Práctica 5

SERVLETS Y PATRÓN MVC

1. Objetivo

El objetivo primordial de esta práctica es el aprendizaje del patrón MVC. Dicho aprendizaje lo vamos a efectuar de tal manera que, partiendo de los ejemplos de las prácticas anteriores, consigamos unos resultados similares pero adaptados a la arquitectura MVC.

Además, en esta práctica vamos a ir un paso más allá y vamos a definir y utilizar por primera vez descriptores de despliegue (archivo web.xml). También veremos por encima cómo desplegar (*deploy*) una aplicación en Tomcat utilizando medios automatizados y herramientas del propio servidor.

2. Introducción a los patrones

2.1. ¿Qué son los patrones?

Los patrones de software son soluciones reutilizables a los problemas que ocurren durante el desarrollo de un sistema de software o aplicación. Estos proporcionan un proceso consistente o diseño que uno o más desarrolladores pueden utilizar para alcanzar sus objetivos. También proporciona una arquitectura uniforme que permite una fácil expansión, mantenimiento y modificación de una aplicación.

Para ampliar información sobre este tema, puede acudirse al libro “Patrones de Diseño” de E. Gamma et al., Ed. Addison-Wesley, detallado en la bibliografiía de la asignatura.

Para ampliar información sobre los patrones que se suelen usar en aplicaciones J2EE :

http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/

2.2. Estructura de los patrones

Los patrones usualmente se describen con la siguiente información:

- Descripción del problema: Que permitirá el patrón y ejemplos de situaciones del mundo real.
- Consideraciones: Que aspectos fueron considerados para que tomara forma esta solución.
- Solución general: Una descripción básica de la solución en si misma.
- Consecuencias: Cuales son los pros y contras de utilizar esta solución.
- Patrones relacionados: Otros patrones con uso similar que deben ser considerados como alternativa.

2.3. Tipos de Patrones

Existen diferentes tipos de patrones. Dependiendo del nivel conceptual de desarrollo donde se apliquen, se distinguen (de más abstractos a más concretos): patrones de análisis, patrones arquitectónicos, patrones de diseño y patrones de implementación o *idioms*.. Al respecto puede encontrarse más información en:

http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html

Dependiendo del propósito funcional del patrón, se distinguen los siguientes tipos:

- Fundamental: construye bloques de otros patrones.
- Presentación: Estandariza la visualización de datos.
- De creación: Creación condicional de objetos.
- Integración: Comunicación con aplicaciones y sistemas y recursos externos.
- De particionamiento: Organización y separación de la lógica compleja, conceptos y actores en múltiples clases.
- Estructural: Separa presentación, estructuras de datos, lógica de negocio y procesamiento de eventos en bloques funcionales.
- De comportamiento: Coordina/Organiza el estado de los objetos.
- De concurrencia: Maneja el acceso concurrente de recursos.

El patrón Modelo-Vista-Controlador (MVC) es un ejemplo de patrón arquitectónico estructural.

3. Patrón MVC

3.1. Introducción

Las aplicaciones Web pueden desarrollarse utilizando cualquier arquitectura posible. La arquitectura del patrón Modelo-Vista-Controlador es un paradigma de programación bien conocido para el desarrollo de aplicaciones con interfaz gráfica (GUI). En esta práctica implementaremos una aplicación web utilizando el MVC.

3.2. En que consiste el MVC?

El principal objetivo de la arquitectura MVC es aislar tanto los datos de la aplicación como el estado (modelo) de la misma, del mecanismo utilizado para representar (vista) dicho estado, así como para modularizar esta vista y modelar la transición entre estados del modelo (controlador). Las aplicaciones MVC se dividen en tres grandes áreas funcionales:

