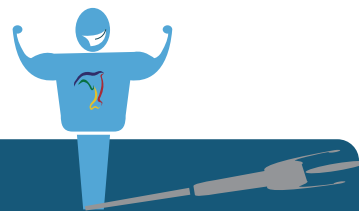


¿Qué ofrece Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

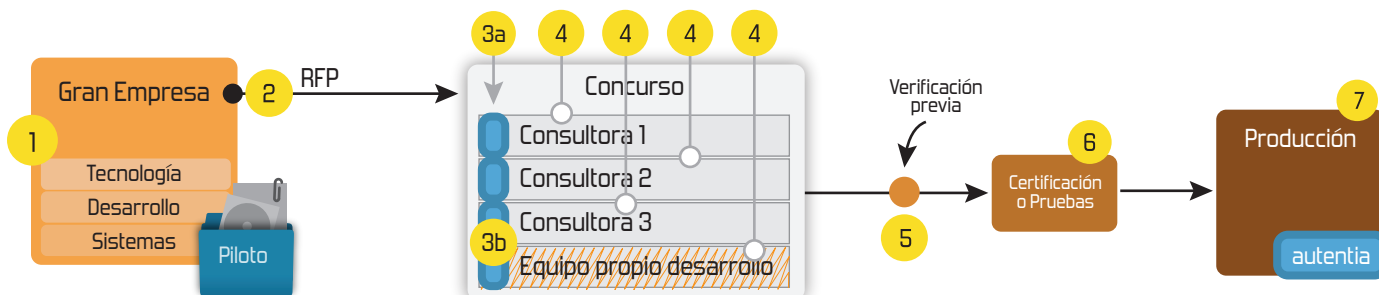
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces,
HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

Control de autenticación y
acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)



[Home](#) | [Quienes Somos](#) | [Empleo](#) | [Foros](#) | [Tutoriales](#) | [Servicios Gratuitos](#) | [Contacte](#)

	Tutorial desarrollado por: Enrique Medina Montenegro Puedes ver mi CV y contratarme en:
---	---

Descargar este documento en formato PDF [strutsb.pdf](#)

CÓMO CREAR UNA APLICACIÓN CON STRUTS PASO A PASO

Como ya se introduce en el tutorial de [Struts Jakarta](#), estamos ante un entorno (framework) normalizado de presentación. Dicho así puede quedar un poco confuso, pero veamos como encajándolo dentro de la arquitectura de desarrollo MVC (Modelo-Vista-Controlador) seguro que descubrimos cuánto nos puede ayudar Struts.

Struts se encarga de normalizar (o al menos intentarlo) el desarrollo de la capa Vista dentro de la arquitectura MVC. Sin embargo, es necesario que para tal fin, también proporcione mecanismos para trabajar con la capa C. Pero nunca Struts se mezclará con la capa del Modelo. Esta característica principal de Struts permite separar la lógica de presentación de la lógica del negocio, con las ya consabidas ventajas que este tipo de desarrollo supone.

Este tutorial que aquí se presenta pretende mostrar cómo llevar a cabo la tarea de desarrollar una aplicación Web utilizando la API de Struts; pero, además, está concebido de forma que se demuestra la independencia completa de Struts con el modelo de negocio y, como consecuencia, con la tecnología utilizada para implementar la capa M. Así, este tutorial utiliza:

- Struts para el desarrollo de la capa de Vista y Control (presentación Web)
- Un servicio JDBC simple para la capa de Modelo (sin EJBs)

Aunque en la mayoría de los casos el uso de Struts se aconseja en el desarrollo de aplicaciones muy complejas y, casi siempre, se utilizarán EJBs para la capa de negocio (modelo), la idea es presentar en este tutorial una primera aproximación a Struts de la manera más sencilla posible (a pesar de la complejidad de este framework). De esta forma, en un posterior tutorial de continuación, se mostrará cómo la inclusión de EJBs en nuestra aplicación no supondrá tocar ni una línea de código de las capas V y C, y sólo habrá que modificar (y ligeramente) la capa de Modelo.

Nuestra aplicación ejemplo: un carrito de la compra

Quizás la aplicación más ilustrativa y más sencilla de asimilar sea un carrito de la compra. Por ello, hemos decidido implementar este tipo de aplicación sin ningún tipo de pretensiones, ya que el objetivo de este tutorial es comprender y aprender a manejar Struts, y no crear una aplicación profesional de e-commerce, por ejemplo.

Entre las características que tendrá nuestra aplicación ejemplo destacamos el uso de:

- **ActionForms**, tanto estáticos como dinámicos (*DynaActionForm*).
- **Tiles**, para creación y trabajo con plantillas (páginas Web).
- **Acciones** únicas (*Action*) y de grupo (*DispatchAction*).
- **ActionMappings**, que permiten mapear las características de las acciones en un fichero externo XML.
- **ActionErrors**, para devolver mensajes de error en la validación de formularios ActionForms.
- **Contenedor**, para guardar toda la información necesaria de sesión.
- **Struts Tags**, librería de etiquetas que nos ahorrarán mucho código.
- **Patrones de diseño**: *Business Delegate*, *Business Interface* y *DTOs* (*Data Transfer Objects*) o *Value Objects*.
- **Servicios**, para conexión con la BD mediante *JDBC*.

Como podéis comprobar, se ha intentado hacer uso de prácticamente toda la tecnología inherente en Struts, aunque por cuestiones de complejidad, no se hace uso de aproximaciones más optimizadas (como usar un mapeador objeto/relacional, el validador de Struts, más patrones de diseño como *Dynamic Proxy*, etc.).

En cuanto a las herramientas utilizadas para desarrollar la aplicación, se ha optado por utilizar las más fáciles de manejar: un editor de texto (os recomiendo alguno que "sepa" Java para que os marque los errores de sintaxis, aunque yo he utilizado uno genérico) y JBoss (ya que es gratuito y soporta EJBs). La idea de hacerlo así se debe a que el uso de herramientas de mercado como JBuilder, obvian algunos detalles del desarrollo que son imprescindibles conocer para entender y comprender cómo funciona Struts.

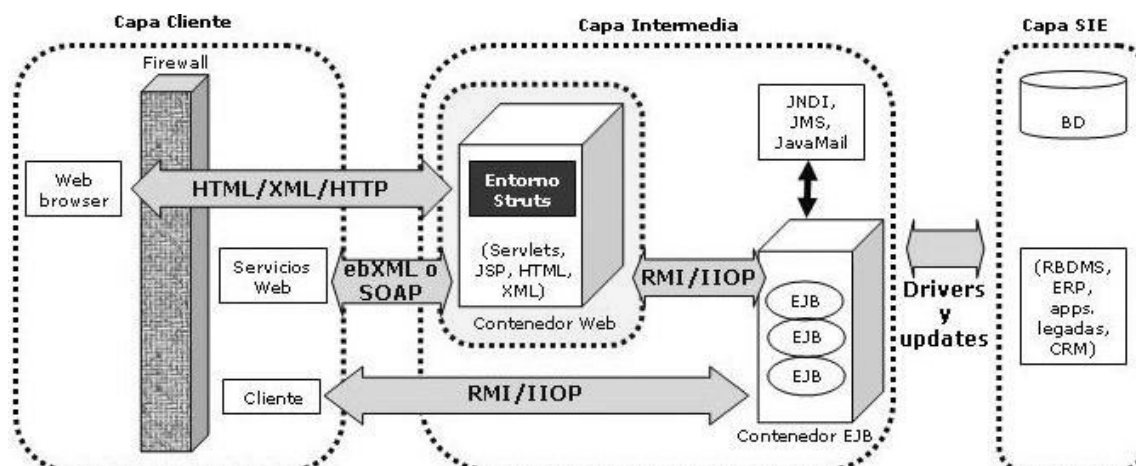
¿Por dónde empiezo?

Esta pregunta es básica a la hora de afrontar un proyecto complejo de desarrollo, sea de la naturaleza que sea. Este tutorial no pretende desarrollar una ingeniería del software, pero sí desmenuzar de una forma directa los pasos a seguir en el desarrollo de nuestra aplicación.

Gracias a la arquitectura MVC, podríamos desarrollar por un lado la interfaz del cliente Web, y delegar en otro equipo de desarrollo la parte de la lógica de negocio. En nuestro caso primero desarrollaremos la parte de negocio, para terminar implementando la capa de presentación y control (más que nada, por claridad en el desglose de los conceptos y su aplicación práctica).

