

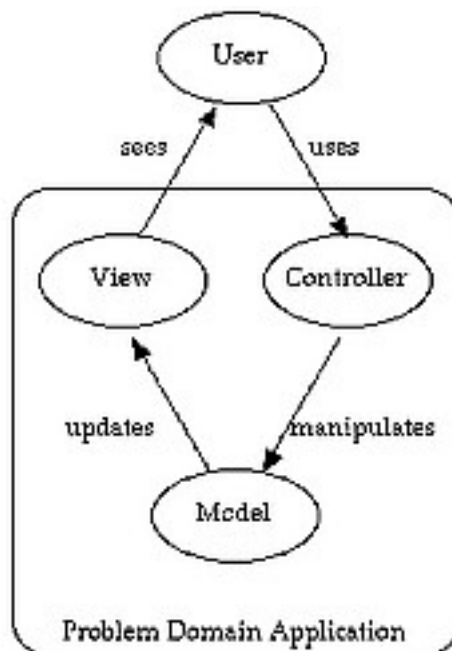
El patrón Command, base de MVC

En el siguiente documento se presenta una variante del patrón Command muy utilizada en la implementación de frameworks Web que siguen el patrón arquitectónico Model-View-Controller (MVC).

Caso práctico

Introducción

En aplicaciones web es frecuente seguir un diseño basado en el patrón arquitectónico Model-View-Controller. La siguiente figura muestra como se organizan cada uno de los componentes que conforman este patrón:



Nota: 'User' representa el navegador web del usuario, el cual habitualmente se comunica con un servidor a través de la red -intranet o Internet- mediante HTTP.

Veamos en qué consiste cada componente:

- El *Model* forma parte de la capa negocio, por lo que es un componente troncal en la aplicación. Contiene las clases que modelan la lógica del negocio o problema que da sentido a la existencia del programa. Por ejemplo, clases que calculan primas pólizas de seguros, clases que permiten llevar las ventas de

una empresa, etc. El Model se suele implementar mediante clases Java “normales” (POJO's) o mediante EJB's.

- La *View* forma parte de la capa de presentación. Tiene una responsabilidad muy delimitada: servir de interfaz gráfica para el usuario, tanto para recabar datos (formularios) como para presentarle resultados. En el caso de Java, es habitual utilizar HTML y JSP para crear las vistas. HTML cuando los contenidos no tienen que ser dinámicos (generados al vuelo) y JSP cuando sí deben serlo.
- El *Controller* también forma parte de la capa de presentación. Este componente, en esencia, recibe peticiones desde el navegador, las interpreta, invoca a los componentes que contienen la lógica de negocio asociada a cada petición, obtiene el resultado y selecciona una nueva vista a partir de la cual generará el código estático que enviará como respuesta al navegador. Se suele utilizar un Servlet como Front Controller.

Dado la gran cantidad de responsabilidades que tiene un Controller, es fundamental establecer un diseño que permita que el código del Servlet se mantenga simple, donde todos sus métodos presenten un nivel de abstracción similar, sin implementar los detalles de cada una de estas responsabilidades.

Para llevar esto a cabo es preciso que el Servlet delegue parte del trabajo en otras clases colaboradoras. En caso de no hacerlo así, terminaríamos teniendo un Servlet “hinchado”, con un código difícil de entender y en el que cualquier modificación o ampliación implicaría un duro trabajo de mantenimiento y poner en riesgo la aplicación.

De esta necesidad de separar conceptos surgieron frameworks como Struts, Java Server Faces, Spring MVC y tantos más. Para la empresa en la que trabajamos es estratégico no delegar en ningún producto de terceros, por lo que nos piden que creamos un framework web MVC sobre el que se puedan crear tanto aplicaciones de gestión como tiendas *on-line*.

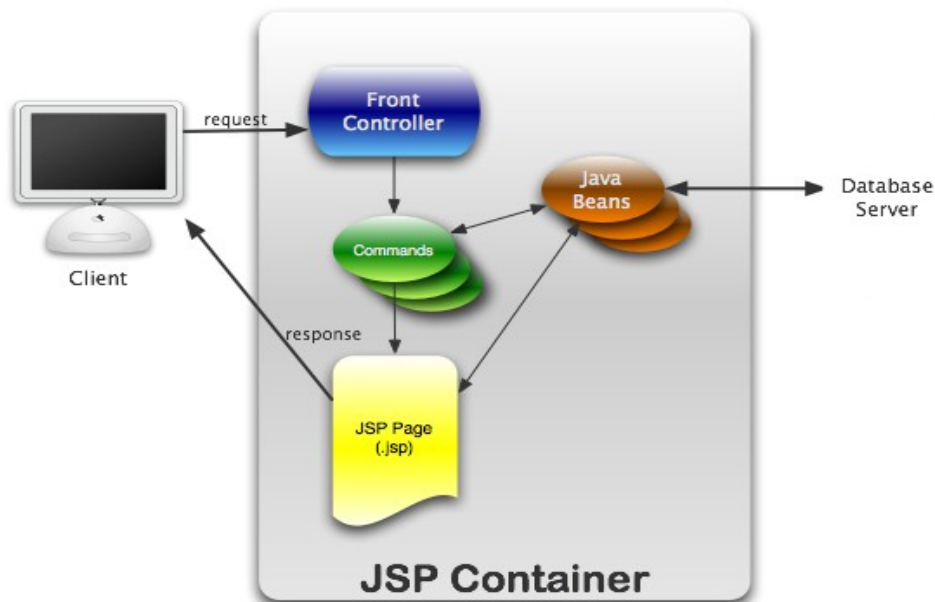
El patrón Command

Después de pensar con detenimiento llegamos a la conclusión de que una de las responsabilidades que más “hinchán” el código del Front Controller es el análisis y la correspondiente implementación de cada petición enviada por navegador. Hacer todo esto en el Servlet es una mala práctica en toda regla: dificultad de mantenimiento,

código difícil de leer, propenso a errores...y desde luego, una violación del principio de diseño abierto/cerrado (abierto para extensión, cerrado para modificación).