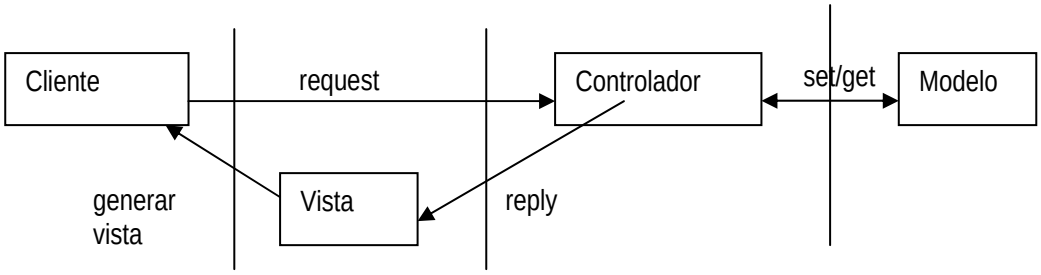
- Vista: la presentación de los datos
- Controlador: el que atenderá las peticiones y componentes para toma de decisiones de la aplicación
- Modelo: la lógica del negocio o servicio y los datos asociados con la aplicación

El propósito del MVC es aislar los cambios. Es una arquitectura preparada para los cambios, que desacopla datos y lógica de negocio de la lógica de presentación, permitiendo la actualización y desarrollo independiente de cada uno de los citados componentes.

El MVC consta de:

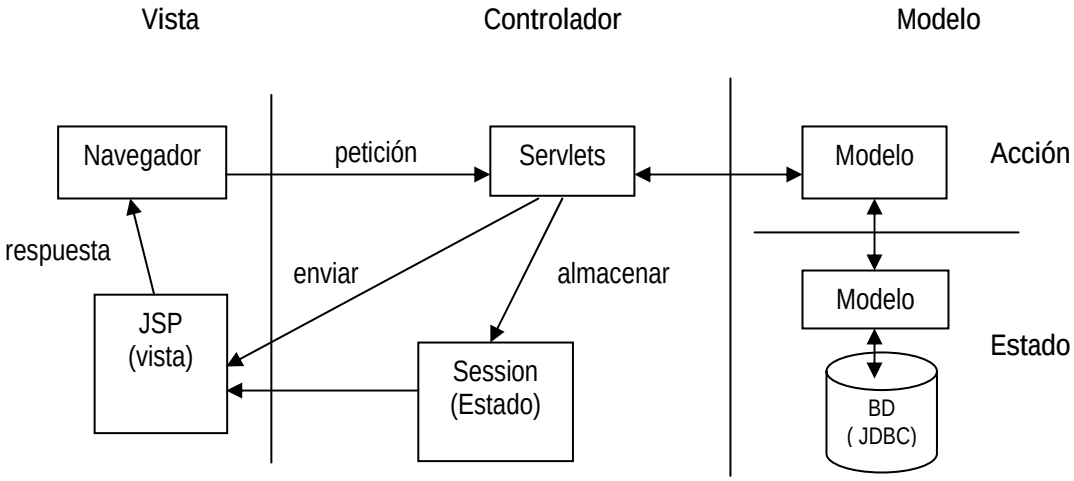
- Una o más vistas de datos
- Un modelo, el cual representa los datos y su comportamiento
- Un controlador que controla la transición entre el procesamiento de los datos y su visualización.

A continuación mostramos un esquema de este modelo:



4. MVC y J2EE

Las aplicaciones de MVC pueden ser implementadas con J2EE utilizando JSP para las vistas, servlets como controladores y, al no haber estudiado aún JavaBeans, que son idóneos para el modelo, utilizaremos de nuevo unos servlets para la capa de acción del modelo y otros servlets claramente diferenciados que manejarán JDBC para la capa de estado del modelo. Por ejemplo:



5. Ejemplo aplicando el MVC

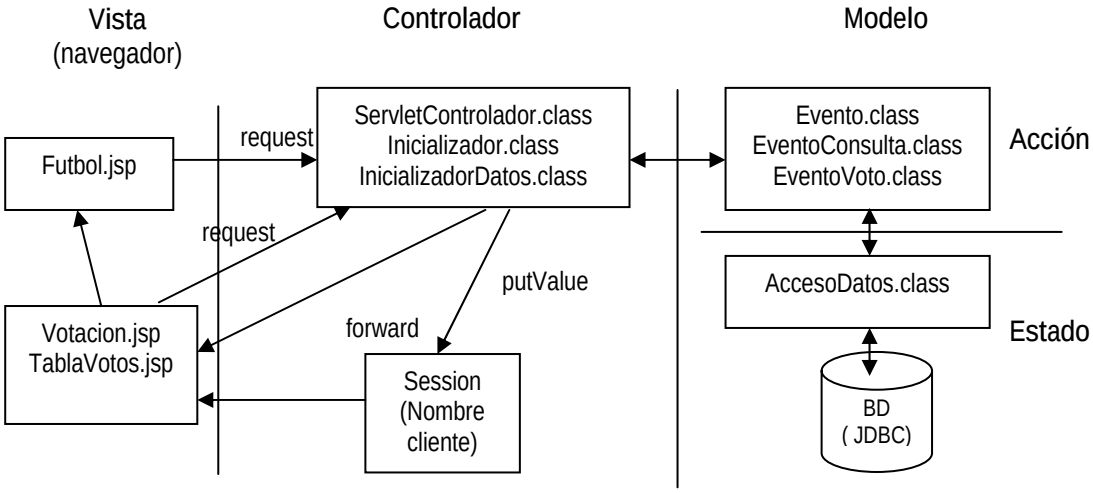
En el ejemplo que hemos venido desarrollando en las dos prácticas anteriores tenemos una página web Futbol.html que permite votar por un jugador. Luego esta información se procesa con un *servlet* intermedio Futbol.java que se encarga de actualizar la tabla de Jugadores de la base de datos BDJugadores (actualizando los votos de jugadores existentes o insertando nuevos

jugadores). Por último desde este servlet se da paso a la página TablaVotos.jsp para mostrarle al cliente como han quedado las estadísticas de votos.

En este ejemplo está aplicado en parte el MVC, puesto que se ha separado la vista (página html y página jsp) del controlador (servlet). Pero queda por separar la parte del modelo, que en este ejemplo consistiría en aislar todo el tema de acceso a datos que ahora mismo está mezclado en el servlet y la página jsp.

El nuevo modelo que se va a utilizar en esta práctica es prácticamente idéntico al visto en las clases teóricas: el controlador prácticamente es idéntico; evidentemente cambia la vista, pero sólo en cuanto a contenido, no el concepto; por último, el modelo se ha implementado también con las dos capas de un modelo: la capa de acción, más ligada al protocolo, HTTP en nuestro caso, y la capa de estado, la que realmente guarda el estado de la aplicación.

A continuación mostramos un esquema de cómo debería quedar el ejemplo aplicando el MVC:



6. Requisitos previos

6.1. Automatización del despliegue

La manera más sencilla de aprender a automatizar un despliegue en Tomcat es observar el ejemplo que se incluye, como una aplicación más, en la carpeta `webapps\tomcat-docs\appdev\sample` dentro de la instalación del servidor. Ahí encontraremos un archivo build.xml que debe ser empleado con la utilidad ant (http://jakarta.apache.org/ant/index.html). Este fichero espera una estructura de directorios de una manera predefinida y que nosotros podemos utilizar perfectamente para nuestras aplicaciones:

- Carpeta web: Ahí copiaremos toda la estructura de páginas (html, jsp, imágenes, etc.) así como una carpeta WEB-INF que contenga el descriptor de despliegue (web.xml).
- Carpeta src: Donde guardaremos nuestros ficheros fuente en java (servlets), con la estructura que determine el paquete que definamos.
- Carpeta build: Se crea automáticamente con todo el contenido y la estructura de nuestra aplicación. El contenido de esta carpeta se copiará íntegramente al lugar adecuado de Tomcat, produciendo el despliegue de la aplicación.

El fichero build.xml para la utilidad ant se maneja a través de lo que se denomina targets: es decir, diferentes objetivos que se construyen desde la línea de comandos. Los más útiles son los siguientes:

- *compile*: compila todo el código fuente.

- *install*: realiza el despliegue hacia Tomcat, copiando todos los ficheros útiles e indicándole al 'manager' de Tomcat la existencia de la aplicación.
- *remove*: hace la operación contraria.
- *clean*: borra todos los ficheros generados.

Otra forma de efectuar el despliegue es mediante dicho 'manager': una pequeña aplicación a la que se puede acceder desde http://localhost:8080/manager/html. Se puede instalar una aplicación desde un archivo .war. También es posible parar el servidor, copiar todos nuestros archivos a una carpeta dentro de `webapps` y volver a arrancar el servidor.