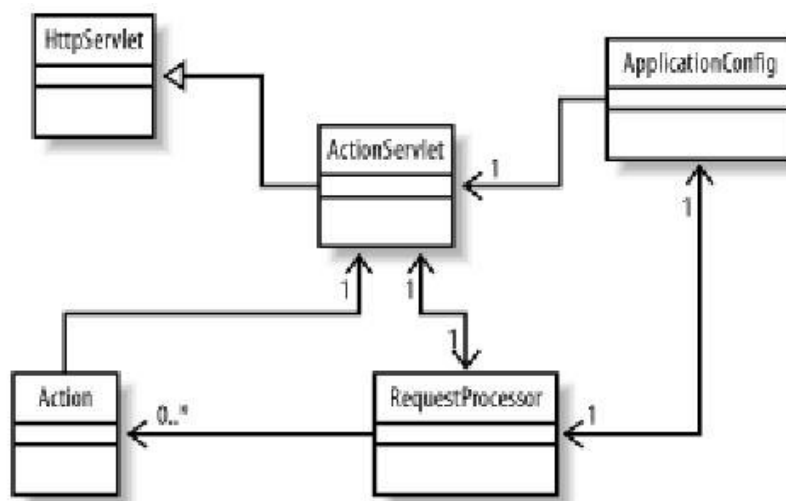
Entender cómo trabaja Struts

Antes de entrar en detalles específicos de nuestra aplicación, debemos entender cómo trabaja Struts para poder afrontar con éxito otro tipo de desarrollos y aplicaciones.



Como se aprecia en la figura, las aplicaciones Struts residen en un contenedor Web (dentro de un servidor de aplicaciones) y pueden hacer uso de los servicios proporcionados por el contenedor, como el manejo de peticiones HTTP o HTTPS. Esto permite al desarrollador olvidarse de la lógica de negocio. Así, Struts hace uso de varias áreas de recursos compartidos para almacenar objetos: petición (`javax.servlet.http.HttpServletRequest`), sesión (`javax.servlet.http.HttpSession`), aplicación (`javax.servlet.ServletContext`) y página (`javax.servlet.jsp.PageContext`).

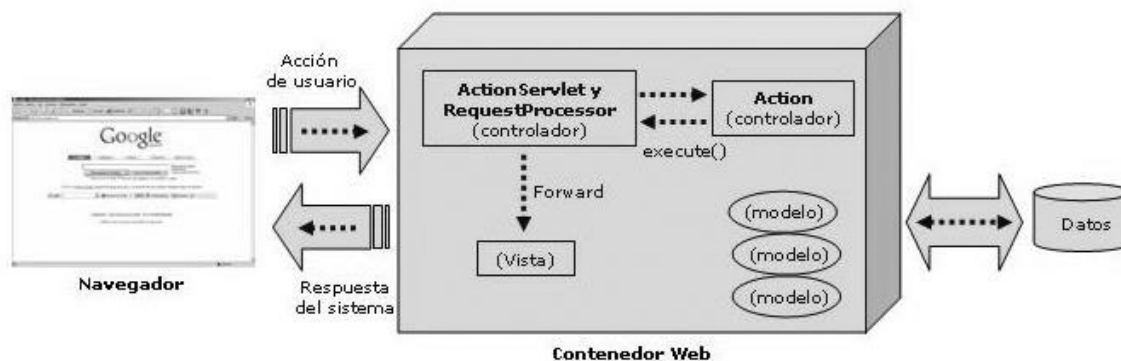
Desde el punto de vista de la arquitectura MVC, las clases que proporciona Struts respecto de la capa C son:



- **ActionServlet**, clase que extiende `javax.servlet.http.HttpServlet` responsable del empaquetado y enrutado del tráfico HTTP hacia el manejador apropiado dentro del entorno (framework). No es abstracta, y por tanto se puede utilizar directamente en cualquier aplicación.
- **RequestProcessor**, clase que permite desacoplar el proceso de petición (request process) del `ActionServlet` y así poder modificar cómo se procesa la petición (haciendo un 'subclass' de `RequestProcessor`).
- **Action**, clase que independiza la petición del cliente del modelo de negocio. Es una extensión del componente de control (capa C) y permite llevar a cabo funciones como autorización, logging, o validación de sesión, antes de invocar la lógica de negocio. Su método más importante es:

```
public ActionForward execute(ActionMapping mapping, HttpServletRequest request, HttpServletResponse response) throws Exception;
```

- **ActionMapping**, clase que representa una acción de mapeado que se define en el fichero de configuración de Struts. Indica al controlador que instancia de `Action` se debe invocar en cada petición.
- **ActionForward**, clase que representa el destino al cual el controlador debe enviar el control una vez que una `Action` se ha completado (nos evitamos meter el 'forward' en la página JSP).



Con respecto de la capa M, ya hemos comentado que Struts no fue diseñado para trabajar con esta capa. Sin embargo, es obvio que Struts recibirá información de esta capa (aunque no sepa cómo está implementada). Así, la mejor forma de solucionar cómo Struts se comunica con la capa de negocio de forma que sea completamente independiente de la tecnología de implementación de ésta, es utilizar un patrón de diseño, como por ejemplo DTOs (Data Transfer Objects), también llamados Value Objects (VO).

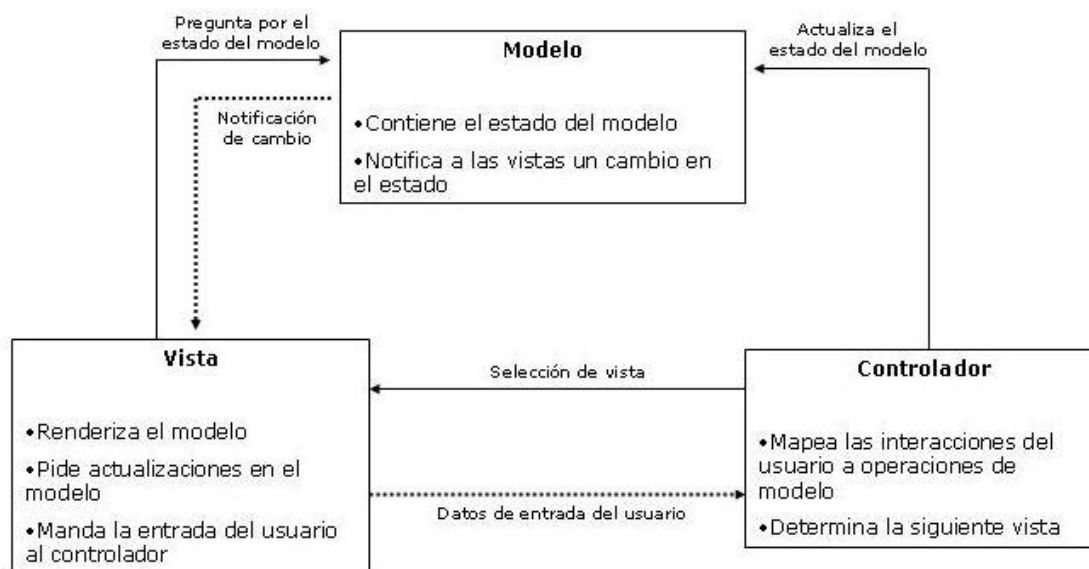
La idea es que Struts reciba la información en forma de vistas (VOs), pero no sepa cómo se han creado esas vistas. Para ello también necesitaremos implementar un nuevo patrón de diseño muy conocido, el Business Delegate. Con este patrón, crearemos un servicio perteneciente a la capa de negocio, que servirá de nexo de unión con la capa de control y será a través del cual Struts pedirá y recibir los únicos objetos que entiende y con los que sabe trabajar: los VOs. Por el otro lado, la capa de negocio sólo sabrá que puede recibir peticiones del servicio de acceso a BD a través de JDBC y devolverá VOs, sin importarle ni quién los pide ni quién los usa a continuación (no os preocupéis que con los ejemplos se verá mucho más claro).

