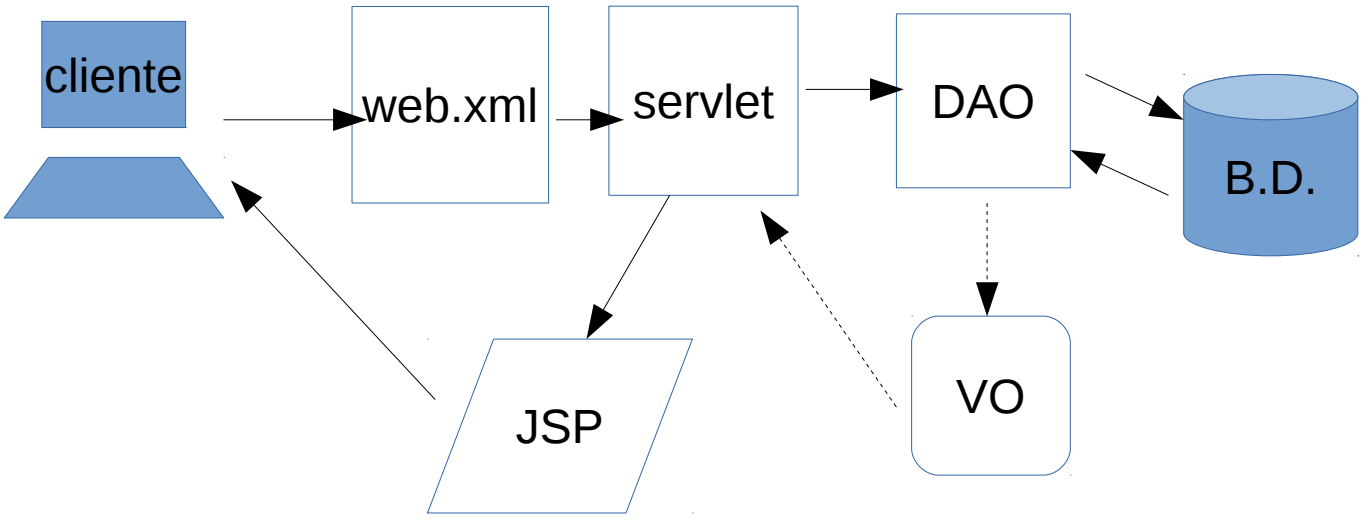
Patrón MVC

El patrón MVC (Model-View-Controller) es un patrón muy conocido y utilizado para el diseño de aplicación Web. Este patrón consiste en diferenciar claramente los componentes de nuestra aplicación que realizan esas tres funciones básicas:
- VIEW: Capa de visualización de la aplicación, o lo que es igual, el Interfaz de Usuario.
- MODEL: Capa de modelo de la aplicación. Incluye el modelo de datos y la capa de persistencia, es decir, la conexión con la base de datos.
- CONTROLLER: Es el corazón de la aplicación. Es el gestor de tráfico que decide cómo atender a una URL concreta, a quién enviarle la petición, qué debe hacer, qué lógica aplicar, y qué componente de la capa View debe responder. El controller lleva el control de la aplicación, y se conecta con las otras dos capas para resolver la funcionalidad de la aplicación.

Podemos representar el patrón MVC con el siguiente esquema:



1.- El cliente lanza una petición al servidor, llamando a una URL concreta.
2.- El controlador (a través de la definición que hay en web.xml), decide qué servlet debe contestar a esa petición.
3.- El servlet responsable ejecuta su lógica de aplicación, y cuando lo necesita, instancia los DAO adecuados para interactuar con la base de datos. Los elementos que intercambian servlet y DAO son, fundamentalmente, objetos VO.
4.- Una vez que el servlet ha ejecutado su lógica, y tiene en memoria los VO necesarios, decide a qué JSP (capa View) delega la confección de la página de respuesta (HTML) que va a enviar de nuevo al cliente, y ejecuta un *forward* a ese JSP, pasándole los VO necesarios a través del objeto *Request.*
5.- El JSP designado, que aparentemente es un documento HTML que incluye pequeñas partes dinámicas, pero que internamente es un Servlet especializado en generar HTML, generará la página de respuesta, y se la enviará al cliente, cerrando el círculo de la petición.

Veamos con un ejemplo cómo funciona esta maquinaria en una aplicación web JEE.

1.- El cliente lanza una petición, llamando por ejemplo a la siguiente URL:

http://localhost:8080/helloWorld/demoList

helloWorld es el nombre de nuestra aplicación, y *demoList* el nombre del recurso al que deseamos acceder.