6.2. Descriptor de despliegue

En las prácticas anteriores nos hemos aprovechado de la posibilidad que ofrece Tomcat de utilizar el descriptor de despliegue por defecto que proporciona. Pero en el momento que queramos hacer un uso mayor del mismo, como por ejemplo la inclusión de parámetros globales o para un servlet, dicha utilización no es deseable, siendo mejor solución la definición de un descriptor propio para nuestra aplicación.

El descriptor que vamos a utilizar nosotros es el siguiente:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>

  <display-name>Práctica 5: Servlets y patrón MVC</display-name>

  <description>
    Práctica que combina servlets y JSPs dentro de una arquitectura MVC.
  </description>

  <context-param>
      <param-name>basedatos</param-name>
      <param-value>jdbc:mysql://localhost/BDJugadores</param-value>
    </context-param>

    <context-param>
      <param-name>driver</param-name>
      <param-value>com.mysql.jdbc.Driver</param-value>
  </context-param>

  <context-param>
      <param-name>user</param-name>
      <param-value>root</param-value>
  </context-param>

  <context-param>
      <param-name>password</param-name>
      <param-value>admin</param-value>
  </context-param>

  <servlet>

    <servlet-name>ServletFutbol</servlet-name>
    <servlet-class>mvc.ServletControlador</servlet-class>

    <init-param>
      <param-name>inicializador</param-name>
      <param-value>mvc.init.InicializadorDatos</param-value>
    </init-param>

    <init-param>
      <param-name>evento.consulta</param-name>
      <param-value>mvc.event.EventoConsulta</param-value>
    </init-param>

    <init-param>
```

```
      <param-name>evento.voto</param-name>
      <param-value>mvc.event.EventoVoto</param-value>
    </init-param>

  </servlet>

  <servlet-mapping>
    <servlet-name>ServletFutbol</servlet-name>
    <url-pattern>/futbol</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
     <welcome-file>Futbol.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

Como se ve, además de algunas cabeceras (display-name, description), podemos encontrar las siguientes entradas clave:

- *context-param*: son variables para el contexto. El valor de dichas variables se podrá obtener desde un servlet a través de la clase ServletContext, o desde una página JSP mediante el eso del objeto implícito application.
- *servlet*: define un servlet mediante una pareja de etiquetas: servlet-name es el nombre que nosotros le damos, mientras que servlet-class es la clase (o el paquete más la clase) que lo contiene. Además se pueden especificar parámetros a los que puede acceder el servlet a través de ServletConfig.
- *servlet-mapping*: indica la ruta (url) para acceder a servlet definido anteriormente.
- *welcome-file-list*: es la página por defecto de nuestra aplicación.

6.3. Acceso a datos

Para poder desarrollar esta práctica es necesario tener disponible los ejemplos de la práctica anterior y la base de datos en MySQL:

BDJugadores

Tabla Jugadores

Campo	Tipo
Nombre	Varchar (50)
Votos	Integer

Tabla Registro

Campo	Tipo
Nombre	Varchar (50)
Correo	Varchar (30)
Visitas	Integer

7. Implementación

7.1. Modelo

Para aplicar el MVC a nuestro ejemplo tendremos que desarrollar una serie de módulos independientes que se encarguen, en dos capas (acción y estado) del acceso a los datos. De esta manera estaremos separando el modelo y la lógica de negocio del controlador y la vista.

