

- *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 uso 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:

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.getParameter()`. 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.