Por tanto, este es un caso en el que patrón Command nos puede ayuda sobremanera. La idea es que el usuario de nuestro framework creará clases comando en las que implementará el comportamiento específico para cada operación de sistema o caso de uso que quiera hacer accesible desde el navegador. Por ejemplo, si desde una página web se solicita un listado de los artículos de un catálogo, una clase comando determinada tendrá que ser instanciada y ejecutada. La ejecución -método execute() o similar- consistirá en recuperar de la capa de negocio la lista de artículos y depositarla en una JSP para que puedan mostrarse.

La figura siguiente muestra como el Front Controller crea objetos comando a partir de la petición del navegador:

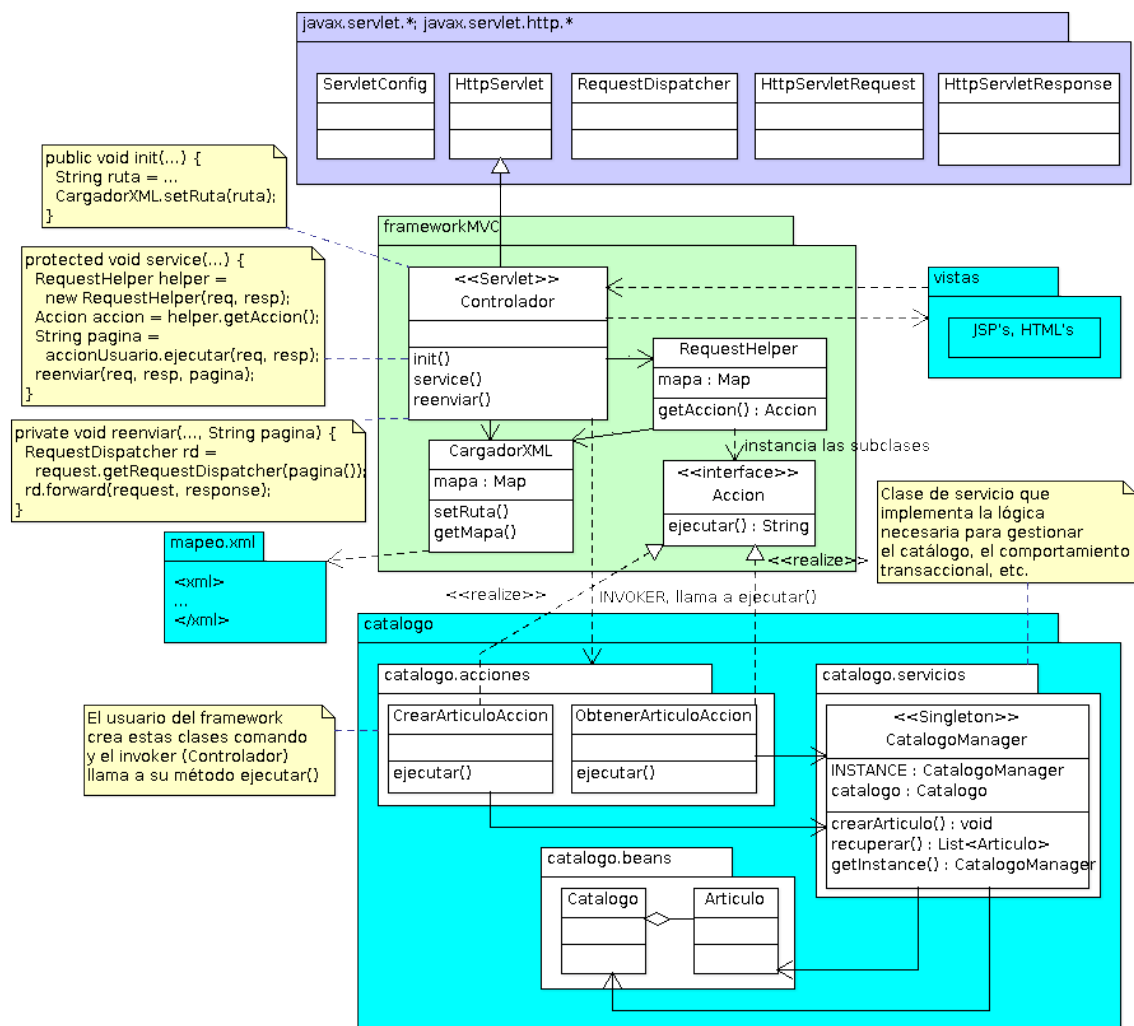


El trabajo principal de nuestro framework será instanciar y ejecutar la clase comando correspondiente a la petición enviada por el navegador; luego un rol de Invoker. En la figura anterior, estas clases se representan mediante elipses de color verde.

Las peticiones se envían al servidor en forma de cadena de texto (String) y el framework crea un objeto a partir de esta cadena. Para que esto sea posible, debe existir algún tipo de equivalencia entre cadena y nombre de clase de comando. Dado que las clases comando las crea el usuario del framework, será responsabilidad del mismo establecer esta correspondencia. Para ello, se utilizará un fichero en formato XML denominado mapeo.xml.

Para entender mejor todo lo explicado hasta el momento, a continuación veremos cómo crear el framework MVC y una pequeña aplicación web que lo utiliza para gestionar un catálogo de productos.

La figura siguiente muestra el sistema en su conjunto:



Legenda de colores del diagrama:

- En el paquete de color lila se representan las clases que utilizaremos del API Servlet.
- En el paquete de color verde se muestran las clases que comprenden el framework MVC que en breve confeccionaremos.

- En color azul (cian) se representan aquellos componentes que tendrá que crear el usuario de nuestro framework MVC para llevar a buen término la aplicación del catálogo. También crearemos estas clases en este documento.

Es normal que en este momento no se comprenda bien el diagrama. Lo que se espera es que conforme se avance en la lectura del documento se vaya asimilando el papel de cada componente del sistema y sus relaciones con otros componentes. No obstante, estaría bien como ejercicio intentar ahora reconocer en el diagrama los componentes que son propios del patrón Command.