Para la capa de estado, podemos utilizar una clase que se encargue de los accesos a cada tabla de la base de datos. En nuestro caso sólo necesitamos acceder a la tabla de jugadores:

```java
package mvc.modelo;

import java.sql.*;

public class Jugadores
{
    private Connection con;
    private Statement set;
    private ResultSet rs;

    public Jugadores( Connection con)   { this.con = con; }

    public boolean existeJugador(String nombre) throws Exception
    {
        boolean existe = false;
        String cad;

        set = con.createStatement();
        rs = set.executeQuery("SELECT * FROM Jugadores");
        while (rs.next())
        {
            cad = rs.getString("Nombre");
            cad = cad.trim();
            if (cad.compareTo(nombre.trim())==0)
            existe = true;
        }

        rs.close();
        set.close();

        return(existe);
    }

    public void actualizarJugador(String nombre) throws Exception
    {
        set = con.createStatement();
        set.executeUpdate( "UPDATE Jugadores SET votos=votos+1 WHERE nombre "
                           + " LIKE '%" + nombre + "%'");
        rs.close();
        set.close();
    }

    public void insertarJugador(String nombre) throws Exception
    {
        set = con.createStatement();
        set.executeUpdate("INSERT INTO Jugadores "
                           + " (nombre,votos) VALUES ('" + nombre + "',1)");
        rs.close();
        set.close();
    }

    public void consultarJugadores() throws Exception
    {
        set = con.createStatement();
        rs = set.executeQuery("SELECT * FROM Jugadores");
    }

    public boolean siguiente() throws Exception
    {
        return (rs.next());
    }

    public String getCampo(String nombre) throws Exception
    {
        return (rs.getString(nombre));
    }

    public void cerrarConsulta()
```

```
    {
        try
        {
            rs.close();
            set.close();
        }
        catch(Exception e) { }
    }

    private void mostrar(String msg)
    {
        System.out.println(msg);
    }
}
```

En cuanto a la capa de acción, como hemos visto en la teoría, podemos implementar una clase para cada tipo de acción que el controlador ejerza sobre el modelo. En nuestra práctica vamos a manejar dos acciones: votar y consultar. A estas acciones las vamos a denominar eventos, con lo que construiremos una clase para cada una. Además, definimos una interfaz para las dos:

```
package mvc;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public interface Evento
{
    // Variables obtenidas del Descriptor de Despliegue (DD)
    public String NOM_EVENTO = "evento";
    public String procesar(ServletContext ctx, HttpServletRequest req)
                            throws IOException, ServletException;
}
```

El método procesar es la implementación del mensaje que envía el controlador. La siguiente clase es para los votos:

```
package mvc.event;

import javax.servlet.*;
import javax.servlet.http.*;
import mvc.*;
import mvc.init.*;
import mvc.modelo.*;

public class EventoVoto implements Evento
{
    // Variables obtenidas del Descriptor de Despliegue (DD)
    public static final String NOMBRE = "evento.voto";

    private static final String JSP_DESTINO = "TablaVotos.jsp";
    private static final String JSP_ERROR = "ErrorVotar.jsp";

    public EventoVoto() {} //Constructor vacío para instanciación dinámica.

    // Método sincronzado para evitar problemas de concurrencia
    // al comprobar el stock
    public synchronized String procesar(ServletContext ctx,
                                        HttpServletRequest req)
    {
        mostrar( "En " + this.toString() + " procesando evento");

        // Obtener el componente de estado del modelo
        Jugadores jug =
            (Jugadores)ctx.getAttribute(InicializadorDatos.NOM_JUGADORES);

        // Obtener la sesion
        HttpSession s = req.getSession(true);

        // Guardar el nombre del cliente en la sesión
        // para poderlo utilizar en el siguiente servlet
```

```
        String nombreP = (String)req.getParameter("txtNombre");
        s.putValue("nombreCliente", nombreP);

        String nombre=(String)req.getParameter("R1");

        boolean existe = false;
        try
        {
            if (nombre.equals("Otros"))
                nombre=(String)req.getParameter("txtOtros");

            if(nombre.equals("")) throw new Exception();

            if(jug.existeJugador(nombre)) jug.actualizarJugador(nombre);
            else jug.insertarJugador(nombre);

            mostrar("Se actualizó o insertó jugador " + nombre);
            return JSP_DESTINO;
        }
        catch( Exception e)
        {
            mostrar("Error al actualizar o insertar jugador " + nombre);
            jug.cerrarConsulta();
            return JSP_ERROR;
        }
    }

    private void mostrar(String msg)
    {
        System.out.println(msg);
    }
}
```

Como vemos, accede a la capa de estado a través de la clase Jugadores. La respuesta del mensaje es simplemente el nombre de la página JSP que el controlador debe utilizar para mostrar la información.