Por último, con respecto a la capa V, los elementos de los que hace uso Struts son (algunos ya los conocemos):

- HTML
- Value Objects (vistas)
- **ActionForms**, clase que permite pasar datos de entrada hacia delante y hacia atrás entre el usuario y la capa de negocio.
- JavaServer Pages
- Struts Tags (librería de etiquetas)
- Otros recursos Java

Paso a paso: la capa del modelo de negocio

Veamos, tal y como se ha comentado, en primer lugar cómo ir desarrollando el código de la lógica de negocio para nuestra aplicación del carrito de la compra.



Así, primero creamos las vistas (VOs) que necesitaremos en nuestra aplicación. Para optimizar la implementación, primero creamos una superclase [BaseVO](#) que extenderán el resto de vistas. La creamos que implemente `java.io.Serializable` para que pueda ser referenciada desde un interfaz remoto (cuando implementemos por ejemplo los EJBs):

```
package com.tutorial.struts.vo;

import java.sql.Timestamp;

// Clase Value Object de la que todas las vistas deberían descender.
public class BaseVO implements java.io.Serializable
{
    private int id;
    private Timestamp timeCreated = null;
```

```

private String description;
private String name;

// Constructor por defecto.
public BaseVO()
{
    super();
    setTimeCreated(new Timestamp(System.currentTimeMillis()));
}

// Constructor a partir de los datos.
public BaseVO(int id, String name, String desc)
{
    this();

    this.id = id;
    this.name = name;
    this.description = desc;
}

public void setName(String name)
{
    this.name = name;
}

public void setTimeCreated(Timestamp now)
{
    timeCreated = now;
}

public void setDescription(String description)
{
    this.description = description;
}

public void setId(int id)
{
    this.id = id;
}

public String getName()
{
    return name;
}

public String getDescription()
{
    return description;
}

public int getId()
{
    return id;
}

public Timestamp getTimeCreated()
{
    return timeCreated;
}
}

```

Como se puede observar, la clase implementa un JavaBean muy simple con los métodos set/get necesarios. Ahora implementamos una clase que extiende de BaseVO para representar a los productos ([ProductoVO](#)):

```

package com.tutorial.struts.vo;

// Clase Value Object que implementa la vista del producto.
public class ProductoVO extends BaseVO
{
    // Datos del VO Producto.
    private String smallImageURL;
    private double basePrice;

    // Constructor por defecto.
    public ProductoVO()
    {
        super();
    }

    // Constructor a partir de los datos.
    public ProductoVO(int id, String name, String desc, double price, String smallImageURL)
    {
        super(id, name, desc);

        setBasePrice(price);
        setSmallImageURL(smallImageURL);
    }
}

```

```

    }

    // ... resto de métodos get/set ...
}

```

Y también una clase que represente la vista del usuario ([UserVO](#)), que utilizaremos posteriormente para crear el contenedor de los datos de la sesión:

```

package com.tutorial.struts.vo;

import com.tutorial.struts.vo.BaseVO;

// Clase Value Object que implementa una vista del usuario.
public class UserVO extends BaseVO
{
    private String lastName;
    private String firstName;
    private String emailAddress;

    public UserVO()
    {
        super();
    }

    // ... resto de métodos get/set ...
}

```

Ya tenemos creadas todas las vistas (VOs) que necesitamos para nuestra aplicación. Ahora vamos a centrarnos en la implementación del servicio que conectará mediante JDBC con nuestra BD y devolverá la vista adecuada (patrón Business Delegate). Para ello, utilizamos el patrón de diseño Business Interface, según el cual primero creamos un interface a modo de '*caja negra*', que será el único que conozca el cliente. Así, incluso podremos modificar el comportamiento del servicio sin que se entere Struts. El código para la clase [ICarritoService](#) se muestra a continuación:

```

package com.tutorial.struts.service;

import javax.servlet.ServletContext;
import java.util.List;

// Imports de clases propias de la aplicación.
import com.tutorial.struts.vo.ProductoVO;
import com.tutorial.struts.vo.UserVO;

public interface ICarritoService
{
    public List getListaProducto();

    public ProductoVO getDetalleProducto(String itemId);

    public void setServletContext(ServletContext ctx);

    public UserVO authenticate(String email, String password);

    public void logout(String email);

    public void destroy();
}

```

Aquí deben declararse todos los métodos de negocio que utilizará nuestra aplicación y que podrán ser llamados desde la parte del cliente. A continuación, implementamos el interface mediante la clase [CarritoServiceImpl](#), que es donde va la lógica de negocio:

```

package com.tutorial.struts.service;

import java.util.List;
import java.util.LinkedList;
import javax.servlet.ServletContext;
import java.sql.*;
import java.io.*;

// Imports de las clases propias de la aplicación.
import com.tutorial.struts.vo.ProductoVO;
import com.tutorial.struts.vo.UserVO;

// Implementamos el servicio que nos da acceso a los datos de la aplicación (BD).
public class CarritoServiceImpl implements ICarritoService
{
    ServletContext servletContext = null;

    // Referencia de conexión a una BD con JDBC.
    Connection con = null;

    /**

```

```

* Creamos el servicio, que incluye inicializar la conexión con la BD..
**/
public CarritoServiceImpl()
{
    super();

    // Abrimos la conexión con la BD.
    init();
}

public void setServletContext(ServletContext ctx)
{
    this.servletContext = ctx;
}

public ServletContext getServletContext()
{
    return servletContext;
}

/**
 * Devolvemos la lista de todos los productos.
 **/
public List getListaProducto()
{
    // Devolvemos una lista de los productos.
    List listaProducto = new LinkedList();
    ResultSet result = null;

    try
    {
        // Componemos la sentencia SQL para obtener los productos.
        String query = "SELECT * FROM PRODUCTO";

        // Ejecutamos la query y obtenemos el resultado.
        Statement stmt = con.createStatement();
        result = stmt.executeQuery(query);

        int id;
        String nombre, descripcion, smallImageURL;
        double basePrice;

        while (result.next())
        {
            id = result.getInt("id");
            nombre = result.getString("nombre");
            descripcion = result.getString("descripcion");
            smallImageURL = result.getString("smallImageURL");
            basePrice = result.getDouble("basePrice");

            ProductoVO productoVO =
new ProductoVO(id, nombre, descripcion, basePrice, smallImageURL);

            System.out.println(productoVO.toString());

            listaProducto.add(productoVO);
        }
    }
    catch (SQLException se)
    {
        System.err.println("Se ha producido un error de BD.");
        System.err.println(se.getMessage());
    }

    return listaProducto;
}

/**
 * Devuelve una vista detallada de un producto.
 **/
public ProductoVO getDetalleProducto(String productoId)
{
    ResultSet result = null;
    ProductoVO productoVO = null;

    try
    {
        // Componemos la sentencia SQL para obtener los productos.
        String query = "SELECT * FROM PRODUCTO WHERE ID = " + productoId;

        // Ejecutamos la query y obtenemos el resultado.
        Statement stmt = con.createStatement();
        result = stmt.executeQuery(query);

        // Vemos si no ha devuelto ningún resultado.
        if (!result.next())
        {
            throw new SQLException();
        }
    }

```

```

    }

    // Construimos una VO para el producto.
    String nombre, descripcion, smallImageURL;
    double basePrice;
    int id = result.getInt("id");

    nombre = result.getString("nombre");
    descripcion = result.getString("descripcion");
    smallImageURL = result.getString("smallImageURL");
    basePrice = result.getDouble("basePrice");

    productoVO = new ProductoVO(id, nombre, descripcion, basePrice, smallImageURL);
}
catch (SQLException se)
{
    System.err.println("Se ha producido un error de BD.");
    System.err.println(se.getMessage());
}

return productoVO;
}

/**
 * Autenticamos los credenciales del usuario y devolvemos un VO o null.
 */
public UserVO authenticate(String email, String password)
{
    ResultSet result = null;
    UserVO userVO = null;

    try
    {
        // Componemos la sentencia SQL para obtener los productos.
        String query = "SELECT * FROM USUARIO WHERE EMAIL = ? AND PASSWORD = ?";

        PreparedStatement ps = con.prepareStatement(query);

        // Pasamos los parámetros a la query.
        ps.setString(1, email);
        ps.setString(2, password);

        // Ejecutamos la query y obtenemos el resultado.
        result = ps.executeQuery();
        if (result.next())
        {
            // Construimos el VO a partir de los datos de la query.
            userVO = new UserVO();

            userVO.setId(result.getInt("id"));
            userVO.setFirstName(result.getString("FirstName"));
            userVO.setLastName(result.getString("LastName"));
            userVO.setEmailAddress(result.getString("Email"));
        }
    }
    catch (SQLException se)
    {
        System.err.println("Se ha producido un error de BD.");
        System.err.println(se.getMessage());
    }

    return userVO;
}

/**
 * Desloguea al usuario del sistema.
 */
public void logout(String email)
{
    // No hacemos nada de momento ...
}

public void destroy()
{
    // Cerramos la conexión con la BD.
    try
    {
        con.close();
    }
    catch (SQLException se)
    {
        System.err.println("Se ha producido un error al cerrar la conexión de BD.");
        System.err.println(se.getMessage());
    }
}

/**
 * Abre la conexión con la BD.