Funcionamiento general de la aplicación

El navegador envía una petición al Servlet Controlador. A partir de aquí, pueden ejecutarse uno o dos métodos pertenecientes al ciclo de vida del Servlet (métodos que son invocados por el propio contenedor de Servlet -Tomcat, Jetty o similares):

Puede que se ejecute el método `init()`. Esto sucede sólo si aún no se ha iniciado el Servlet, como es el caso cuando se acaba de poner en funcionamiento el servidor de aplicaciones. En este supuesto, el método `init()` llamará al método estático `setRuta()` de la clase `CargadorXML` para indicarle dónde localizar el fichero XML con las parejas de equivalencia petición-nombre de clase comando. `CargadorXML` almacena esta ruta pero aún no realiza proceso alguno. Esta ruta siempre es relativa al directorio de instalación del servidor de aplicaciones, por lo que el único que puede proporcionarla es el Servlet mediante métodos específicos del API Servlet, como `getRealPath()` y similares.

Independientemente de si se ha ejecutado el apartado anterior, siempre se ejecutará el método `service()`, en el cual el controlador crea una instancia de la clase `RequestHelper`. Este objeto solicita a la clase `CargadorXML` un Map en el que las claves se corresponden con las peticiones enviadas por el navegador, mientras que los valores son los nombres -completamente cualificados- de las clases comando a instanciar en cada petición. La clase `CargadorXML` genera el Map mientras procesa el fichero de mapeo, trabajo que sólo realiza una vez, pues se trata de un proceso relativamente costoso. El objeto `RequestHelper`, una vez que obtiene del Map el nombre de la clase a instanciar, crea un objeto mediante el API Reflection y lo retorna al Controlador, quién en ese momento ya puede invocar al método `ejecutar()`,

consiguiendo así la ejecución de la lógica que el programador del framework implementó en su clase comando (la creación de un artículo, la obtención del catálogo, etc).

Una clase comando no debe implementar lógica de negocio, sino invocarla. Tal invocación puede o no producir resultados que la clase comandos deberá incorporar a una vista.

Es común que una clase comando ejecute la lógica de negocio utilizando un objeto de una clase de servicio, encargada de proporcionar todos los métodos que la capa de presentación necesite. En el diagrama anterior esto se refleja mediante el uso de una referencia de la clase CatalogoManager, la cual expone una serie de métodos para tratar con el catálogo (crear artículo, obtener catálogo, etc). Los principales objetivos de CatalogoManager son:

- Definir la lógica de negocio necesaria para la gestión del catálogo. Esto suele incluir políticas y reglas empresariales, como descuentos, promociones, etc.
- Determinar el comportamiento transaccional de la aplicación.
- Ocultar a la parte front-end las clases de bajo nivel y/o ligadas a alguna tecnología en particular (EJB, sistemas legacy, etc), así como los detalles del sistema de persistencia del catálogo.

Creación del framework MVC

Si bien el framework será una parte esencial de la aplicación del catálogo (o de cualquier otra aplicación final que realicemos), no es una aplicación web en sí misma. Por lo tanto, creamos un proyecto Java "normal" (no web) con el nombre 'DCTfrmwkMVC'.

A continuación creamos una carpeta con el nombre 'lib' e incluimos en ella la librería 'servlet-api.jar'.

Ahora creamos el paquete 'org.dct.frmwrkMVC.accion'. En este paquete vamos a crear la interfaz Accion. Como veremos más adelante, esta interfaz tendrá que ser implementada por toda aquella clase comando en las aplicaciones finales (como el catálogo).

Accion.java

```
package org.dct.frmwrkMVC.accion;

import java.io.IOException;

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

public interface Accion {

    public String ejecutar(HttpServletRequest peticion,
                           HttpServletResponse respuesta) throws
ServletException, IOException;

}
```

Creamos un nuevo paquete, denominado 'org.dct.frmwrkMVC.control', dentro del cual creamos la siguientes tres clases:

- CargadorXML.java
- RequestHelper.java
- Controlador.java

A continuación se muestra el código para cada una:

CargadorXML.java

```
package org.dct.frmwrkMVC.control;

/**
 * @author Daniel Colomer
```

```

*
* Clase que carga el fichero mapeo.xml y genera un HashMap,
* donde las claves son los nombres logicos de las operaciones
* (consulta, alta, etc) y los valores son las clases de accion
* que despacharan las peticiones
*/

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;

import org.xml.sax.SAXException;

import java.io.File;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

class CargadorXML {

    // Ubicacion y nombre del fichero XML
    private static String CONFIG_FILENAME;

    private static final String MAP_FILENAME = "mapeo.xml";

    private static Map<String, String> mapa;

```



```

/**
 * Constructor
 */
private CargadorXML() { // No instanciable
    mapa = new HashMap<String, String>();

    /*
     * La clase org.w3c.dom.Document contiene los componentes
'document',
     * 'element', etc
     */
    Document doc = null;

    // Analizar el documento
    try {

        /*
         * Obtener un DocumentBuilderFactory a partir de un
método estático
         * newInstance()
         */
        DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();

        // Obtener un DocumentBuilder a partir del
DocumentBuilderFactory
        // anterior
        DocumentBuilder db = dbf.newDocumentBuilder();

        // El método parse() de DocumentBuilder devuelve un
// objeto Document a partir de un InputStream (o un
File)

        doc = db.parse(new File(CONFIG_FILENAME));

```

```

/**
 * Alternativa File:
 * InputStream is = this.getClass().getClassLoader()
 *                     .getResourceAsStream(CONFIG_FILENAME);
 *                     doc = db.parse(is);
 */

// Recuperamos el elemento raíz y mostramos su nombre
Element root = doc.getDocumentElement();

// Mostramos el numero de elementos del documento
NodeList l = root.getElementsByTagName("peticion");

// Buscamos todas las entradas <peticion>
for (int i = 0; i < l.getLength(); i++) {

    // Leemos sus atributos
    String nombre = null, accion = null;

    int num_attr =
l.item(i).getAttributes().getLength();

    if (num_attr == 2) {
        String nomAttr =
l.item(i).getAttributes().item(0).getNodeName();

        if (nomAttr.equals("nombre")) {
            nombre = l.item(i)
                .getAttributes().item(0).getNodeValue();

        } else if (nomAttr.equals("accion")) {
            accion = l.item(i)
                .getAttributes().item(0).getNodeValue();

        }
    }
}