Esta otra clase es para la consulta:

```
package mvc.event;

import javax.servlet.*;
import javax.servlet.http.*;
import mvc.*;

public class EventoConsulta implements Evento
{
    // Variables obtenidas del Descriptor de Despliegue (DD)
    public static final String NOMBRE = "evento.consulta";
    private final static String JSP_DESTINO = "Votaciones.jsp";

    public EventoConsulta() {} //Constructor vacío para instanciación dinámica.

    public String procesar(ServletContext ctx, HttpServletRequest req)
    {
        mostrar( "En " + this.toString() + " procesando evento");
        return JSP_DESTINO;
    }

    private void mostrar(String msg)
    {
        System.out.println(msg);
    }
}
```

De nuevo se devuelve una página JSP que el controlador mostrará con la información.

7.2. Vista

Utilizaremos cuatro páginas JSP para la vista: la página inicial, dos páginas que muestran información y una última con un mensaje de error.

La página Futbol.jsp permite que un cliente introduzca su nombre y correo electrónico y pueda emitir su voto por algún jugador de los que se visualizan. Si no le conviene ninguno, podrá añadir un nuevo jugador. Esta página enviará su petición al servlet *ServletFutbol* tal como este está definido en el descriptor de despliegue, para éste envíe los mensajes oportunos al modelo. La página mantiene el código:

```
<%-- Futbol.jsp --%>

<%@ page contentType="text/html;charset=ISO-8859-1" %>
<%@ page import="mvc.*, mvc.event.*"%>

<html>
  <head>
  <title>Estadísticas de Futbol</title>
  </head>

  <body>

  <center><H1>Estadísticas de Jugadores de Futbol</H1></center>

  <p align="center"><font color="#002424" size="7">
    <u>VOTE POR EL MEJOR JUGADOR</u></font></p>
  <p align="center"><font color="#002424" size="7">
    <u>DE FUTBOL DE 2005</u></font></p>

  <form action="futbol" method="POST" left>

    <p align="left">Nombre del Visitante: <input type="text" size="20"
      name="txtNombre">
    eMail: <input type="text" size="20" name="txtMail"></p>
    <p align="left"><input type="radio" name="R1" value="Casillas">Casillas</p>
    <p align="left"><input type="radio" name="R1" value="Etoo">Etoo</p>
    <p>...</p>
    <p align="left"><input type="radio" name="R1" value="Otros">Otros <input
      type="text" size="20" name="txtOtros"></p>
    <p align="left"><input type="submit" name="B1" value="Votar"> <input
      type="reset" name="B2" value="Reset"></p>

    <input type="hidden" name="<%= Evento.NOM_EVENTO %>"
      value="<%= EventoVoto.NOMBRE %>" />

  </form>
  </body>
</html>
```

Como vemos, la página envía una petición al servlet controlador, utilizando un campo oculto (*hidden*) del formulario para indicarle al controlador el tipo de acción que se requiere del modelo.

La página Votacion.jsp muestra al usuario un mensaje de aceptación del voto y le ofrece la posibilidad de ver la tabla con todas las votaciones. Al igual que en el caso anterior, la página envía la petición al controlador utilizando un campo oculto.

```
<%-- Votacion.jsp --%>

<%@ page contentType="text/html;charset=ISO-8859-1" %>
<%@ page import="mvc.*, mvc.event.*"%>

<html>
  <head><title>Estadísticas de Futbol</title></head>

  <body>
    <h2>Voto completado</h2>

    <form action="futbol" method="POST">
      <input type="submit" value="Ver estadísticas" />
```

```
      <input type="hidden" name="<%= Evento.NOM_EVENTO %>"
                  value="<%= EventoConsulta.NOMBRE %>" />
    </form>

    <h3>Muchas gracias <%= session.getValue("nombreCliente") %> por su
      visita</h3>

    <a href="<%= request.getContextPath() %>">Inicio</a>
   </body>

</html>
```