2.- El fichero web.xml, núcleo del controlador, decide a qué servlet va a enviar la petición. Para ello, tendremos configurado el fichero de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd">
 <description>Hola Mundo</description>
 <display-name>helloWorld</display-name>
 <servlet>
  <display-name>Demo</display-name>
  <servlet-name>DemoController</servlet-name>
  <servlet-class>es.unizar.sisinf.controller.Demo</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>DemoController</servlet-name>
  <url-pattern>/demoList</url-pattern>
 </servlet-mapping>

</web-app>
```

Es decir, creamos un servlet, llamado DemoController, que va a ser gestionado por la clase *es.unizar.sisinf.controller.Demo*. No hará falta que instanciemos en ningún caso la clase Demo (d = new Demo()). Eso lo hará Tomcat por nosotros cuando corresponda.

Además, mapeamos la URI */demoList* hacia el servlet *DemoController*.

Con esta configuración, cuando recibimos la URL http://localhost:8080/helloWorld/demoList en nuestro servidor, automáticamente Tomcat instancia nuestro servlet, y llamará al procedimiento *doGet o doPost*, según sea la petición http recibida.

3.- Nuestro servlet Demo se hace cargo de la petición. Nuestro servlet tiene la siguiente forma:

```
package es.unizar.sisinf.controller;

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 es.unizar.sisinf.data.dao.DemoDAO;
import es.unizar.sisinf.data.vo.DemoVO;


public class Demo extends HttpServlet {

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

```
	/**
	 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
	 */
	protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
		// Llamada al metodo post
		doPost(request, response);
	}



	/**
	 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
	 */
	protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
		try {
			List<DemoVO> lista = new DemoDAO().findAll();


			System.out.println(lista.get(1).getName());

			request.setAttribute("listaDemo", lista);

			request.getRequestDispatcher("listaDemo.jsp").forward(request, response);

		}catch(Throwable theException) {

		}
	}


}
```

- doGet apunta a doPost, de manera que la respuesta será la misma independientemente del método http utilizado.
- doPost instancia el DAO DemoDAO, y ejecuta el método findAll(), que devuelve la lista de todos los registros de la tabla DEMO de nuestra base de datos, y los guarda en una lista.
- el servlet copia esa lista en un atributo llamado “listaDemo” del Request. El Request es un parámetro que reciben automáticamente todos los servlets desde Tomcat. Ese parámetro es un contenedor de información que tiene, entre otras cosas, todos los parámetros que hayamos enviado en nuestra URL, pero además permite “pegarle” más información por el camino. Lo que hace nuestro servlet es pegar al Request la lista de registros leídos en la base de datos, y pasarle ese Request al JSP de respuesta (no olvidemos que el JSP es, en definitiva, un servlet, y por tanto recibe también un Request).
- Finalmente, el servlet busca el JSP *listaDemo.jsp*, y hace un forward al mismo pasando el Request (ahora ya con la lista de registros) y el mismo response que recibió. Un *forward* no es más que una forma de decirle al JSP: ¡Hazte cargo tú de contestar al cliente, y pásale esta información que te doy! El servlet es el Jefe, y el JSP el mensajero que lleva la información al cliente.

4.- El JSP responde. El JSP debe generar un HTML que combine partes estáticas (una plantilla de diseño de la página de respuesta), junto con un contenido dinámico (los datos que ha recibido del servlet, y que debe enviar al cliente).

Los JSP pueden contener, en medio del código HTML, pequeños fragmentos de código java para ejecutar esas partes dinámicas. Sin embargo SIEMPRE ES PREFERIBLE construir las partes dinámicas utilizando tags de la librería de tags dinámicos de JSP. De esa manera, nuestro documento JSP será siempre XML compliant, y evitamos errores de código, al mezclar partes de interfaz de usuario (HTML) con partes de código dinámico (java).

Nuestro JSP podría ser así:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!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=UTF-8">
<title>Lista Demo</title>
</head>
<body>
        <table>
        <c:forEach var="demo" items="${listaDemo}">
            <tr><td>
                    <c:out value="${demo.id}" />
                </td><td>
                    <c:out value="${demo.name}" />
                </td></tr>
        </c:forEach>
        </table>

</body>
</html>
```

En nuestro JSP hemos incluido una librería de etiquetas estándar de JSP (jstl core), que luego utilizamos en expresiones tales como <c:forEach …

En este caso, el tag *forEach* buscará un atributo llamado *listaDemo*, (el que habíamos guardado en el Request), y realiza una iteración por todos sus elementos. No olvidemos que *listaDemo* era un objeto de tipo List, y por tanto, iterable. Con cada uno de los elementos leídos en la iteración (cada elemento de nuestro List era de tipo DemoVO) hacemos un *out* de su *id* y de su *name*, dentro de las casillas de una tabla.

Si juntamos todos los componentes, y los encapsulamos en un WAR, ya tenemos nuestra primera aplicación Web con conexión a una base de datos.