```

```

        nomAttr =
l.item(i).getAttributes().item(1).getNodeName();
        if (nomAttr.equals("nombre")) {
            nombre = l.item(i)
                .getAttributes().item(1).getNodeName();
odeValue();
        } else if (nomAttr.equals("accion")) {
            accion = l.item(i)
                .getAttributes().item(1).getNodeName();
odeValue();
        }
    } else { // No tiene dos atributos -> error
        System.out.println("Cada elemento
<peticion> debe tener 2 atributos exactamente.");
    }

    if (nombre != null && !nombre.equals("") &&
        accion != null && !accion.equals(""))
    {
        addMapeo(nombre, accion);
    }
}

//showMapa();

} catch (ParserConfigurationException e) {
    e.printStackTrace();
} catch (SAXException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

```

    static void addMapeo(String nombre, String accion) {
        mapa.put(nombre, accion);
    }

    static void showMapa() {
        for (Entry<String, String> e : mapa.entrySet()) {
            System.out.println(e.getKey() + "-" + e.getValue());
        }
    }

    static Map<String, String> getMapa() {
        if (mapa == null)
            new CargadorXML();
        return mapa;
    }

    static void setRuta(String ruta) {
        CONFIG_FILENAME = ruta + MAP_FILENAME;
    }
}

```

La clase anterior debería analizar correctamente un fichero XML como el siguiente:

mapeo.xml

```

<?xml version="1.0" encoding="UTF-8"?>

<mapeo>
    <peticion nombre="detalleArticulo"
        accion="acciones.DetalleArticuloAccion"/>

</mapeo>

```

Pasemos a ver a otra de las clases del framework:

RequestHelper.java

```
package org.dct.frmwrkMVC.control;

/**
 * Extrae de la request el nombre de la acción que se quiere ejecutar
 * y
 * se comprueba que existe una clase asociada a tal nombre. En caso
 * afirmativo se instancia un objeto de esa clase y se devuelve al
 * Controlador para que invoque el método ejecutar() de tal instancia.
 */

import java.util.Map;

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

import org.dct.frmwrkMVC.accion.Accion;

class RequestHelper {

    private HttpServletRequest request;
    private HttpServletResponse response;

    // Obtener el Map de asociación peticiones-clases
    private static Map<String, String> mapa = loadMap();

    private static final String ERR_MSG_XML =
        "Error: Revise que el fichero mapeo.xml contenga
    acciones asociadas a clases.";

    private static final String ERR_MSG_PETICION =
```

```

        "Error: La petición al controlador es incorrecta.";

    private static final String ERR_MSG_CLASE_NO_ENCONTRADA =
        "Error: Se ha indicado una clase en el fichero mapeo.xml
        que no puede localizarse. " +
        "¿Ha especificado su nombre correctamente y ha
        indicado la ruta de paquete(s)?";

    private static final String ERR_MSG_PETICION_NO_ENCONTRADA =
        "Error: Se ha indicado una petición que no puede
        localizarse en el fichero mapeo.xml.";

    /**
     * Constructor
     */
    RequestHelper(HttpServletRequest request, HttpServletResponse
    response) {
        this.request = request;
        this.response = response;
    }

    /**
     * A partir de la petición del usuario se obtiene una nueva
     instancia de
     * la clase asociada a la petición.
     *
     * @return un nuevo objeto Accion
     */
    Accion getAccion() {
        String token = limpiarPetición(request);

        if (esAdmisable(token)) {
            return obtenerAccion(token, request, response);
        }
    }

```

```

        } else {
            throw new IllegalArgumentException(ERR_MSG_PETICION);
        }
    }

    /**
     * Extraer de la petición el token que servirá como clave para
    el Map
     * de asociación entre peticiones y clases Accion
     *
     * @return
     */
    private String limpiarPeticion(HttpServletRequest request) {
        String url = request.getRequestURL().toString();

        String peticionConExtension =
            url.substring(url.lastIndexOf('/')+1,
                url.length());

        String token =
            peticionConExtension.substring(0,
                peticionConExtension.lastIndexOf('.')); // *.go

        return token;
    }

    /**
     * Verificar que la petición no esté vacía.
     */
    private boolean esAdmisible(String token) {
        return token != null && !token.equals("") &&
            token.length()>1;
    }

```

```

/**
 * A partir del parametro 'token' buscamos en el mapa la
subclase de Action a
 * instanciar y reflexivamente creamos un objeto.
 * @param token
 * @param request
 * @param response
 * @return un nuevo objeto Accion
 */

private Accion obtenerAccion(String token, HttpServletRequest
request, HttpServletResponse response) {

    String claseAccionUsuario = mapa.get(token);

    if (claseAccionUsuario == null) {

        throw new
RuntimeException(ERR_MSG_PETICION_NO_ENCONTRADA + ": " + token);
    }

    try {

        return (Accion)
Class.forName(claseAccionUsuario).newInstance();
    } catch (Exception e) {

        throw new
RuntimeException(ERR_MSG_CLASE_NO_ENCONTRADA + ": " +
claseAccionUsuario);
    }

}

/**
 * Obtener el hashMap en el que:

```



```

        * - Las claves son el nombre lógico de la petición web
        (consulta, alta ,...).

        * - Los valores son los nombres de las clases cuyos objetos
        tenemos que instanciar.

        */

    private static Map<String, String> loadMap() {
        Map<String, String> mapa = CargadorXML.getMapa();

        if (mapa == null || mapa.isEmpty()) {
            throw new RuntimeException(ERR\_MSG\_XML);
        }

        return mapa;
    }
}

```

Y por último, veamos el Servlet controller:

Controlador.java

```

package org.dct.frmwrkMVC.control;

/**
 * @autor Daniel Colomer
 *
 * Front Controller (Servlet controlador).
 */

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;

```

```

import javax.servlet.http.HttpServletResponse;

import org.dct.frmwrkMVC.accion.Accion;

public class Controlador extends HttpServlet {

    private static final long serialVersionUID = 1L;

    /**
     * Este método sólo se ejecuta una vez, cuando se carga el
Servlet.
     * Informar al cargador sobre la ruta de acceso al fichero de
mapeo de acciones
     */
    @Override
    public void init(ServletConfig config)
    {
        String ruta =
            config.getServletContext().getRealPath("WEB-INF") +
            System.getProperty("file.separator");
        CargadorXML.setRuta(ruta);
    }