```



```

/**/
private void init()
{
    // Aquí debe ir la carga del driver y la conexión
    try
    {
        Class.forName("org.gjt.mm.mysql.Driver");
        con = DriverManager.getConnection("jdbc:mysql://localhost/carrito", "root", "");
    }
    catch (SQLException se)
    {
        System.err.println("Se ha producido un error al abrir la conexión de BD.");
        System.err.println(se.getMessage());
    }
    catch (java.lang.ClassNotFoundException s)
    {
        System.out.println("No se encuentra la clase "+ s.toString());
    }
}
}

```

Una vez implementado el servicio, necesitamos crear una clase factoría de servicios para que el cliente pueda crear el servicio y hacer uso de él. Para ello tenemos la clase [ICarritoServiceFactory](#):

```

package com.tutorial.struts.service;

public interface ICarritoServiceFactory
{
    public ICarritoService createService()
        throws ClassNotFoundException, IllegalAccessException, InstantiationException;

    public void destroy();
}

```

La idea es igual que en el caso anterior. Ahora la clase que implementa el interface, [CarritoServiceFactory](#):

```

package com.tutorial.struts.service;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import org.apache.struts.action.PlugIn;
import org.apache.struts.action.ActionServlet;
import org.apache.struts.config.ModuleConfig;
import com.tutorial.struts.service.*;

/**
 * Una factoría para crear instancias del servicio para la aplicación.
 */
public class CarritoServiceFactory implements ICarritoServiceFactory, PlugIn
{
    private ActionServlet servlet = null;
    String serviceClassname = "com.tutorial.struts.service.CarritoServiceImpl";

    public ICarritoService createService()
    throws ClassNotFoundException, IllegalAccessException, InstantiationException
    {
        ICarritoService instance = (ICarritoService)Class.forName(serviceClassname).newInstance();

        instance.setServletContext(servlet.getServletContext());

        return instance;
    }

    public void init(ActionServlet servlet, ModuleConfig config) throws ServletException
    {
        // Nos guardamos el servlet para después.
        this.servlet = servlet;

        // Guardamos la factoría del servicio en el contexto del servlet que Struts utiliza como controlador.
        // Así estará siempre disponible para quien la necesite dentro de Struts.
        servlet.getServletContext().setAttribute("com.tutorial.struts.service.ICarritoServiceFactory", this);
    }

    public void destroy()
    {
        // No hacemos nada de momento ...
    }
}

```

Explicaremos con detalle la implementación de la factoría de servicios. Primero, implementa la clase PlugIn, exclusiva de Struts, que permite que una clase externa a Struts se cargue como un módulo adicional en el entorno; en otras palabras, permite introducir en el contexto del controlador de Struts el servicio para que esté disponible siempre (public void init(ActionServlet servlet, ModuleConfig config) throws ServletException).

Luego vemos también que proporciona el método a través del cual el cliente podrá crear una instancia del servicio siempre que lo necesite (public ICarritoService createService() throws ClassNotFoundException, IllegalAccessException, InstantiationException).

Para decirle a Struts que se trata de un PlugIn y que lo cargue al inicializar la aplicación debemos incluir lo siguiente en el fichero externo struts-config.xml:

```
<plug-in className="com.tutorial.struts.service.CarritoServiceFactory"/>
```

Paso a paso: la capa del controlador Struts

Una vez hemos creado las clases e interfaces necesarios para la lógica de negocio, pasamos a desarrollar la parte correspondiente a la capa de control. Como vimos anteriormente, esta capa sí forma parte del entorno Struts y, por lo tanto, hará uso de la API de Struts en su desarrollo.

Primero vamos a extender las clases de control de Struts para adaptarlas a nuestras necesidades. Empezaremos por extender la clase org.apache.struts.action.Action para que, cada vez que realicemos alguna acción Struts, tengamos disponible una instancia del servicio, tal y como lo hemos implementado anteriormente. La idea es que siempre que ejecutemos una acción podamos usar los métodos del servicio para obtener los productos, autenticarnos, etc. El código de la clase abstracta [CustomBaseAction](#) es:

```
package com.tutorial.struts;

import java.util.Collection;
import java.util.LinkedList;
import java.util.List;
import java.util.Locale;
import java.util.Iterator;
import javax.servlet.http.*;
import org.apache.struts.action.*;

// Imports de clases propias de la aplicación.
import com.tutorial.struts.service.ICarritoService;
import com.tutorial.struts.service.ICarritoServiceFactory;

// Clase abstracta de Action que todas las acciones deben extender.
abstract public class CustomBaseAction extends Action
{
    protected ICarritoService getCarritoService()
    {
        // Obtenemos la factoría que nos permitirá crear el servicio.
        ICarritoServiceFactory factory = (ICarritoServiceFactory)getApplicationObject
("com.tutorial.struts.service.ICarritoServiceFactory");
        ICarritoService service = null;

        try
        {
            service = factory.createService();
        }
        catch(Exception ex)
        {
            System.out.println("Error al crear el servicio...");
            System.out.println(ex.getMessage());
        }

        return service;
    }

    // Método auxiliar para obtener un objeto de la sesión por su nombre.
    protected Object getSessionObject(HttpServletRequest req, String attrName)
    {
        Object sessionObj = null;
        HttpSession session = req.getSession(false);

        if (session != null)
        {
            sessionObj = session.getAttribute(attrName);
        }

        return sessionObj;
    }

    // Obtenemos el contenedor del usuario que está en la sesión.
    protected UserContainer getUserContainer(HttpServletRequest request)
    {
        UserContainer userContainer = (UserContainer)getSessionObject(request, "UserContainer");

        // Creamos un UserContainer para este usuario si no existe ...
        if(userContainer == null)
        {
            userContainer = new UserContainer();

            HttpSession session = request.getSession(true);
            session.setAttribute("UserContainer", userContainer);
        }

        return userContainer;
    }
}
```

```

    }

    // Obtenemos un objeto del contexto de la aplicación por su nombre.
    protected Object getApplicationObject(String attrName)
    {
        return servlet.getServletContext().getAttribute(attrName);
    }
}

```

Además, también hacemos lo mismo extendiendo la clase `org.apache.struts.action.DispatchAction`. El código de la clase [CustomDispatchAction](#) es prácticamente idéntico al anterior:

```

package com.tutorial.struts;

import javax.servlet.http.*;
import com.tutorial.struts.service.ICarritoService;
import com.tutorial.struts.service.ICarritoServiceFactory;
import org.apache.struts.actions.DispatchAction;

public class CustomDispatchAction extends DispatchAction
{
    // ... igual que la clase anterior CustomBaseAction ...

}

```

Los dos métodos más importantes son el que permite crear el servicio para el cliente (`protected ICarritoService getCarritoService()`) y el que permite obtener el contenedor del usuario con los datos de la sesión (`protected UserContainer getUserContainer(HttpServletRequest request)`). Por ello, debemos a continuación crear la clase que implementa el contenedor [UserContainer](#), que contendrá tanto la información del usuario como el carrito de la compra actual:

```

package com.tutorial.struts;

import java.util.Locale;
import javax.servlet.http.HttpSessionBindingListener;
import javax.servlet.http.HttpSessionBindingEvent;
import com.tutorial.struts.vo.UserVO;

// Almacena la información del usuario en su sesión.
// Implementa el interfaz HttpSessionBindingListener para que se le notifique
// el timeout y así pueda liberar los recursos de la sesión.
public class UserContainer implements HttpSessionBindingListener
{
    // El carrito de la compra del usuario.
    private Carrito cart = null;
    // Almacenamos la información del usuario.
    private UserVO userVO = null;

    // Constructor por defecto.
    public UserContainer()
    {
        super();
        initialize();
    }

    // El contenedor llama a este método cuando se termina la sesión.
    public void valueUnbound(HttpSessionBindingEvent event)
    {
        // Liberamos recursos de la sesión.
        System.out.println( "Liberando sesión...");
        cleanUp();
    }

    public Carrito getCart()
    {
        return cart;
    }

    public void setCart(Carrito newCart)
    {
        cart = newCart;
    }

    // El contenedor llama a este método cuando comienza la sesión.
    public void valueBound(HttpSessionBindingEvent event)
    {
        // No hacemos nada por ahora ...
    }

    public UserVO getUserVO()
    {
        return userVO;
    }
}