La página TablaVotos.jsp muestra el contenido de toda la tabla de los jugadores con sus votos:

```
<%-- TablaVotos.jsp --%>
<%@ page import="mvc.*, mvc.event.*, mvc.modelo.*, mvc.init.*"%>
<html>
  <head><title>Estadísticas de Futbol</title></head>

  <body><font size=10>
    <h1>Total de votaciones</h1>
    <table border=1>
      <tr><td><b>Jugador</b></td><td><b>Votos</b></td></tr>
      <%
        Jugadores jug = (Jugadores)application.getAttribute(
                            InicializadorDatos.NOM_JUGADORES);

        jug.consultarJugadores();
        while (jug.siguiente())
        {
          String nombre = jug.getCampo("Nombre");
          String votos = jug.getCampo("Votos");
          out.println("<tr><td>"+ nombre + "</td><td>" + votos + "</td></tr>");
        }

        jug.cerrarConsulta();
      %>
    </table>
    </font>
    <br>

    <a href="<%= request.getContextPath() %>">Inicio</a>
  </body>
</html>
```

Por último, la página ErrorVotar.jsp aparecerá si ocurre un error en la votación, como por ejemplo intentar añadir un jugador sin nombre.

```
<%-- ErrorVotar.jsp --%>

<%@ page contentType="text/html;charset=ISO-8859-1"
  import="mvc.modelo.*, mvc.event.*, mvc.init.*"%>

<html>
  <head><title>Estadísticas de Futbol</title></head>

  <body>
    <h2>Error en la votación</h2>

    Puede intentar votar otra vez.<br>
    <a href="<%= request.getContextPath() %>"> Inicio</a>
  </body>

</html>
```

7.3. Controlador

A continuación mostramos el código del servlet controlador definido como *ServletFutbol* en el descriptor de despliegue e implementado por la clase ServletControlador.java. El servlet atenderá las peticiones de Futbol.jsp y Votacion.jsp, e acudirá a las clases que definen el modelo para la actualización y consulta de los datos de los jugadores.

Además, hemos utilizado una interfaz, Inicializador.java, y una clase InicializadorDatos.java para abstraer al propio servlet del manejo de los datos. Se puede comprobar que todo el contenido del paquete (*package*) `mvc`, incluida la clase ServletControlador.java, son de propósito general, pudiendo servir para cualquier otra aplicación. La concreción en este caso se hace a través de los parámetros definidos en el descriptor de despliegue: *inicializador*, *evento.consulta* y *evento.voto*.

Así pues, el servlet es el siguiente:

```
package mvc;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletControlador extends HttpServlet
{
    // Variables obtenidas del Descriptor de Despliegue (DD)
    private static final String INICIALIZADOR = "inicializador";
    private static final String PREFIJO_EVENTO = "evento.";

    // Otras variables
    private static final String NOM_TABLA_EVENTOS = "tablaEventos";

    // Inicialización del servlet
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);

        // Util para no perderse en los logs
        mostrar("");
        mostrar("********** Comienza práctica 5 **********");

        try
        {
            // Recuperar la clase inicializadora de la aplicación (ver DD).
            String inicializador = config.getInitParameter(INICIALIZADOR);
            Inicializador ini = (Inicializador)
                Class.forName(inicializador).newInstance();
            ini.init(config);

            // Recuperar las clases de los eventos
            Map eventos = new HashMap();
            Enumeration e = config.getInitParameterNames();

            // Recorrer los parámetros de inicio buscando eventos
            while (e.hasMoreElements())
            {
                String nombre = (String)e.nextElement();
                if (nombre.startsWith(PREFIJO_EVENTO))
                {   //Es un evento
                    String clase = config.getInitParameter(nombre);
                    //Clase que maneja el evento
                    Evento evento = (Evento)Class.forName(clase).newInstance();
                    eventos.put(nombre, evento);
                    mostrar("Clase " + clase +
                        " registrada para eventos de tipo " + nombre);
                }
            }
```

```
            //Guardar la tabla de eventos en el contexto
            config.getServletContext().setAttribute(
                NOM_TABLA_EVENTOS, eventos);
        }
        catch (Exception ex)
        {
            mostrar(ex.getMessage());
            ex.printStackTrace();
            throw new UnavailableException(ex.getMessage());
        }
    }

    public void doPost(HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException
    {
        // Busca en la estructura HTTP el parámetro y el valor que se lanzan
        // desde el formulario
        String nomEvento = req.getParameter(Evento.NOM_EVENTO);

        // Obtiene la colección de eventos definidos
        Map eventos = (Map)getServletContext().getAttribute(NOM_TABLA_EVENTOS);

        if (!eventos.containsKey(nomEvento))
        {
            //Si no se encuentra el evento se lanza excepción
            String msg = "Evento no encontrado " + nomEvento;
            mostrar(msg);
            throw new ServletException(msg);
        }
        else
        {
            // Si se encuentra el evento se procesa
            mostrar("Procesando evento " + nomEvento);

            // Se recupera la clase controladora
            Evento evento = (Evento)eventos.get(nomEvento);

            // Se delega el evento del proceso
            String path = evento.procesar(getServletContext(), req);
            mostrar("Evento procesado, redirigiendo a " + path);

            // Se redirige la petición
            req.getRequestDispatcher(path).forward(req, res);
            mostrar("Evento " + nomEvento + " procesado");
        }
    }

    private void mostrar(String msg)
    {
        System.out.println(msg);
        //log(msg);
    }
}
```