    /**
     * Este método se ejecuta en cada petición del usuario, sea esta
GET o POST.
     *
     * Se utiliza un objeto HelperRequest para obtener una instancia
de la clase
     * de Accion asociada a la petición del usuario.
     *
     * A continuación se invoca el método ejecutar() sobre esta
instancia, con lo

```

```

    * que se consigue ejecutar la lógica implementada por el
programador del
    * framework en la acción.
    *
    * El método ejecutar() devuelve el nombre de la página JSP a la
que se le
    * tiene que pasar el control del programa una vez terminada la
acción.
    */
@Override
    protected void service(HttpServletRequest request,
                            HttpServletResponse response)
                            throws ServletException, IOException
    {
        RequestHelper helper = new RequestHelper(request,
response);
        Accion accionUsuario = helper.getAccion();
        String siguientePagina = accionUsuario.ejecutar(request,
response);
        reenviar(request, response, siguientePagina);
    }

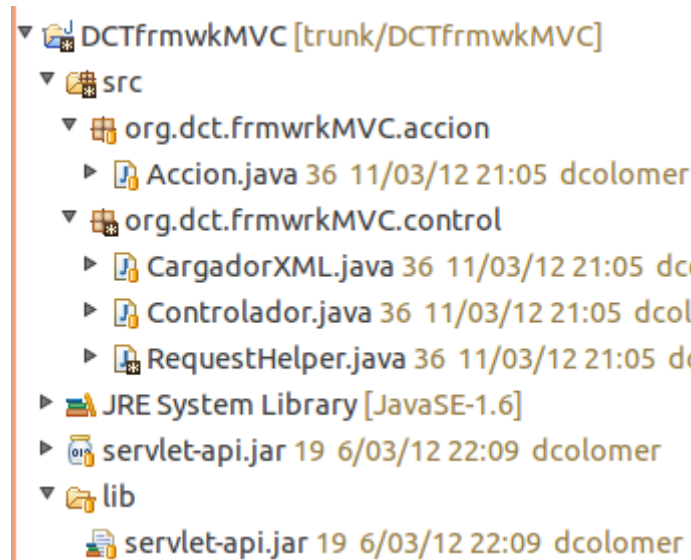
    /**
    * Pasar el control del programa a la siguiente JSP
    */
    private void reenviar(HttpServletRequest request,
HttpServletResponse response,
                            String pagina) throws ServletException, IOException
    {
        RequestDispatcher rd =
            request.getRequestDispatcher(pagina.toString());

        rd.forward(request, response);
    }

```

}

La imagen siguiente muestra la estructura de clases y paquetes para el framework:



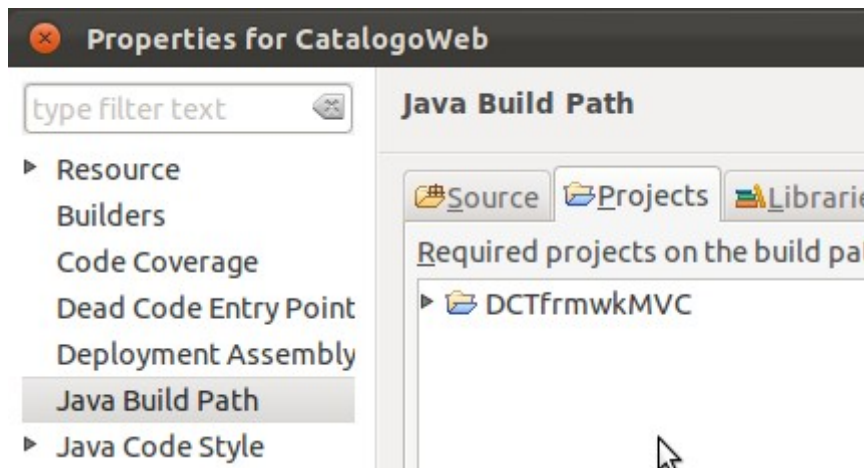
Pasemos a ver el proyecto de la aplicación final que se basará en el framework MVC.

Creación de la aplicación del catálogo web

Creemos un proyecto web dinámico que como nombre tenga 'CatalogoWeb'.

El proyecto utilizará como 'Targeted Runtime' Apache Tomcat, preferiblemente la versión 7, aunque no tendríamos que tener problemas con ningún servidor de aplicaciones.

Dado que vamos a utilizar clases e interfaces del framework MVC, para poder compilar necesitamos que 'CatalogoWeb' localice el proyecto 'DCTfrmwkMVC'. Para esto, nos dirigimos a las propiedades del proyecto y seleccionamos la entrada 'Java Build Path'. En las pestañas que aparecen a la derecha, pulsamos 'Projects' y seleccionamos el proyecto del framework:



Sigamos con la creación de los recursos para la aplicación final. Comenzamos por las clases relativas al modelo de datos de la aplicación: el catálogo y los artículos que lo componen.

En 'JavaResources->src' creamos el paquete 'org.dct.catalogoweb.beans'. Dentro de este paquete creamos las clases `Articulo.java` y `Catalogo.java`.

`Articulo.java`

```
package org.dct.catalogoweb.beans;
```

```
import java.math.BigDecimal;
```

```
public class Articulo {
```

```
    private String codigo;
```

```
    private String descripcion;
```

```
    private BigDecimal precio;
```

```
    public Articulo(String codigo, String descripcion, BigDecimal  
precio) {
```

```
        this.codigo = codigo;
```

```
        this.descripcion = descripcion;
```

```
        this.precio = precio;
```

```
    }
```

```

    public String getCodigo() { return codigo; }
    public String getDescripcion() { return descripcion; }
    public BigDecimal getPrecio() { return precio; }

    @Override
    public String toString() {
        return "Articulo [codigo=" + codigo + ", descripcion=" +
descripcion
        + ", precio=" + precio + "];"
    }
}

```

Catalogo.java

```

package org.dct.catalogoweb.beans;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

public class Catalogo {

    private static Map<String, Articulo> catalogo =
inicializarCatalogo();

    private static Map<String, Articulo> inicializarCatalogo() {
        Map<String, Articulo> catalogo = new LinkedHashMap<String,
Articulo>();
        catalogo.put("1000", new Articulo("1000", "Televisor plano
X100", new BigDecimal("540.50")));
        catalogo.put("1001", new Articulo("1001", "Televisor plano
X600", new BigDecimal("920.60")));
    }
}

```

