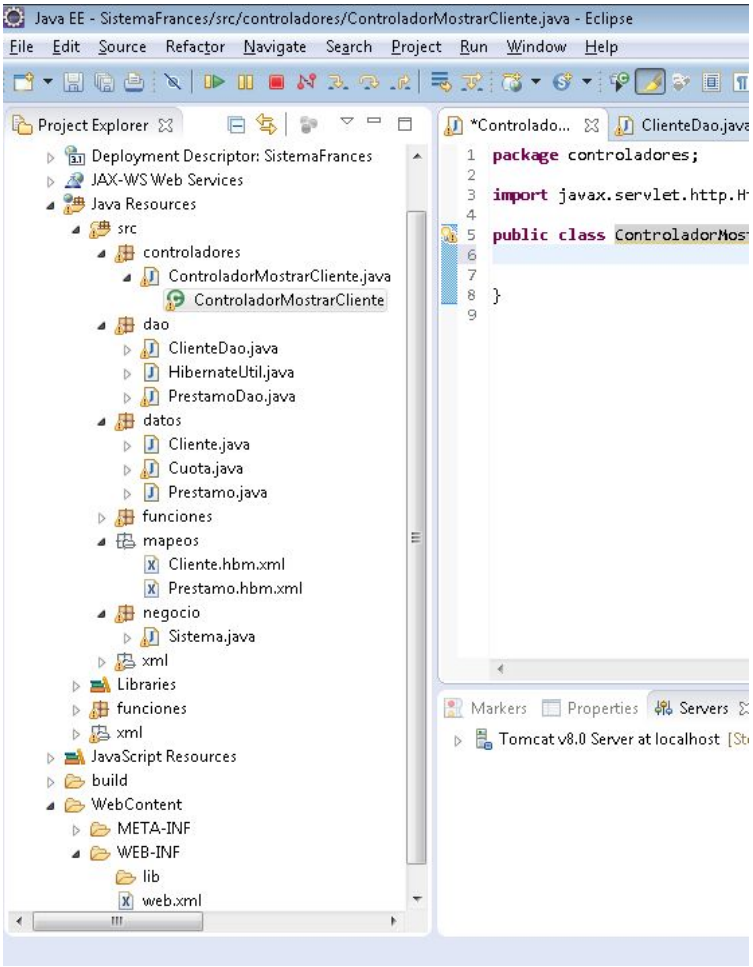
Se abre el ya conocido diálogo para crear una nueva clase:



Allí ingresamos el nombre del paquete al que pertenecerá nuestra nueva clase (controladores) , el nombre de la misma (ControladorMostrarCliente) y la superclase de la misma. Por defecto nos ofrece java.lang.Object, pero deberemos cambiarla a javax.servlet.http.HttpServlet. Para ello, damos click sobre el botón "Browse" que se encuentra al lado del campo de texto "Superclass"



Una vez seleccionada la superclase, damos click sobre OK, y sobre Finish en el diálogo de nueva clase java. Debe quedarnos una estructura de proyecto similar a ésta:



Código del controlador ControladorMostrarCliente:

```
package controladores;

import javax.servlet.http.HttpServlet;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.hibernate.HibernateException;
import datos.Cliente;
import negocio.ClienteABM;

public class ControladorMostrarCliente extends HttpServlet {

	protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		procesarPeticion(request, response);
	}

	protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		procesarPeticion(request, response);
	}

	private void procesarPeticion(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
		response.setContentType("text/html;charset=UTF-8");
		try {

			int dni = Integer.parseInt(request.getParameter("dni"));
			ClienteABM clienteAbm=new ClienteABM();
			Cliente cliente=clienteAbm.traerCliente(dni);

			response.setStatus(200);
			PrintWriter salida = response.getWriter();
			salida.println("");
			salida.println("<!DOCTYPE 4.01 Transitional//EN\">");
			salida.println("<HTML>");
			salida.println(" <HEAD>");
			salida.println(" <TITLE>Sistema Francés</TITLE>");
			salida.println(" </HEAD>");
			salida.println(" <BODY>");
			salida.println("    Apellido: "+cliente.getApellido()+"<BR>");
			salida.println("    Nombre  : "+cliente.getNombre()+"<BR>");
			salida.println("    DNI     : "+cliente.getDocumento()+"<BR>");
			salida.println("    ID      : "+cliente.getIdCliente()+"<BR>");

			salida.println("<A
href=\"/SistemaFrances/cliente.html\">Volver...</A>");
			salida.println(" </BODY>");
			salida.println("</HTML>");
		}
		catch (Exception e) {
			response.sendError(500, "El DNI Ingresado no existe en la base de
datos.");
		}
	}
}
```

Éste controlador implementa los métodos doPost y doGet. Ambos delegan el tratamiento las peticiones al método procesarPeticion, ya que ambos tipos de petición se procesan del mismo modo. Lo único que cambia es la forma en la que el navegador envía los parámetros de la petición al servidor: en el caso de GET se envían como parte de la URL, y en el caso de POST en el formulario. La diferencia es que en el primer caso son visibles, mientras que en el segundo no. Del lado del servlet los parámetros se reciben siempre en el objeto HttpServletRequest, sin importar cómo se hayan enviado.