Como decíamos, se puede observar que los parámetros se leen del contexto mediante llamadas a la función `config.getInitParameter()`. Además, se va a crear una colección de eventos para cada tipo conocido.

Por otro lado, las clases que manejan la inicialización son las siguientes:

```
package mvc;
import javax.servlet.*;

public interface Inicializador
{
    public void init(ServletConfig cfg) throws ServletException;
}
```

para la interfaz, mientras que la clase que realiza todo el trabajo es la siguiente:

```
import java.sql.*;

import javax.servlet.*;
import mvc.*;
import mvc.modelo.*;

public class InicializadorDatos implements Inicializador
{
    // Nombre para el objeto Jugadores que se guardará en el contexto
    public static final String NOM_JUGADORES = "jugadores";
    // ... Añadir para más objetos

    // Variables obtenidas del Descriptor de Despliegue (DD)
    private static final String BASE_DATOS = "basedatos";
    private static final String DRIVER = "driver";
    private static final String USER = "user";
    private static final String PASSWORD = "password";

    //Constructor vacío para instanciación dinámica.
    public InicializadorDatos() {}

    public void init(ServletConfig cfg) throws ServletException
    {
        ServletContext ctx = cfg.getServletContext();
        String basedatos = ctx.getInitParameter(BASE_DATOS);
        String driver = ctx.getInitParameter(DRIVER);
        String user = ctx.getInitParameter(USER);
        String password = ctx.getInitParameter(PASSWORD);

        mostrar( "InicializadorDatos lee basedatos " + basedatos);
        mostrar( "InicializadorDatos lee driver " + driver);

        try
        {
            // Conectarse a la base de datos
            Class.forName(driver).newInstance();
            Connection con = DriverManager.getConnection(basedatos, user,
password);
            mostrar("Se ha conectado a " + basedatos);

            // Crear el objeto Jugadores del modelo
            Jugadores jug = new Jugadores(con);
            ctx.setAttribute(NOM_JUGADORES, jug);

            // Crear otros objetos ...

        }
        catch(Exception e)
        {
            mostrar( "No se ha conectado a " + basedatos);
            throw new ServletException(e.getMessage());
        }
    }

    private void mostrar(String msg)
    {
        System.out.println(msg);
    }
}
```

Podemos comprobar cómo se inician aquí las posibles fuentes de los datos (bases de datos en este ejemplo) y cómo se guarda en el contexto la conexión a la información de los jugadores para que pueda ser utilizada desde otras clases o desde páginas JSP.

8. Ampliación de la práctica

Para culminar la práctica se deben desarrollar por parte de los estudiantes las siguientes tareas:

- Ampliar la aplicación para que permita guardar los datos de los clientes que visitan la misma, guardando para cada uno su nombre, correo y las veces que ha votado, almacenando estos datos en la tabla *Registro*. Luego, además de darle las gracias al cliente y de ofrecerle la posibilidad de ver las estadísticas, se le debe informar también de cuántas veces ha votado.
- Todo lo anterior debe implementarse intentando cumplir al máximo con el patrón MVC.