```

        return catalogo;
    }

    public List<Articulo> recuperar() {

        List<Articulo> articulos = new ArrayList<Articulo>();
        if (!catalogo.isEmpty()) {
            for (Articulo articulo : catalogo.values()) {
                articulos.add(articulo);
            }
        }
        return articulos;
    }

    public void crear(Articulo articulo) {
        catalogo.put(articulo.getCodigo(), articulo);
    }

    public void eliminar(String codigo) {
        catalogo.remove(codigo);
    }

    public Articulo get(String codigo) {
        return catalogo.get(codigo);
    }
}

```

Creación de una clase de servicio. Esta clase abstrae a la capa de presentación de los detalles de implementación del catálogo, su sistema de persistencia y de cualquier otro aspecto de bajo nivel. Para ello proporciona un conjunto de métodos simples de usar. En nuestro caso, resulta hasta simplista pero en una aplicación real esta clase

aportaría mucho valor, proporcionando una fachada unificada de acceso a métodos pertenecientes a distintos componentes.

Creamos el paquete 'org.dct.catalogoweb.servicios' y dentro de él la clase 'CatalogoManager':

CatalogoManager.java

```
package org.dct.catalogoweb.servicios;

import java.util.List;

import org.dct.catalogoweb.beans.Articulo;
import org.dct.catalogoweb.beans.Catalogo;

public class CatalogoManager {

    private static CatalogoManager catalogoMgr = new
CatalogoManager();

    private Catalogo catalogo;

    private CatalogoManager() { // No instanciable
        catalogo = new Catalogo();
    }

    public static CatalogoManager getInstance() {
        return catalogoMgr;
    }

    public List<Articulo> recuperar() {
        return catalogo.recuperar();
    }

    public boolean crearArticulo(Articulo articulo) {
        String codigo = articulo.getCodigo();
```



```

        Artículo articuloExistente = catalogo.get(codigo);

        if (articuloExistente != null) {
            return false;
        }
        catalogo.crear(articulo);
        return true;
    }
}

```

Pasemos a crear las clases comando. Creamos un paquete denominado 'org.dct.catalogoweb.acciones', en cuyo interior definiremos las clases de la capa web responsables de recuperar el catálogo y de crear nuevos artículos.

Comenzamos por la clase que recupera el catálogo:

ObtenerCatalogoAccion.java

```

package org.dct.catalogoweb.acciones;

import java.io.IOException;
import java.util.List;

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

import org.dct.catalogoweb.beans.Articulo;
import org.dct.catalogoweb.servicios.CatalogoManager;
import org.dct.frmwrkMVC.accion.Accion;

```

```

public class ObtenerCatalogoAccion implements Accion {

    @Override

    public String ejecutar(HttpServletRequest peticion,
                           HttpServletResponse respuesta) throws
ServletException, IOException {

        CatalogoManager catalogoMgr =
            CatalogoManager.getInstance();

        List<Articulo> catalogo = catalogoMgr.recuperar();

        // Dejamos los resultados en la request
        peticion.setAttribute("catalogo", catalogo);

        return "mostrarCatalogo.jsp";
    }
}

```

La idea del código anterior es utilizar el objeto CatalogoManager para obtener una lista de artículos, la cual se deposita en la request para que quede accesible a la JSP encargada de mostrar una tabla con los artículos del catálogo. Notad que el último paso del método ejecutar() es retornar un String cuyo valor es el nombre de la JSP a la que el framework pasará el control del programa.

La otra clase comando es la responsable de crear un artículo.

CrearArticuloAccion.java

```
package org.dct.catalogoweb.acciones;

import java.io.IOException;
import java.math.BigDecimal;
import java.util.List;

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

import org.dct.catalogoweb.beans.Articulo;
import org.dct.catalogoweb.servicios.CatalogoManager;
import org.dct.frmwrkMVC.accion.Accion;

public class CrearArticuloAccion implements Accion {

    @Override
    public String ejecutar(HttpServletRequest peticion,
                           HttpServletResponse respuesta) throws
        ServletException, IOException {

        Articulo articulo = crearArticulo(peticion);

        if (articulo == null) {
            return "error.jsp";
        }

        CatalogoManager catalogoMgr =
            CatalogoManager.getInstance();
```

```

        boolean grabado = catalogoMgr.crearArticulo(articulo);

        if (grabado) {

            List<Articulo> catalogo = catalogoMgr.recuperar();

            // Dejamos los resultados en la request
            peticion.setAttribute("catalogo", catalogo);

            return "mostrarCatalogo.jsp";
        } else {
            return "error.jsp";
        }
    }

    private Articulo crearArticulo(HttpServletRequest peticion) {

        String paramCodigo = peticion.getParameter("codigo");
        String paramDescripcion =
            peticion.getParameter("descripcion");
        String paramPrecio = peticion.getParameter("precio");

        // Comprobar que los tres parametros tienen contenido
        if (paramCodigo != null && paramCodigo != null &&
            paramCodigo != null) {

            BigDecimal precio;

            try {

```

```

        // Comprobar que el precio es un valor numérico

        double precioDouble =
Double.parseDouble(paramPrecio);

        precio = new
BigDecimal(String.valueOf(precioDouble));

        return new Artículo(paramCodigo,
paramDescripcion, precio);

    } catch (NumberFormatException e) {

        return null;

    }

    } else {

        return null;

    }

    }

}

```

Del código anterior vemos que la clase utiliza un método interno para crear un objeto artículo a partir de los datos procedentes del formulario rellenado por el usuario. Si por cualquier motivo no se ha podido crear el artículo, el control del programa se pasa a una página de error, mientras que si todo ha ido bien se utiliza un objeto CatalogoManager para proceder a la grabación del artículo y pasar el control del programa a la página que muestra el catálogo, donde podremos ver el nuevo artículo creado junto con el resto de artículos.

Hasta aquí el código Java. El resto de código es XML, HTML y etiquetas JSP.

Comenzamos por los ficheros XML. Nuestra aplicación necesita dos ficheros XML. Ambos tienen que crearse en la carpeta WebContent->WEB-INF:

- web.xml: Descriptor de despliegue estándar JEE.

- mapeo.xml: Fichero de configuración propio de nuestro framework MVC que define las equivalencias entre nombres de petición y nombres de clases comando.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID"
version="2.5">

    <display-name>FrameworkMVC</display-name>

    <welcome-file-list>

        <welcome-file>index.html</welcome-file>

    </welcome-file-list>

    <servlet>

        <description></description>

        <display-name>Controlador</display-name>

        <servlet-name>Controlador</servlet-name>

        <servlet-class>org.dct.frmwrkMVC.control.Controlador</servlet-
class>

    </servlet>

    <servlet-mapping>

        <servlet-name>Controlador</servlet-name>

        <url-pattern>*.go</url-pattern>

    </servlet-mapping>

</web-app>
```