Lo que este controlador hace es sencillo: recibe una petición con un DNI como parámetro, con ese DNI invoca al modelo para que le devuelva un cliente y con los datos del mismo genera el HTML que devolverá el servidor. Vamos a verlo paso a paso:

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        procesarPeticion(request, response);
}
```

La implementación de doPost recibe la petición (request) y nos ofrece un parámetro de salida para que devolvamos la respuesta. Lo único que hace es pasar ambas a procesarPeticion para que éste último se encargue de generar la respuesta.

Lo primero que hace procesarPeticion es

```
response.setContentType("text/html;charset=UTF-8");
```

Con lo que le indica al objeto response (de la clase HttpServletResponse) que el contenido que va a escribir en el mismo es del tipo text/html (una página HTML) y que estará codificado con el juego de caracteres UTF-8.

A continuación extrae el parámetro dni de la petición:

```
int dni = Integer.parseInt(request.getParameter("dni"));
```

con el que consulta al modelo por el cliente que tenga ese DNI. Seguidamente, establece el estado de la respuesta en 200 (ok)

```
response.setStatus(200);
```

y obtiene el objeto PrintWriter correspondiente a la respuesta:

```
PrintWriter salida = response.getWriter();
```

El objeto PrintWriter nos permite escribir texto en un stream de salida. Todo lo que escribamos en salida será enviado por el servidor web al navegador:

```
salida.println("");
salida.println("<!DOCTYPE 4.01 Transitional//EN\">");
salida.println("<HTML>"); salida.println(" <HEAD>");
```

```
salida.println(" <TITLE>Sistema Francés</TITLE>");
salida.println(" </HEAD>");
```

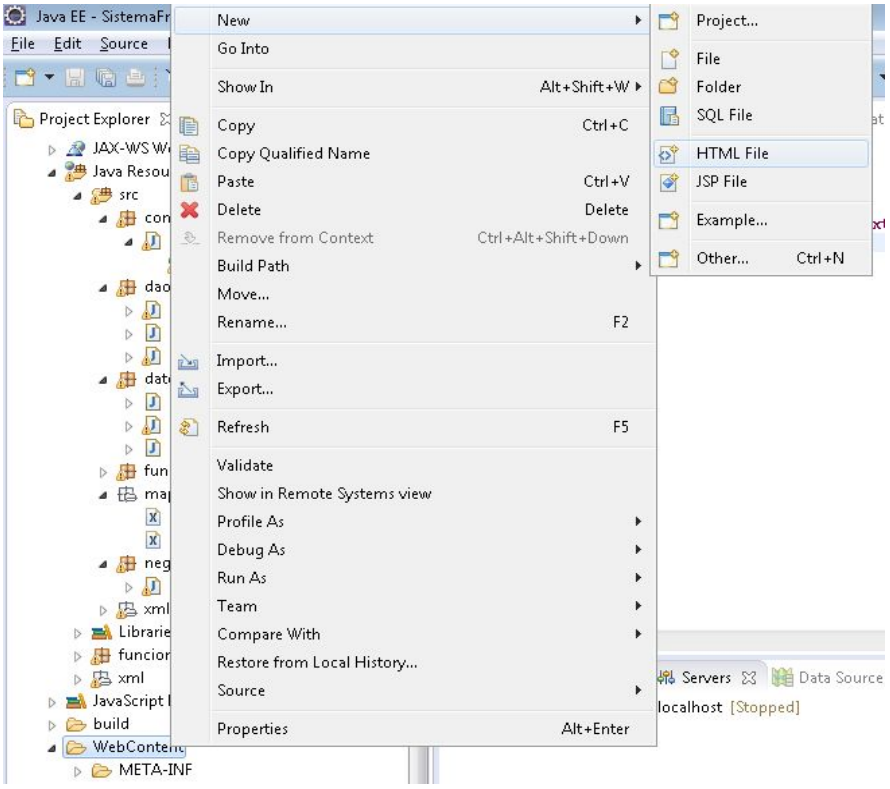
Con los datos del cliente obtenido, completamos el HTML de la respuesta:

```
salida.println("    Apellido: "+cliente.getApellido()+"<BR>");
salida.println("    Nombre  : "+cliente.getNombre()+"<BR>");
salida.println("    DNI     : "+cliente.getDocumento()+"<BR>");
salida.println("    ID      : "+cliente.getIdCliente()+"<BR>");
```

Finalmente, agregamos un link para volver a la página inicial.

En caso de que ocurra un error se activa la cláusula catch, que causa el envío de un mensaje de error 500 al navegador. Esta forma no es la más recomendada para avisar al usuario de que el DNI que acaba de ingresar no existe en la base de datos. Mejor sería ofrecer otra página HTML indicando las posibles causas del error y algunas opciones de recuperación. ***Ese ejercicio se deja para los estudiantes.***

Para que el controlador se active, es necesario que se lo invoque de algún modo. A tal fin crearemos una página HTML simple, con un formulario para el ingreso del DNI del cliente a buscar y con un botón para efectuar la búsqueda. Para ello crearemos un nuevo archivo HTML en la carpeta WebContent y lo llamaremos cliente.html:



Ingresamos el siguiente código HTML:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
```

```
"http://www.w3.org/TR/html4/loose.dtd">
<HTML>
  <HEAD>
    <META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <TITLE>Sistema Francés - Cliente </TITLE>
  </HEAD>
  <BODY>
    <FORM method="POST" action=" /SistemaFrances/MostrarCliente ">
      Búsqueda de clientes<BR><BR>
      <TABLE border="0">
        <TR>
          <TD>DNI:</TD>
          <TD><INPUT name="dni"></TD>
        </TR>
        <TR>
          <TD>
            <INPUT type="submit" value="Consultar">
          </TD>
        </TR>
      </TABLE>
    </FORM>
  </BODY>
</HTML>
```

Como podemos observar, el método POST invoca la URL /SistemaFrances/MostrarCliente. Nuestro servlet controlador deberá ser invocado cada vez que se haga una petición a esa URL. Para conseguirlo, editamos el archivo de configuración web.xml en la carpeta WEB-INF y lo dejamos de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" id="WebApp_ID" version="3.1">
  <display-name>SistemaFrances</display-name>
  <welcome-file-list>
    <welcome-file>index.html</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>cliente.html</welcome-file>
  </welcome-file-list>

  <servlet>
        <servlet-name>ControladorMostrarCliente</servlet-name>
        <servlet-class>controladores.ControladorMostrarCliente</servlet-class>
  </servlet>
  <servlet-mapping>
        <servlet-name>ControladorMostrarCliente</servlet-name>
        <url-pattern>/MostrarCliente</url-pattern>
  </servlet-mapping>
</web-app>
```

En el tag <welcome-file-list> informamos todos los posibles nombres de los archivos que debe devolver el servidor si la petición se hace sin indicar un

archivo determinado (localhost:8080/SistemaFrances). El primero que encuentra en la lista es el que se envía. Como nosotros sólo tenemos creado el archivo cliente.html, ése será el que se devuelva.

A continuación, informamos al servidor de los servlets que tenemos disponibles. Le asignamos un nombre e informamos la clase de la que serán instancia:

```
<servlet>
	<servlet-name>ControladorMostrarCliente</servlet-name>
	<servlet-class>controladores.ControladorMostrarCliente</servlet-class>
</servlet>
```

Finalmente, ingresamos un mapeo entre el nombre del servlet y la URL a la que responderá. En nuestro caso, es /MostrarCliente, que al agregarle el identificador del sitio quedará como /SistemaFrances/MostrarCliente.

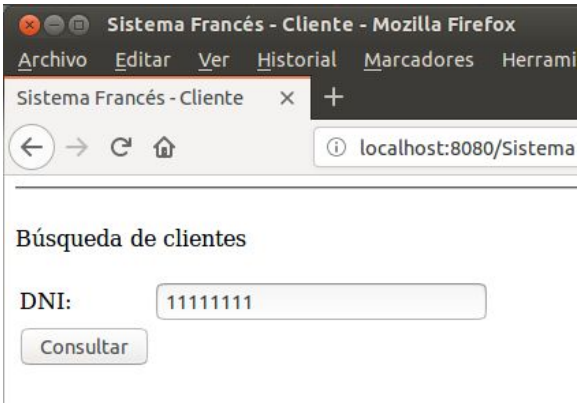
```
<servlet-mapping>
	<servlet-name>ControladorMostrarCliente</servlet-name>
	<url-pattern>/MostrarCliente</url-pattern>
</servlet-mapping>
```

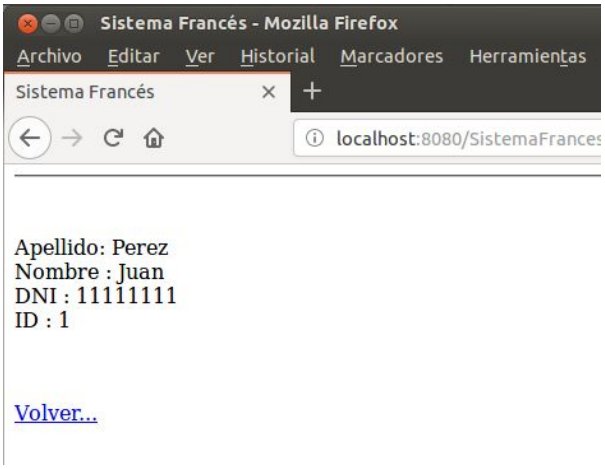
Recordemos la definición del método POST en cliente.html:

```
<FORM method="POST" action="/SistemaFrances/MostrarCliente">
```

Cuando el navegador realice esta petición POST al servidor, éste relacionará la URL recibida con el controlador asociado, lo instanciará e invocará a su método doPost, pasándole los objetos petición y respuesta correspondientes.

Para ejecutar la aplicación, botón derecho sobre el proyecto, Run as/Run on server. Una vez que el servidor haya cargado el proyecto y éste arranque, se abrirá una ventana de navegador apuntando a http://localhost:8080/SistemaFrances. Si la ventana no se abriera y el proyecto se hubiera inicializado correctamente, podemos abrir el navegador a mano e ingresar la dirección para acceder a la aplicación.





JSP (Java Server Pages)

Las vistas en nuestra implementación de MVC pueden, como ya hemos visto, implementarse mediante HTML generado en el controlador. Esto hace engorrosa la implementación del controlador y de la vista, ya que hay que expresarla como una serie de llamadas a métodos PrintWriter.out() que emitirá una línea de HTML por llamada.

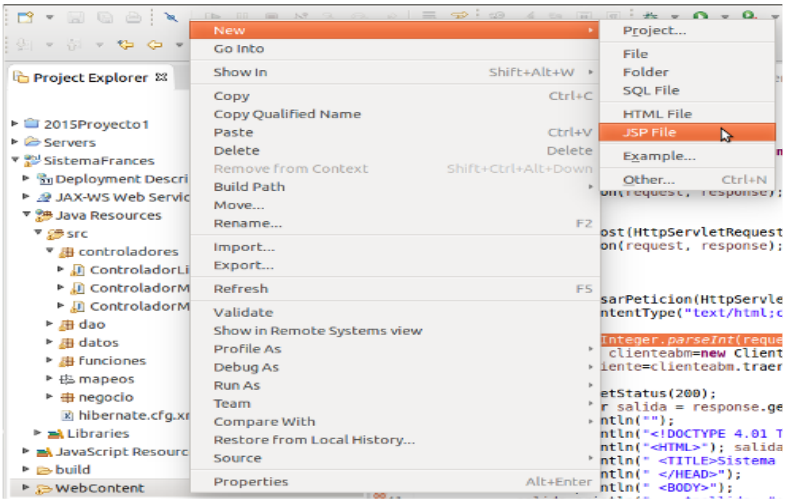
Otra forma sería tener un archivo HTML con tags especiales que el controlador pueda leer y reemplazarlos con contenido generado dinámicamente. Esto nos ahorraría gran parte de las llamadas a out() necesarias. En PHP, por ejemplo, se utilizan archivos HTML con tags especiales que nos permiten embeber código PHP. El servidor, al levantar el archivo, va ejecutando el código a medida que lo encuentra, generando contenido dinámico.

En las aplicaciones web dinámicas JavaEE, se adopta un enfoque parecido pero diferente: También se utiliza un archivo HTML con tags especiales que permiten embeber código Java con extensión JSP (Java Server Pages). A primera vista podría parecer similar a PHP, pero la diferencia radica en que nuestras JSP se transforman en servlets antes de invocarlas. Es decir: El servidor las transforma en algo similar al controlador que acabamos de escribir. El código HTML de la JSP se transforma en una serie de métodos out y junto con nuestro código java se incorpora al método service. La ventaja de éste método es que es más simple diseñar la vista desde el HTML con código Java embebido que desde Java con HTML embebido, y además, el servlet JSP tiene acceso a todas las clases de nuestro programa y a las del entorno en el que se está ejecutando.

Las JSP, como los servlets, tienen un ciclo de vida. Poseen constructor y métodos init, service y destroy. En general no deberemos hacer llamadas directas a los mismos, pero en caso de ser necesario, es posible.

Vamos a agregar una vista JSP al ejemplo que venimos desarrollando. Para ello deberemos agregar otro controlador (podríamos modificar el existente, pero lo perderíamos) que se encargue de la misma, los archivos JSP necesarios y deberemos modificar el descriptor de despliegue (web.xml) de

nuestro proyecto. Comenzamos agregando una nueva página JSP a WebContent:



La llamamos vistacliente.jsp y la editamos para que quede de la siguiente manera:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@page import="datos.Cliente"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Sistema Francés</title>
</head>
<BODY>
    <%@ include file="/cabecera.jsp" %>
    <% Cliente cliente=(Cliente)request.getAttribute("cliente"); %>
    <BR>
        Apellido: <%= cliente.getApellido() %><BR>
        Nombre  : <%= cliente.getNombre() %><BR>
        DNI     : <%= cliente.getDocumento() %><BR>
        ID      : <%= cliente.getIdCliente() %><BR>
        <BR>
    <A href="/SistemaFrances/index.jsp">Volver...</A>
</BODY>
</html>
```

Aquí podemos ver varios ejemplos de etiquetas JSP:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
```

Informa al servidor de que se trata de una página JSP, el contenido, el juego de carárcteres empleado y la codificación de la página.