```

```

    public void setUserVO(UserVO newVO)
    {
        userVO = newVO;
    }

    // Inicialización de los datos del usuario en sesión.
    private void initialize()
    {
        // Create a new Shopping cart for this user
        cart = new Carrito();
    }

    // Liberamos recursos de la sesión.
    public void cleanUp()
    {
        setUserVO(null);
        cart = new Carrito();
    }
}

```

Como se puede apreciar, hemos tomado la decisión de no hacer el carrito persistente, es decir, trabajamos con el carrito en memoria dentro del contenedor en la sesión, de forma que se pierde cuando nos salimos de la aplicación. Una buena idea para optimizar la aplicación sería dotar de persistencia al carrito y que el usuario pudiera recuperar su pedido en posteriores sesiones.

Pero para nuestro ejemplo, el carrito se implementará como un JavaBean que irá dentro del contenedor en la sesión del usuario. Para ello creamos dos clases: una llamada [Carrito](#), y otra llamada [CarritoProducto](#), que serán las líneas de detalle de nuestro carrito:

```

package com.tutorial.struts;

import java.util.List;
import java.util.LinkedList;

// Clase JavaBean que implementa un carrito de la compra en memoria (no es persistente).
public class Carrito
{
    private List productos = null;

    // Constructor por defecto (crea una lista enlazada vacía).
    public Carrito()
    {
        productos = new LinkedList();
    }

    public void addProducto(CarritoProducto newProducto)
    {
        // Miramos si este producto ya está en el carrito.
        // Si ya está simplemente incrementamos su cantidad.
        CarritoProducto cartProducto = findProducto(Integer.toString(newProducto.getId()));

        if (cartProducto != null)
        {
            cartProducto.setQuantity(cartProducto.getQuantity() + newProducto.getQuantity());
        }
        else
        {
            // Es un producto que no estaba; lo añadimos al carrito.
            productos.add(newProducto);
        }
    }

    // Añadimos una lista de productos.
    public void setProductos(List otherProductos)
    {
        productos.addAll(otherProductos);
    }

    public void setSize(int size)
    {
        // El tamaño es el de la lista enlazada.
        // Pero hay que implementar este método al tratarse de un JavaBean.
    }

    public int getSize()
    {
        return productos.size();
    }

    public void empty()
    {
        productos.clear();
    }

    public double getTotalPrice()
    {
        double total = 0.0;
    }
}

```

```

        int size = productos.size();

        for (int i = 0; i < size; i++)
        {
            total += ((CarritoProducto)productos.get(i)).getImporte();
        }

        return total;
    }

    public void removeProducto(String productoId)
    {
        CarritoProducto producto = findProducto(productoId);

        if (producto != null)
        {
            productos.remove(producto);
        }
    }

    public void removeProductos(List productoIds)
    {
        if (productoIds != null)
        {
            int size = productoIds.size();

            for (int i = 0; i < size; i++)
            {
                removeProducto((String)productoIds.get(i));
            }
        }
    }

    public void updateQuantity(String productoId, int newQty)
    {
        CarritoProducto producto = findProducto(productoId);

        if (producto != null)
        {
            producto.setQuantity(newQty);
        }
    }

    public List getProductos()
    {
        return productos;
    }

    private CarritoProducto findProducto(String productoId)
    {
        CarritoProducto producto = null;
        int size = getSize();

        for (int i = 0; i < size; i++)
        {
            CarritoProducto cartProducto = (CarritoProducto)productos.get(i);

            if (productoId.equals(Integer.toString(cartProducto.getId())))
            {
                producto = cartProducto;
                break;
            }
        }

        return producto;
    }
}

package com.tutorial.struts;

import com.tutorial.struts.vo.ProductoVO;

// Clase JavaBean que almacena las líneas de un carrito de la compra.
public class CarritoProducto
{
    // Importe es el precio de unidad * cantidad.
    private double importe = 0.0;
    private ProductoVO producto = null;
    // Cantidad por defecto a 1.
    private int quantity = 1;

    // Constructor de línea a partir de un producto y su cantidad.
    public CarritoProducto(ProductoVO producto, int qty)
    {
        this.producto = producto;
        this.quantity = qty;
        calculateImporte();
    }
}

```

```

    public void setProducto(ProductoVO newProducto)
    {
        producto = newProducto;
        calculateImporte();
    }

    public void setImporte(double newPrice)
    {
        importe = newPrice;
    }

    public String getDescription()
    {
        return producto.getDescription();
    }

    public int getId()
    {
        return producto.getId();
    }

    public String getName()
    {
        return producto.getName();
    }

    public int getQuantity()
    {
        return quantity;
    }

    public void setQuantity(int newQuantity)
    {
        quantity = newQuantity;
        calculateImporte();
    }

    public ProductoVO getProducto()
    {
        return producto;
    }

    public double getBasePrice()
    {
        return producto.getBasePrice();
    }

    public double getImporte()
    {
        return importe;
    }

    private void calculateImporte()
    {
        if (producto.getBasePrice() != 0)
        {
            importe = producto.getBasePrice() * getQuantity();
        }
    }
}

```

La implementación del carrito no tiene mayor complicación y se ajusta a una implementación típica de la mayoría de aplicaciones que hacen uso de este elemento.

Paso a paso: la capa de presentación Struts

Hasta ahora hemos ido desarrollando nuestra aplicación ejemplo de forma similar a cómo lo haríamos si no conociésemos Struts. La única característica específica que hemos estado obligados a tener en cuenta es la existencia de las Action (o Dispatcher). Sin embargo, lo más seguro es que en nuestras anteriores aplicaciones sin conocer Struts ya nos hubiéramos montado mecanismos de control al estilo de las acciones.

Pero cuando llegamos a la capa Vista, entonces Struts cambia toda nuestra forma de trabajar, ya que es aquí donde haremos mayor uso de la tecnología y normalización que propone Struts. Veamos con calma como desarrollar cada uno de los componentes de la capa de presentación.

Lo primero, y para seguir el sentido de desarrollo que estamos llevando, vamos a implementar las acciones Struts. Para ello debemos tener muy claro las necesidades de nuestra aplicación. En nuestro ejemplo necesitaremos:

- Loguearnos y desloguearnos del sistema
- Ver la lista de productos en venta
- Ver el detalle de un producto en particular
- Ver nuestro perfil
- Trabajar con el carrito de la compra: ver, modificar, actualizar, añadir, borrar, etc.

Según estas acciones expuestas, necesitamos ahora decidir cómo las implementaremos usando Struts. Como ya se ha comentado al

principio del tutorial, Struts proporciona dos clases para las acciones: `org.apache.struts.actions.Action` y `org.apache.struts.actions.DispatchAction` (extendidas por nosotros como `CustomBaseAction` y `CustomDispatchAction`, respectivamente). En nuestro caso vamos a utilizar ambas para mostrar su uso: usaremos `CustomDispatchAction` para las acciones exclusivas del carrito, y `CustomBaseAction` para el resto.

El grupo de acciones exclusivas del carrito lo implementamos en [CarritoActions](#) de la siguiente forma:

```
package com.tutorial.struts.actions;

import java.io.IOException;
import java.text.Format;
import java.text.NumberFormat;
import java.util.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import org.apache.struts.actions.DispatchAction;

import com.tutorial.struts.service.ICarritoService;
import com.tutorial.struts.vo.ProductoVO;
import com.tutorial.struts.UserContainer;
import com.tutorial.struts.CarritoProducto;
import com.tutorial.struts.Carrito;
import com.tutorial.struts.CustomDispatchAction;

/**
 * Implementa toda la funcionalidad mediante acciones de un carrito de la compra.
 */
public class CarritoActions extends CustomDispatchAction
{
    /**
     * Este método simplemente reenvía al estado de Success,
     * el cual debería representar la página shoppingcart.jsp.
     */
    public ActionForward view(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        // Nos aseguramos que el usuario está en sesión.
        UserContainer userContainer = getUserContainer(request);

        return mapping.findForward("Success");
    }

    /**
     * Este método actualiza los productos y cantidades del carrito.
     */
    public ActionForward update(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        updateProductos(request);
        updateQuantities(request);

        return mapping.findForward("Success");
    }

    /**
     * Este método añade un producto al carrito según los parámetros
     * id y qty de la petición.
     */
    public ActionForward addProducto(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        System.out.println("Añadiendo producto al carrito");

        UserContainer userContainer = getUserContainer(request);

        System.out.println("El usuario es " + userContainer);
        System.out.println("Empezamos");
        System.out.println("Buscando: " + request.getSession().getAttribute("UserContainer"));
        System.out.println("Terminamos" );

        // Obtenemos el id del producto a buscar.
        String productoId = request.getParameter("id");
        String qtyParameter = request.getParameter("cantidad");

        int quantity = 1; // Valor por defecto a añadir.
        if(qtyParameter != null)
        {
            Locale userLocale = request.getLocale();
            Format nbrFormat = NumberFormat.getNumberInstance(userLocale);

            try
            {
                Object obj = nbrFormat.parseObject(qtyParameter);
                quantity = ((Number)obj).intValue();
            }
        }
    }
}
```

Este ejemplo demuestra las ventajas de uso de `org.apache.struts.actions.DispatchAction` para agrupar acciones comunes, frente al uso de múltiples acciones individuales.

Pero vamos a centrarnos en las partes del código más interesantes. Quizás lo primero que llama la atención es que la mayoría de los métodos devuelven un objeto `ActionForward`. Este objeto es exclusivo de Struts y permite definir en un fichero externo (`struts-config.xml`) mediante mapeados las características de las acciones: nombre, ruta, ámbito, validación, qué hacer si éxito y qué hacer si fracaso (hacia donde redirigir la petición). De esta forma, en nuestro código podemos indicar mediante `return mapping.findForward("Success")` que continúe el flujo de la aplicación por otra acción o JSP. Veamos el mapeado para `CarritoActions`:

```
<action
  path="/carrito"
  type="com.tutorial.struts.actions.CarritoActions"
  scope="request"
  input="/jsp/carrito.jsp"
  validate="false"
  parameter="method">

  <forward name="Success" path="/jsp/carrito.jsp" redirect="true"/>
</action>
```


Si nos fijamos en los atributos de la etiqueta `action` podemos comprobar cómo indicamos las características de nuestra acción. Como en este caso se trata de un grupo de acciones (`DispatchAction`), necesitamos indicar `parameter="method"`. De esta forma podremos llamar a la subacción dentro del grupo de acciones de la forma `/carrito?method=view`, por ejemplo. Además, indicamos que si todo ha ido bien, nos redirigimos a la página JSP llamada `/jsp/carrito.jsp` (la lógica de redirecciones la controla Struts y nos olvidamos de tener que incluirla en los JSP, lo cual es una gran ventaja, como podréis suponer).

Un método muy importante, que demuestra como desde la parte del control de la presentación podemos llamar a métodos de negocio a través de nuestro mecanismo de servicio implementado antes, es `AddProducto`. En él se puede observar cómo hacemos uso de `getCreateService()` de la superclase `CustomDispatchAction` para obtener una instancia de `ICarritoService`, y así poder invocar los métodos de negocio necesarios: `serviceImpl.getDetalleProducto(productId)`. Este mecanismo proporciona gran reusabilidad a nuestra aplicación.

También destacar el método `getUserContainer()`, implementado también en la superclase `CustomDispatchAction`, que nos permite obtener el contenedor con los datos del usuario y el carrito a través de la sesión.

Pasemos ahora a la acción [GetListaProductoAction](#), encargada de devolver una lista de productos existentes en nuestra BD:

```
package com.tutorial.struts.actions;

import javax.servlet.http.*;
import org.apache.struts.action.*;
import java.util.List;

// Imports de clases propias de la aplicación.
import com.tutorial.struts.CustomBaseAction;
import com.tutorial.struts.vo.ProductoVO;
import com.tutorial.struts.service.ICarritoService;

public class GetListaProductoAction extends CustomBaseAction
{
    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        // Creamos el servicio para obtener la lista de productos.
        ICarritoService serviceImpl = getCarritoService();
        List listaProducto = serviceImpl.getListaProducto();

        // Almacenamos el resultado en la sesión.
        request.setAttribute("ListaProductoKey", listaProducto);

        // Devolvemos el Action Forward para el caso de Success.
        return mapping.findForward("Success");
    }
}
```

En el caso de una acción del tipo `CustomBaseAction`, ya comentamos que se debía implementar el método `execute()`. De nuevo se puede comprobar cómo se hace uso similar al caso anterior del servicio almacenado en el contexto de `ActionServlet` (recordemos) para invocar métodos de la capa de negocio de forma transparente al cliente. Luego metemos el resultado como un atributo de la petición (no de la sesión, ojo) y nos redirigimos donde indique el mapeado de la acción en el fichero `struts-config.xml`:

```
<action
    path="/viewlistaproducto"
    input="/jsp/listaproducto.jsp"
    type="com.tutorial.struts.actions.GetListaProductoAction"
    scope="request"
    validate="false">

    <forward name="Success" path="/jsp/listaproducto.jsp"/>
</action>
```

En este mapeado observamos que no hay `parameter="method"` puesto que se trata de una acción simple que se indica en el atributo `type="com.tutorial.struts.actions.GetListaProductoAction"`.

El resto de acciones (no se detallan por cuestiones de espacio) son [LoginAction](#), [LogoutAction](#), [GetDetalleProductoAction](#) y [GetPerfilAction](#), para cada una de las cuales deberemos mapear sus características en el fichero `struts-config.xml`:

```
<action
    path="/login"
    type="com.tutorial.struts.actions.LoginAction"
    scope="request"
    name="loginForm"
    validate="true"
    input="/jsp/login.jsp">

    <forward name="Success" path="/action/viewperfil" redirect="true"/>
    <forward name="Failure" path="/action/login" redirect="true"/>
</action>

<action
    path="/logout"
    type="com.tutorial.struts.actions.LogoutAction"
    scope="request">
```

```

        <forward name="Success" path="/action/login" redirect="true"/>
    </action>

    <action
        path="/viewdetalleproducto"
        name="detalleProductoForm"
        input="/jsp/index.jsp"
        type="com.tutorial.struts.actions.GetDetalleProductoAction"
        scope="request"
        validate="false">

        <forward name="Success" path="/jsp/detalleproducto.jsp"/>
    </action>

    <action
        path="/viewperfil"
        name="perfilForm"
        input="/jsp/index.jsp"
        type="com.tutorial.struts.actions.GetPerfilAction"
        scope="request"
        validate="false">

        <forward name="Success" path="/jsp/perfil.jsp"/>
    </action>

```

Pero además, Struts nos permite definir mapeados de acciones genéricas, como por ejemplo un simple 'forward':

```

    <action
        path="/viewcarrito"
        parameter="/jsp/carrito.jsp"
        type="org.apache.struts.actions.ForwardAction"
        scope="request"
        validate="false">
    </action>

```

Como se puede comprobar fácilmente en el mapeado, indicamos que se trata de una acción predefinida en `type="org.apache.struts.actions.ForwardAction"`.

Con todas las acciones ya creadas, sólo nos queda la parte de la interfaz con la que trabajará el usuario. Para nuestra aplicación usaremos un navegador Web y, por tanto, tecnología JSP para presentar resultados e información. Pero además, haremos uso de las Struts Tags para facilitar la tarea de crear formularios, trabajar con los datos, etc.