Notad que declaramos el Servlet Controlador, perteneciente al framework MVC, e indicamos que tal Servlet atenderá todas las peticiones cuya extensión sea '.go'. Esto nos permite discriminar aquellas peticiones enviadas por el navegador que no vayan dirigidas a nuestro framework.

Pasemos a ver el fichero de mapeo:

mapeo.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mapeo>
    <peticion nombre="mostrarcatalogo"
        accion="org.dct.catalogoweb.acciones.ObtenerCatalogoAccion"/>

    <peticion nombre="creararticulo"
        accion="org.dct.catalogoweb.acciones.CrearArticuloAccion"/>

</mapeo>
```

Como vemos, se trata de un fichero muy simple en el que se definen dos correspondencias nombre petición/acción:

mostrarcatalogo->ObtenerCatalogoAccion

creararticulo->CrearArticuloAccion

Fijaos en que los nombres no tienen la extensión '.go', ya que esta extensión sólo es necesaria para discriminar nombres procedentes del navegador a nivel de framework, esto es, a nivel API Servlet, mientras que en el fichero de mapeo nos encontramos en un contexto en el que no hay necesidad de discriminar nada.

Ahora pasemos a ver los ficheros CSS, HTML y JSP. Todos irán dentro de la carpeta WebContent.

Hoja de estilos

Para hacer la presentación un tanto interesante, aunque no mucho, se utiliza la siguiente hoja de estilos CSS:

estilos.css

```
body {
```

```
    font-family: Verdana, Arial, Helvetica, sans-serif;
    font-size: 12px;
    line-height: 24px;
    background: silver;
    width: 800px;
}
```

```
p.titulo {
    font-size: 18px;
    font-weight: bold;
    font-style: italic;
    font-family: Arial, Helvetica, sans-serif;
    padding-left: 4px;
    color: white;
    background: grey;
    border-bottom: 1px solid #ccc;
}
```

```
table.articulos {
    width: 750px;
    margin: 0 auto 1em auto;
    padding: 0em;
    text-align: middle;
    vertical-align: middle;
    background: none;
}
```

```
table.articulos th {
    background: #A6B4BF;
    color: white;
    font-style: italic;
```



```

        text-align: middle;
        font-size: 13px;
        border: 1px solid grey;
    }

    table.articulos tbody tr:nth-child(odd) /*impar*/ {
        background: #D6E7FA;
        color: #666;
        padding: 0px;
    }

    table.articulos tbody tr:nth-child(even) /*par*/ {
        background: #E8F1FE;
        padding: 0px;
        color: #666;
    }

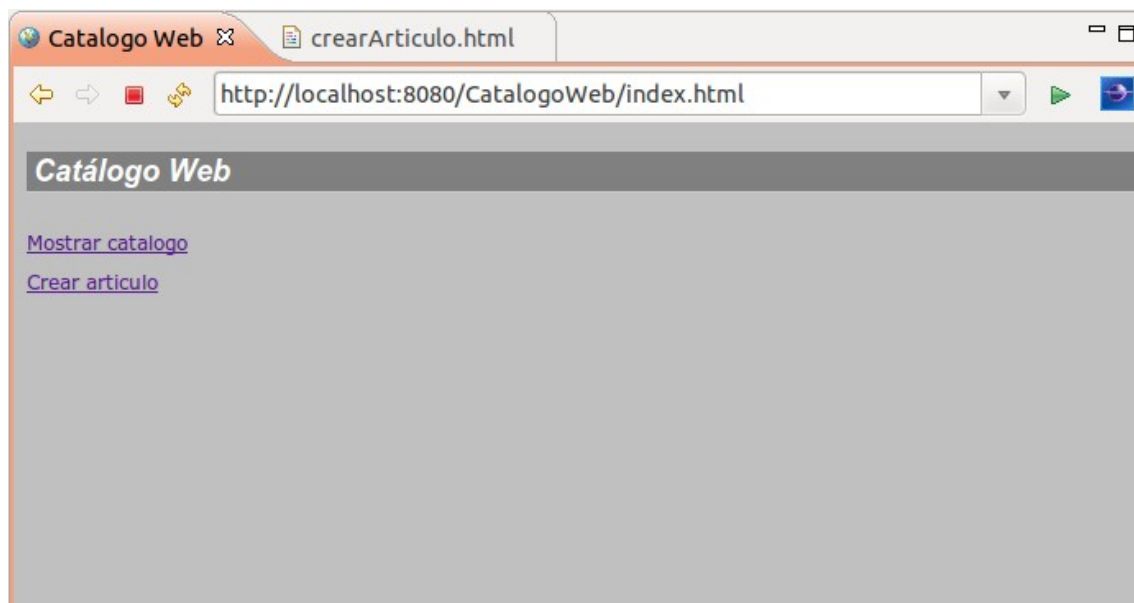
    table.articulos td { text-align: center; }