```
<%@page import="datos.Cliente"%>
```

Nos permite importar una de nuestras clases para poder utilizarla en el código de la página. Es similar a la sentencia import de Java. Cuando el servidor transforme a la página JSP en un servlet, la declaración <%@page

import se transformará en el import correspondiente en el encabezado del servlet.

```
<%@ include file="/cabecera.jsp" %>
```

Esta declaración nos permite incluir el código de una JSP dentro de otra. El efecto práctico es que en lugar del include se insertará todo lo que contenga el archivo cabecera.jsp, que definiremos más adelante.

```
<% Cliente cliente=(Cliente)request.getAttribute("cliente"); %>
```

Este tag nos permite embeber código java en el texto de la página. Todo lo que aparezca dentro de <% ... %> se agregará como código al método service del servlet resultante. En este caso, declaramos una variable local cliente en el método service. De este modo estamos cargando una instancia de Cliente en la variable cliente, que obtuvimos del parámetro request (recuerden que esto se transforma en un servlet, y que el método service recibe una ServletRequest y una ServletResponse). Más adelante veremos cómo el controlador carga datos en la petición (request).
Si necesitáramos declarar métodos o variables de instancia, éstos se declaran con la etiqueta <%! ... %>

```
<%= cliente.getApellido() %>
```

Esta etiqueta nos permite insertar en el texto de la página el resultado de la expresión java que se encuentre dentro de ella; el valor de una variable o el resultado de la llamada a un método. En este caso insertará el apellido del cliente en la línea HTML.
A continuación, creamos dos archivos JSP más:

cabecera.jsp:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<TABLE border="0" width="100%">
  <TR>
    <TD align="left">
      <IMG src="/SistemaFrances/images/LogoUnla.jpg">
    </TD>
    <TD align="center">
      <H1>Orientación a Objetos II</H1>
    </TD>
  </TR>
</TABLE>
<HR>
<BR>
```

Como vemos, salvo la etiqueta de inicio es HTML simple.

index.jsp:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<HTML>
  <HEAD>
    <META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <TITLE>Sistema Francés - Cliente </TITLE>
  </HEAD>
  <BODY>
    <%@ include file="/cabecera.jsp" %>
      <FORM method="POST" action=" /SistemaFrances/MostrarClienteJSP ">
        Búsqueda de clientes<BR><BR>
        <TABLE border="0">
           <TR>
             <TD>DNI:</TD>
             <TD><INPUT name="dni"></TD>
           </TR>
           <TR>
            <TD>
              <INPUT type="submit" value="Consultar">
            </TD>
          </TR>
        </TABLE>
      </FORM>
   </BODY>
</HTML>
```

También sólo HTML, salvo la etiqueta de inclusión. Es necesario que sea un JSP porque HTML 4 no permite incluir archivos.

A continuación, agregamos un controlador, al que llamaremos ControladorMostrarClienteJSP.java, con el siguiente código:

```
package controladores;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import negocio.ClienteABM;
import datos.Cliente;

public class ControladorMostrarClienteJSP extends HttpServlet {

        protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
                procesarPeticion(request, response);
        }

        protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
                procesarPeticion(request, response);
        }
```

```
	private void procesarPeticion(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
		response.setContentType("text/html;charset=UTF-8");
		try {
			int dni = Integer.parseInt(request.getParameter("dni"));
			ClienteABM clienteabm=new ClienteABM();
			Cliente cliente=clienteabm.traerCliente(dni);
			request.setAttribute("cliente", cliente);
			request.getRequestDispatcher("/vistacliente.jsp").forward(request ,
response);
		}
		catch (Exception e) {
			response.sendError(500, "El DNI Ingresado no existe en la base de
datos.");
		}
	}
}
```

Lo único que cambió con respecto al controlador original es el método procesarPeticion. Podemos ver cómo en lugar de generar el HTML línea por línea, simplemente agrega nuestro cliente como un atributo al objeto request con la llamada request.setAttribute, para luego pasar el control a vistacliente.jsp.

Este pase de control se hace pidiendo un RequestDispatcher al objeto request. El término dispatcher (despachador) en general se refiere a aquello que encamina algo de algún modo. En nuestro caso, nuestra petición, que ya está siendo atendida por nuestro controlador, deberá seguir su camino hacia vistacliente.jsp. Esa redirección la realiza el RequestDispatcher al invocar su método forward(). La página JSP (recordemos que se trata de un servlet!) será invocada con las mismas petición y respuesta que recibimos, pero a la primera le agregamos un atributo más (cliente). Este atributo será el que se recuperará en la línea

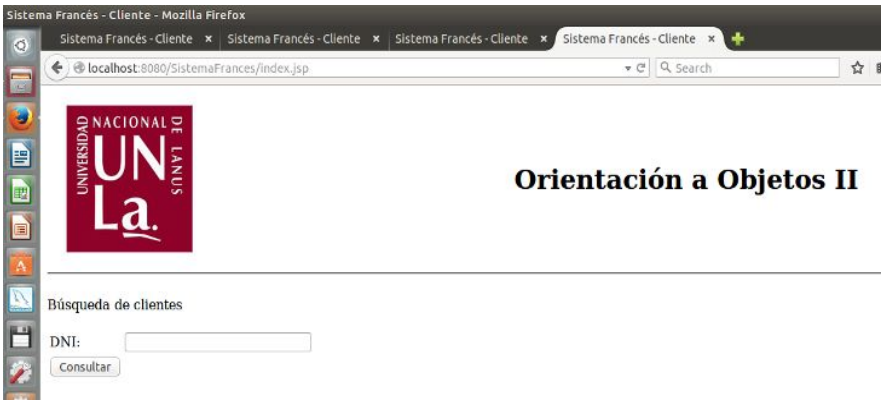
```
<% Cliente cliente=(Cliente) request.getAttribute("cliente"); %>
```

de vistacliente.jsp

Para que todo esto esto funcione, debemos informarle a nuestra aplicación web que agregamos un controlador más. Para ello, agregamos a web.xml:

```
<servlet>
	<servlet-name>ControladorMostrarClienteJSP</servlet-name>
	<servlet-class>controladores.ControladorMostrarClienteJSP</servlet-class>
</servlet>
<servlet-mapping>
	<servlet-name>ControladorMostrarClienteJSP</servlet-name>
	<url-pattern>/MostrarClienteJSP</url-pattern>
</servlet-mapping>
```

Una vez más, botón derecho sobre el proyecto, Run as/Run on server para correrlo.



Agregaremos más funcionalidad a nuestro proyecto: queremos que nos muestre una lista de todos los clientes en la base de datos. Además vamos a hacerlo de dos formas: Una, usando JSP como hasta ahora y otra utilizando JSTL (JSP Standard Tag Library). JSTL es una librería que agrega etiquetas al JSP para facilitar la tarea de programar páginas JSP. Veremos un ejemplo de utilización, sin profundizar demasiado en su uso. El sitio oficial de JSTL es https://jstl.java.net/ . A la vez, veremos como invocar dos vistas desde un mismo controlador dependiendo de un parámetro recibido en la petición. De este modo podremos mostrar los dos ejemplos sin necesidad de implementar dos controladores que serían básicamente iguales.

Primero vamos a modificar el archivo index.jsp agregándole la funcionalidad necesaria para pedir los listados. Entre las etiquetas </FORM> y </BODY> agregamos lo siguiente:

```
<BR><BR><BR><BR>
<FORM method="POST" action="/SistemaFrances/ListarClientes">
  <input type="hidden" name="jstl" value="0" />
  <TABLE border="0">
    <TR>
      <TD>
        <INPUT type="submit" value="Listar Clientes">
      </TD>
    </TR>
    </TABLE>
</FORM>
<BR><BR><BR><BR>
<FORM method="POST" action="/SistemaFrances/ListarClientes">
  <input type="hidden" name="jstl" value="1" />
    <TABLE border="0">
      <TR>
        <TD>
          <INPUT type="submit" value="Listar Clientes con JSTL">
        </TD>
      </TR>
    </TABLE>
</FORM>
<BR><BR><BR><BR>
```

Lo único que estamos haciendo es agregar dos forms que se enviarán al servidor con una petición POST al presionar el botón correspondiente. Estos forms contienen un parámetro en un inputbox escondido (<input type="hidden" name="jstl" value="1" />), que es el que le indica al controlador que versión de la vista tiene que invocar: con o sin JSTL.

Creamos un nuevo archivo JSP, lo llamamos vistalistacliente.jsp y lo completamos con el siguiente código:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@page import="datos.Cliente"%>
<%@page import="java.util.List"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>Sistema Francés</title>
    </head>
    <body>
      <%@ include file="/cabecera.jsp" %>
      <BR>
      Listado Sin utilizar JSTL
      <table border="1">
        <tr>
          <th>Cliente</th>
          <th>DNI</th>
        </tr>
        <% List<Cliente> clientes=(List)request.getAttribute("clientes");
          for(Cliente cliente:clientes){ %>
            <tr>
              <td><%= cliente.getApellido()+", "+cliente.getNombre() %></td>
              <td><%= cliente.getDocumento() %></td>
            </tr>
        <% } %>
      </table>
      <BR>
      <A href="/SistemaFrances/index.jsp">Volver...</A>
  </body>
</html>
```

Además de importar java.util.List, la principal diferencia que vemos en este JSP es el uso de iteración java para armar la tabla de clientes. Todo lo que haya entre las líneas

```
<% List<Cliente> clientes=(List)request.getAttribute("clientes");
  for(Cliente cliente:clientes){ %>
```

y

```
<% } %>
```

se repetirá en el HTML final. Iteramos de la misma manera que en java, con la salvedad de tener que utilizar las etiquetas JSP. Por supuesto que el

controlador deberá haber cargado en el objeto HttpServletRequest la lista de clientes como atributo.

Ahora crearemos la vista que utilizará JSTL. Para ello, deberemos agregar el archivo jstl-1.2.jar a nuestro proyecto. Lo bajamos del sitio de JSTL (estamos usando la versión 1.2) y lo copiamos en la carpeta Webcontent/WEB-INF/lib de nuestro proyecto. Luego agregamos la librería como en el caso de las de Hibernate.

Creamos entonces un nuevo archivo JSP, lo llamamos vistalistaclientejstl.jsp y lo completamos con el siguiente código:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>Sistema Francés</title>
  </head>
  <body>
    <%@ include file="/cabecera.jsp" %>
    <BR>
    <BR>
    Listado utilizando JSTL
    <table border="1">
      <tr>
        <th>Cliente</th>
        <th>DNI</th>
      </tr>
      <c:forEach items="${clientes}" var="cliente">
        <tr>
          <td>${cliente.apellido}, ${cliente.nombre}</td>
          <td>${cliente.documento}</td>
        </tr>
      </c:forEach>
    </table>
    <BR>
    <A href="/SistemaFrances/index.jsp">Volver...</A>
  </body>
</html>
```

La primera diferencia que notamos es la línea

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

que indica al servidor que esta página utiliza JSTL. Luego podemos observar la ausencia de código java, reemplazado por los tags foreach y ${} y la falta de inicialización de la variable clientes, tarea que realiza JSTL por nosotros. El código queda más compacto y legible de este modo.

Una nota de atención: en lugar de ${cliente.getApellido()} el lenguaje de expresiones de JSTL (Expression Language, EL) permite escribir ${cliente.apellido}. Lo que en realidad ocurre es que como tanto EL como las clases que escribimos nosotros adhieren a la convención JavaBeans, EL sabe que cuando pedimos cliente.apellido en realidad debe invocar un getter denominado getApellido(). Si tenemos una propiedad que no tenga un getter definido, no se podrá acceder de este modo.

Creamos un nuevo controlador, con nombre ControladorListaCliente y con el siguiente contenido:

```
package controladores;

import java.io.IOException;
import java.util.List;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import negocio.ClienteABM;
import datos.Cliente;

public class ControladorListaCliente extends HttpServlet {

	protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		procesarPeticion(request, response);
	}

	protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		procesarPeticion(request, response);
	}

	private void procesarPeticion(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
		int jstl = Integer.parseInt(request.getParameter("jstl"));
		try {
			ClienteABM clienteabm=new ClienteABM();
			List<Cliente> clientes=clienteabm.traerCliente();
			request.setAttribute("clientes", clientes);
			if (jstl==0)
				request.getRequestDispatcher("/vistalistacliente.jsp").forward(
							request, response);
			else
				request.getRequestDispatcher("/vistalistaclientejstl.jsp").forward(
							request , response);
		}
		catch (Exception e) {
			response.sendError(500, "Hubo un problema al traer el listado
					de clientes.");
		}
	}
}
```

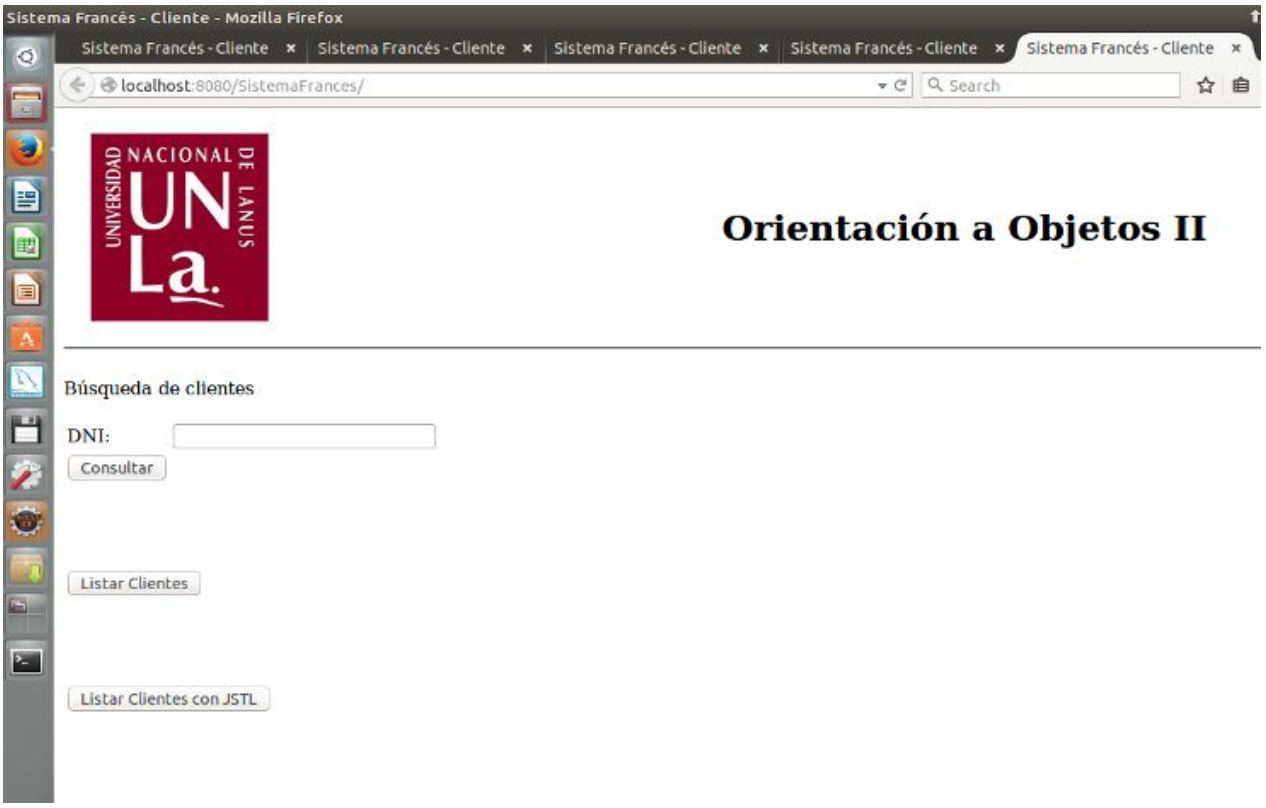
La estructura del controlador es similar a la del anterior, salvo que el atributo que agregamos a request es una lista de clientes en lugar de un cliente solo.

Traemos el parámetro jstl de request también y lo utilizamos para elegir la vista a mostrar.

Sólo nos falta agregar la configuración del servlets al archivo web.xml:

```
<servlet>
	<servlet-name>ControladorListaCliente</servlet-name>
	<servlet-class>controladores.ControladorListaCliente</servlet-class>
</servlet>
<servlet-mapping>
	<servlet-name>ControladorListaCliente</servlet-name>
	<url-pattern>/ListarClientes</url-pattern>
</servlet-mapping>
```

Una vez más, corremos en el servidor:

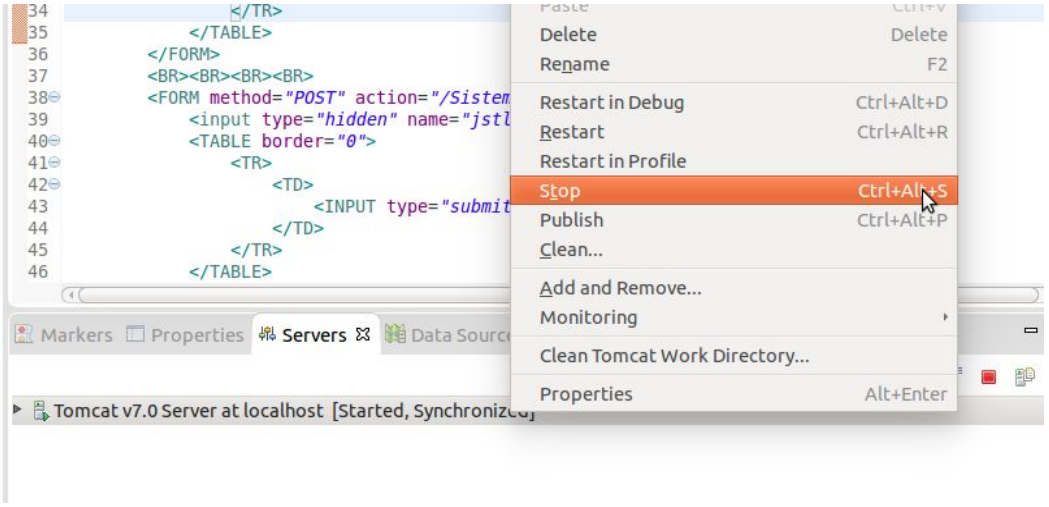


Algunas cosas a tener en cuenta:

Las librerías siempre deben ir en la carpeta WebContent/WEB-INF/lib. Todo lo que tenga que ir al servidor tiene que almacenarse en la carpeta WebContent. Recordemos que el programa se ejecuta en el servidor, así que si el servidor no tiene las librerías va a fallar con un mensaje de error poco explicativo.

Si vemos que modificamos el proyecto, y al correrlo nuestra modificaciones no están, conviene detener el servidor y volver a correr la aplicación. Para ello, en la pestaña servidores del panel inferior de eclipse,

damos click con el botón derecho sobre el servidor que tengamos corriendo y elegimos stop del menú contextual:



Para profundizar los conceptos de esta clase ver los capítulos 4 a 7 del libro “Programación Web en Java” que se encuentra en la bibliografía.