Lo primero de todo es entender los mecanismos que nos proporciona Struts para las tareas exclusivas de presentación. Así, para cada página HTML que trabaje con datos, deberíamos usar un `ActionForm` e implementar obligatoriamente dos métodos de la superclase:

```

public void reset(ActionMapping mapping, HttpServletRequest request);

public ActionErrors validate(ActionMapping mapping, HttpServletRequest request);

```

Para hacer esta tarde más fácil todavía, Struts proporciona una forma dinámica de trabajar con formularios mediante la superclase `org.apache.struts.action.DynaActionForm`. Este tipo de formulario utiliza un `Map` para almacenar valores internamente a partir de la estructura definida en nuestros VOs, por ejemplo.

Según esto, veamos a continuación un par de ejemplos sacados de nuestra aplicación que ilustran tanto un 'action form' construido manualmente, como uno generado de forma automática. El del primer tipo se implementa mediante [LoginForm](#):

```

package com.tutorial.struts.forms;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;

/**
 * Bean de formulario para que el usuario se valide.
 */
public class LoginForm extends ActionForm
{
    private String password = null;
    private String email = null;

    public void setEmail(String email)
    {
        this.email = email;
    }

    public String getEmail()
    {
        return (this.email);
    }

    public String getPassword()
    {
        return (this.password);
    }
}

```

```

public void setPassword(String password)
{
    this.password = password;
}

/**
 * Validamos las propiedades que se han establecido para esta petición HTTP,
 * y devolvemos un objeto <code>ActionErrors</code> que encapsula cualquier
 * error de validación que encontremos. Si no se encuentran errores, devolvemos
 * <code>null</code> o un objeto <code>ActionErrors</code> sin mensajes de error.
 */
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request)
{
    ActionErrors errors = new ActionErrors();

    if (getEmail() == null || getEmail().length() < 1)
    {
        errors.add(ActionErrors.GLOBAL_ERROR, new ActionError("global.required", "email"));
    }

    if(getPassword() == null || getPassword().length() < 1)
    {
        errors.add(ActionErrors.GLOBAL_ERROR, new ActionError("global.required", "password"));
    }

    return errors;
}

/**
 * Resetea todas las propiedades a sus valores por defecto.
 */
public void reset(ActionMapping mapping, HttpServletRequest request)
{
    this.password = null;
    this.email = null;
}
}

```

Como se puede comprobar, una subclase `ActionForm` debe implementar los dos métodos que veíamos anteriormente. Además, se trató de un simple `JavaBean` con sus métodos `set/get`, que son utilizados por `Struts` para poder trabajar con los datos del formulario y hacerlos disponibles al usuario. Pero, además, podemos hacer uso de `ActionErrors` para presentar errores, por ejemplo de validación. Al igual que las acciones, los `ActionForm` hay que declararlos en el fichero `struts-config.xml`, en la sección `<form-beans>`:

```
<form-bean name="loginForm" type="com.tutorial.struts.forms.LoginForm"/>
```

En cuanto al segundo tipo de formularios, no hay necesidad de hacer un 'subclass' de `ActionForm`. Simplemente se declaran en el fichero XML de la siguiente forma:

```

<form-bean
    name="detalleProductoForm" dynamic="true"
    type="org.apache.struts.action.DynaActionForm">

    <form-property name="detalle" type="com.tutorial.struts.vo.ProductoVO"/>
</form-bean>

<form-bean
    name="perfilForm" dynamic="true"
    type="org.apache.struts.action.DynaActionForm">

    <form-property name="perfil" type="com.tutorial.struts.vo.UserVO"/>
</form-bean>

```

Hemos de indicar que se trata de un `DynaActionForm` y a partir de que `JavaBean` obtendrá su estructura. Si nos fijamos, es prácticamente igual al `ActionForm` anterior, sólo que como los métodos `set/get` ya están definidos en los `VOs`, los aprovechamos y nos ahorramos tener que redefinirlos.

Una vez creados los formularios ya sólo nos queda comenzar a desarrollar los JSP. `Struts` también nos ayuda en esta tarea a través de los `Tags`, que nos permiten acceder encapsular tareas relacionadas con diseño HTML, lógica (iteraciones, condiciones, etc.) y trabajo con `beans` (`ActionForms`). Estas últimas 'tags', las relacionadas con los `bean`, proporcionan también un mecanismo de internacionalización de nuestra aplicación, al permitirnos declarar un fichero externo de cadenas de texto para utilizar en nuestras páginas web.

Para indicar que vamos a utilizar un fichero de recursos para nuestra aplicación, debemos crear un fichero externo situado a partir de `WEB-INF/classes`, con extensión `.properties` y que debemos declarar en el fichero `struts-config.xml` de la siguiente forma:

```
<message-resources parameter="resources.ApplicationResource"/>
```

Como podemos apreciar, le decimos a `Struts` que busque los recursos de mensaje en un subdirectorio llamado `resources` (dentro de `WEB-INF/classes`) y en un fichero llamado `ApplicationResource` (sin la extensión `.properties`). En la documentación de `Struts` viene como crear más de un fichero de recursos de mensajes (por ejemplo, se podría crear uno para las 'labels' de la aplicación, otro para los mensajes de error, otro para los 'alt' de las imágenes, etc.).

Ya sólo nos queda entonces la parte de los JSP exclusivamente. Para ello hemos utilizado el paquete `Tiles` incorporado a `Struts`, por el cual podemos trabajar con plantillas de una forma muy práctica. Esto nos permitirá realizar cambios en el diseño sin tener que

modificar cada uno de los JSP de nuestra aplicación.

Veamos lo primero de todo como se define un 'layout' para luego crear una plantilla a partir de nuestro layout [defaultLayout.jsp](#) (el uso de Struts Tags queda fuera del alcance de este tutorial, aunque son muy intuitivas):

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles"%>

<html:html locale="true">

<head>
<title><bean:message key="global.title"/></title>
<html:base/>
</head>

<body topmargin="0" leftmargin="0" bgcolor="#9999CC">
<font face="Verdana, Arial, Helvetica, sans-serif" size="2" color="#000000">

<table width="100%" border="1" cellspacing="0" cellpadding="0" border="0">

<tr>
<!-- Información de cabecera-->
<td colspan="2"><tiles:insert attribute="header"/></td>
</tr>

<tr>
<!-- Barra de menú-->
<td valign="top" width="25%"><tiles:insert attribute="menubar"/></td>
<!-- Información principal-->
<td width="75%" valign="top"><tiles:insert attribute="body-content"/></td>
</tr>

<tr>
<!-- Información de pie de página-->
<td colspan="2"><tiles:insert attribute="copyright"/></td>
</tr>

</table>

</font>
</body>

</html:html>
```

Si nos fijamos en la distribución que hemos diseñado, se trata de un 'layout' típico con una parte superior, una media dividida en dos (izquierda menú y derecha donde se trabaja y presenta la información) y una parte inferior, a modo de copyright, por ejemplo.

Ahora sólo nos queda definir un JSP que haga uso en forma de plantilla de este 'layout'. En la especificación del 'layout' anterior, utilizamos la etiqueta `<tiles:insert>` para especificar un atributo que nos servirá para indicar ahora en la plantilla cómo insertar el diseño de cada zona. Veamos el código de [template.jsp](#):

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles"%>

<tiles:definition id="template.default" page="defaultLayout.jsp" scope="request">

<tiles:put name="header" value="header.jsp"/>
<tiles:put name="menubar" value="menu.jsp"/>
<tiles:put name="copyright" value="copyright.jsp"/>

</tiles:definition>
```

Fijaos que sumamente sencillo es crear un template a partir del 'layout'. Sólo declaramos un nombre para este template `id="template.default"` y qué 'layout' estamos utilizando `page="defaultLayout.jsp"`. Luego simplemente indicamos las zonas fijas de nuestra plantilla, que para nuestro caso serán todas menos la zona central donde presentaremos la información (en una aplicación más compleja pueden incluso ser todas las zonas variables).

De esta forma, cualquiera de nuestras futuras páginas Web podremos generarla a partir de un par de JSP: uno para indicar que utilizaremos la plantilla, y el otro para realmente implementar el código de la página Web. Veamos un ejemplo ilustrativo de nuestra aplicación con [carrito.jsp](#) y [carrito_body.jsp](#):

```
carrito.jsp

<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles"%>
<%@include file="template.jsp"%>

<tiles:insert beanName="template.default" beanScope="request">
<tiles:put name="body-content" value="carrito_body.jsp"/>

</tiles:insert>

carrito_body.jsp
```

04/08/2004

```

<html:link page="/action/viewlistaproducto">
Continuar comprando
</html:link>
</p>

</div>
</form>
</p>
</blockquote>
</logic:greaterThan>

<br>

```

Aunque no podemos entrar en detalle del código aquí ilustrado, sí merece la pena destacar el uso de etiquetas de lógica (<logic:greaterThan> o <logic:iterate>), etiquetas para mostrar los elementos de los formularios ActionForm y DynaActionForm definidos antes (<bean:write>) y etiquetas de HTML (<html:link>).

Paso a paso: unir todo el trabajo

Ya hemos terminado toda la parte del código para toda la arquitectura y ahora sólo nos queda desplegar la aplicación. Puesto que esta tarea ya será conocida por todos, simplemente se muestra a continuación las modificaciones exclusivas de Struts en el descriptor de despliegue (web.xml):

```

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

<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/tds/web-app_2_2.dtd">

<web-app>

  <servlet>
    <servlet-name>carrito</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
      <param-name>debug</param-name>
      <param-value>3</param-value>
    </init-param>
    <init-param>
      <param-name>detail</param-name>
      <param-value>3</param-value>
    </init-param>
    <init-param>
      <param-name>application</param-name>
      <param-value>ApplicationResources</param-value>
    </init-param>
    <init-param>
      <param-name>validating</param-name>
      <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>carrito</servlet-name>
    <url-pattern>/action/*</url-pattern>
  </servlet-mapping>

  <!-- The Welcome File List -->
  <welcome-file-list>
    <welcome-file>jsp/login.jsp</welcome-file>
  </welcome-file-list>

  <!-- Template Tag Library Descriptor -->
  <taglib>
    <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
  </taglib>

```

```

    <taglib>
      <taglib-uri>/WEB-INF/struts-template.tld</taglib-uri>
      <taglib-location>/WEB-INF/struts-template.tld</taglib-location>
    </taglib>

    <taglib>
      <taglib-uri>/WEB-INF/struts-nested.tld</taglib-uri>
      <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
    </taglib>

  </web-app>

```

y el fichero de configuración de Struts (struts-config.xml) completo:

```

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

<!DOCTYPE struts-config PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>

  <form-beans>

    <form-bean name="loginForm" type="com.tutorial.struts.forms.LoginForm"/>

    <form-bean name="detalleProductoForm" dynamic="true"
      type="org.apache.struts.action.DynaActionForm">
      <form-property name="detalle" type="com.tutorial.struts.vo.ProductoVO"/>
    </form-bean>

    <form-bean name="perfilForm" dynamic="true"
      type="org.apache.struts.action.DynaActionForm">
      <form-property name="perfil" type="com.tutorial.struts.vo.UserVO"/>
    </form-bean>

  </form-beans>

  <action-mappings>

    <action
      path="/login"
      type="com.tutorial.struts.actions.LoginAction"
      scope="request"
      name="loginForm"
      validate="true"
      input="/jsp/login.jsp">
      <forward name="Success" path="/action/viewperfil" redirect="true"/>
      <forward name="Failure" path="/action/login" redirect="true"/>
    </action>

    <action
      path="/logout"
      type="com.tutorial.struts.actions.LogoutAction"
      scope="request">
      <forward name="Success" path="/action/login" redirect="true"/>
    </action>

    <action
      path="/viewcarrito"
      parameter="/jsp/carrito.jsp"
      type="org.apache.struts.actions.ForwardAction"
      scope="request"
      validate="false">
    </action>

    <action
      path="/carrito"
      type="com.tutorial.struts.actions.CarritoActions"
      scope="request"
      input="/jsp/carrito.jsp"
      validate="false"
      parameter="method">
      <forward name="Success" path="/jsp/carrito.jsp" redirect="true"/>
    </action>

    <action
      path="/viewdetalleproducto"
      name="detalleProductoForm"
      input="/jsp/index.jsp"
      type="com.tutorial.struts.actions.GetDetalleProductoAction"
      scope="request"
      validate="false">
      <forward name="Success" path="/jsp/detalleproducto.jsp"/>
    </action>

```

```

<action
path="/viewlistaproducto"
input="/jsp/listaproducto.jsp"
type="com.tutorial.struts.actions.GetListaProductoAction"
scope="request"
validate="false">
<forward name="Success" path="/jsp/listaproducto.jsp"/>
</action>

<action
path="/viewperfil"
name="perfilForm"
input="/jsp/index.jsp"
type="com.tutorial.struts.actions.GetPerfilAction"
scope="request"
validate="false">
<forward name="Success" path="/jsp/perfil.jsp"/>
</action>

</action-mappings>

<message-resources parameter="resources.ApplicationResource"/>

<plug-in className="com.tutorial.struts.service.CarritoServiceFactory"/>

</struts-config>

```

Con todo ello creamos un fichero WAR [carrito.war](#) (he incluido también el código fuente) y ya podemos desplegar nuestra aplicación en cualquier servidor de aplicaciones como JBoss o Tomcat (ya que de momento no hemos incluido EJBs). Os incluyo además un [ZIP](#) con la BD que yo he utilizado en MySQL.

Resumen

Si hemos conseguido desplegar la aplicación, podremos comenzar a utilizar nuestra aplicación del carrito de la compra. Algunas pantallas son:

- [Captura 1](#)
- [Captura 2](#)
- [Captura 3](#)
- [Captura 4](#)

Espero que este tutorial os sirva de ayuda, ya que su intención no es ser un manual de Struts, sino iniciar al programador en este entorno de presentación, del que seguro obtendrá muchas ventajas. Si veo que tiene mucha aceptación, os mostraré como añadir EJB de forma inmediata ...

Autor: Enrique Medina Montenegro (c) 2003

Si desea contratar formación, consultoría o desarrollo de piezas a medida puede contactar con



Somos expertos en:
J2EE, C++ , OOP, UML, Vignette, Creatividad ..
 y muchas otras cosas

Nuevo servicio de notificaciones

Si deseas que te enviemos un correo electrónico cuando introduzcamos nuevos tutoriales, inserta tu dirección de correo en el siguiente formulario.

Subscribirse a Novedades	
e-mail	
	Enviar

Otros Tutoriales Recomendados ([También ver todos](#))

Nombre Corto

[Aplicación básica con RMI](#)

Descripción

Gracias a este tutorial, podreis aprender paso a paso como crear una aplicación cliente-servidor con RMI

[Primeras aplicaciones con Bea Weblogic Platform](#)

Os mostramos como instalar Bea Weblogic Platform así como a crear la primera aplicación, con su entorno visual, utilizando la implementación particular basada en Struts....

[Generador automático de Webservices](#)

Os mostramos como crear un servicio Web a partir de una clases, gracias a generadores automáticos de código y NetBeans

[Despliegue gráfico de EJBs](#)

Os mostramos como crear y desplegar de un modo gráfico un EJB de sesión el el servidor de aplicaciones de referencia de Sun

[Cachear porciones de JSPs](#)

En este tutorial os ensañamos como incrementar increíblemente el rendimiento de vuestro Web basado en tecnología JSP con el FrameWork de cache OSCACHE

[JSP 2.0, JSTL y Lenguaje de expresiones](#)

Os mostramos las novedades de JSP 2.0: Nuevas librerías estandar de etiquetas y el lenguaje de expresiones con ejemplos de acceso a base de datos, XML y XSL en JSP

[Struts Jakarta](#)

Cuando se ha trabajado creando aplicaciones Java poco a poco se va viendo la necesidad de normalizar los desarrollo. Uno de los Framework (entornos) más extendidos es Struts

[Pool de Conexiones y Tomcat5](#)

Os mostramos como instalar Tomcat5 en vuestro PC y como ejemplo de uso, configuramos un Pool de Conexiones y lo usamos contra MySQL

[JDBC y MySql](#)

En el tutorial anterior vimos como instalar MySQL en Windows, ahora vamos a ver como acceder desde una aplicación Java.

[Integración de Struts y eclipse](#)

Alejandro Perez nos enseña como construir un entorno de alta eficiencia de desarrollo on Struts a través de plugins de eclipse

[Patrocinados por enredados.com Hosting en Castellano con soporte Java/J2EE](#)



**¿Buscas un hospedaje de calidad
por sólo 2 € al mes?**

www.AdictosAlTrabajo.com Optimizado 800X600