    table.articulos td.numerica {
        text-align: right;
        padding-right: 4px;
    }

```

La mayoría del código CSS anterior es relativo a la tabla que presenta el catálogo.

Veamos ahora la página inicial, cuyo aspecto será como el de la siguiente imagen:



index.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>Catalogo Web</title>
```

```
<link rel="stylesheet" type="text/css" href="estilos.css">
```

```
</head>
```

```
<body>
```

```
<p class="titulo">
```

```
Catálogo Web
```

```
</p>
```

```
<a href="mostrarcatalogo.go">Mostrar catalogo</a>

<br />

<a href="crearArticulo.html">Crear articulo</a>

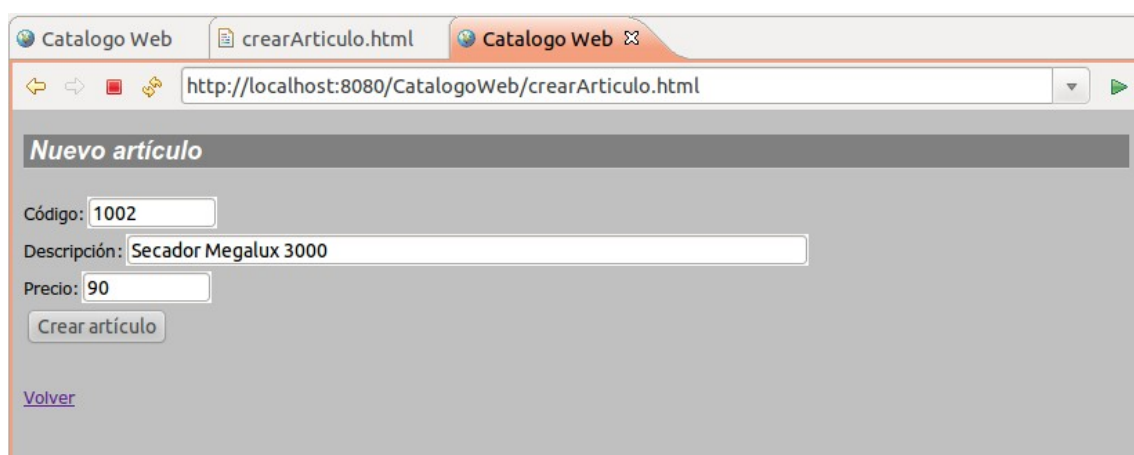

</body>

</html>
```

Como vemos del código anterior, para mostrar el catálogo enviamos al servidor la petición 'mostrarcatalogo.go', la cual será resuelta por nuestro framework consultando el fichero de mapeo y determinando que tendrá que instanciar la clase comando ObtenerCatalogoAccion e invocar su método ejecutar().

En cambio, para el caso de crear un artículo no se indica ninguna acción, sino el nombre de la página crearArticulo.html. Esto es así, porque para este segundo caso no queremos ejecutar ningún comando en el servidor, sino que nos aparezca un formulario en el que poder rellenar dos datos del nuevo artículo. Por tanto, queremos una redirección simple de una página a otra sin que medie ninguna acción intermedia (nuestro framework no intervendrá en estos casos). Lógicamente, una vez cumplimentado el formulario sí queremos que se lleve a cabo la ejecución de un comando: la grabación del artículo.

La figura siguiente muestra el formulario para crear nuevos artículos:



The screenshot shows a web browser window with two tabs: 'Catalogo Web' and 'crearArticulo.html'. The address bar displays 'http://localhost:8080/CatalogoWeb/crearArticulo.html'. The page content features a header 'Nuevo artículo' in a grey bar. Below this, there are three input fields: 'Código:' with the value '1002', 'Descripción:' with the value 'Secador Megalux 3000', and 'Precio:' with the value '90'. A 'Crear artículo' button is positioned below the 'Precio' field. At the bottom left, there is a 'Volver' link.

crearArticulo.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=UTF-8">
<title>Catalogo Web</title>

<link rel="stylesheet" type="text/css" href="estilos.css">

</head>
<body>

<p class="titulo">
Nuevo artículo
</p>

<form action="creararticulo.go">

Código:<input type="text" size="10" name="codigo"/>
<br />
Descripción:<input type="text" size="60" name="descripcion"/>
<br />
Precio:<input type="text" size="10" name="precio"/>
<br />

<input type="submit" value="Crear artículo" />
</form>
<br />
<a href="index.html">Volver</a>
</body>
```

</html>

Fijaos que esta página envía la petición 'creararticulo.go', la cual se acabará traduciendo en la ejecución del método ejecutar() de la clase CrearArticuloAccion.

La siguiente imagen muestra la tabla con todos los artículos, a la cual podemos llegar tanto después de crear un artículo como directamente desde la página de inicio:



The screenshot shows a web browser window with the address bar displaying 'http://localhost:8080/CatalogoWeb/creararticulo.go?codigo=1002&descripcion=Secador+Megalux+3000'. The page title is 'Listado de artículos'. Below the title is a table with three columns: 'Código', 'Descripción', and 'Precio'. The table contains three rows of data. Below the table is a link labeled 'Volver'.

Código	Descripción	Precio
1000	Televisor plano X100	540.50
1001	Televisor plano X600	920.60
1002	Secador Megalux 3000	90.0

[Volver](#)

Veamos el código:

mostrarCatalogo.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@ page import="org.dct.catalogoweb.beans.Articulo, java.util.List"
%>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

<!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>Catalogo Web</title>

<link rel="stylesheet" type="text/css" href="estilos.css">
```

```

</head>

<body>

<p class="titulo">
Listado de artículos
</p>

<table class="articulos">
    <thead>
        <tr>
            <th>Código</th><th>Descripción</th><th>Precio</th>
        </tr>
    </thead>
    <tbody>
        <c:forEach var="articulo" items="${
{requestScope['catalogo']}}">
            <tr>
                <td><c:out value="${articulo.codigo}" /></td>
                <td><c:out value="${articulo.descripcion}"
/></td>
                <td class="numerica"><fmt:formatNumber value="${
{articulo.precio}}" /></td>
            </tr>
        </c:forEach>
    </tbody>
</table>

<p />
<a href="index.html">Volver</a>
</body>
</html>

```

La página de error es lo más simple y genérica posible, lo cual tendríamos que mirar de mejorar en una aplicación más 'seria':

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!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>Insert title here</title>
</head>
<body>
Se ha producido un error
</body>
</html>
```

Librerías

Para que la aplicación web pueda encontrar las clases de terceros en tiempo de ejecución necesitamos los siguientes JAR's en la carpeta WebContent->WEB-INF->lib:

- DCTfrmwkMVC.jar
- jstl.jar
- standard.jar

Notad que la primera librería se trata de nuestro framework MVC.

La siguiente imagen muestra la distribución de los diferentes recursos del proyecto:

