

índice general_

1. INTRODUCCIÓN	1
2. CONTENEDOR DE BEANS. CICLO DE VIDA	27
3. CONFIGURACIÓN Y MANEJO DE PERSISTENCIA CON SPRING JDBC	39
4. SPRING MVC	61
5. EJEMPLO PRÁCTICO	77
 GLOSARIO	127
REFERENCIAS WEB	131
BIBLIOGRAFÍA	133

índice_

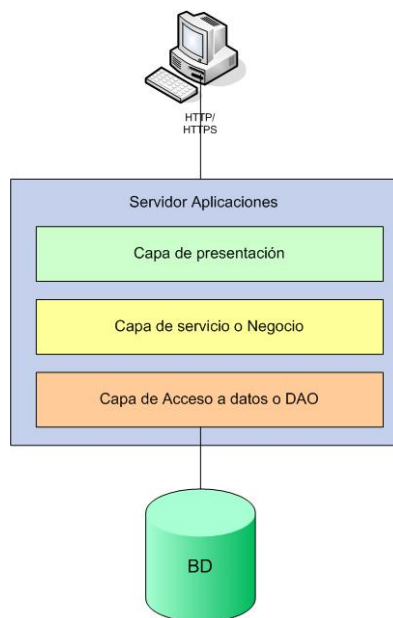
1.1. SPRING FRAMEWORK	3
1.1.1. Concepto de Framework	4
1.1.2. Características del Framework Spring	4
1.2. MÓDULOS DEL FRAMEWORK DE SPRING	5
1.2.1. Spring Core	6
1.2.2. Spring Context	6
1.2.3. Spring DAO	6
1.2.4. Spring AOP	6
1.2.5. Spring ORM	7
1.2.6. Spring Web	7
1.2.7. Spring MVC	7
1.3. FUNCIONALIDADES DE SPRING	7
1.4. PREPARACIÓN DEL ENTORNO DE DESARROLLO	8
1.4.1. Descarga e instalación del IDE Eclipse Web Tools	10
1.4.2. Configuración de los plug-ins a utilizar	10
1.4.2.1. Instalación del plug-in SpringIDE	12
1.4.3. Librerías de Spring	15
1.4.4. Instalación del software necesario	16
1.4.4.1. Instalación del servidor Apache Tomcat	16
1.4.4.2. Instalación de MySQL	21

1.1. SPRING FRAMEWORK

Spring es un framework de desarrollo J2EE que pretende dar soluciones a problemas técnicos innecesariamente complejos. Por esta razón, en los últimos años está siendo ampliamente utilizado por los desarrolladores de aplicaciones J2EE. Además, es fácil de aprender, al basarse en el empleo de POJO's (clases tradicionales de Java), y proporciona los elementos de integración con los frameworks Web y de persistencia más utilizados del mercado.

Como menciona la página web www.springframework.org, Spring es un framework de aplicaciones java/J2EE dividido en capas, basado en el código publicado en "Expert One-on-One J2EE Design and Development" por Rod Johnson (Wrox, 2002).

El framework Spring nos facilita la infraestructura para realizar aplicaciones **multi-capas**. Las aplicaciones, en un principio, se instalaban en el ordenador que las debía ejecutar, desarrollándose, posteriormente, aplicaciones de dos capas (aplicaciones cliente-servidor); no obstante, gracias a la **arquitectura J2EE**, se comenzó a realizar aplicaciones en varias capas relacionadas entre ellas, siendo, en la actualidad, la mayor parte de las aplicaciones que se realizan de **tres capas**. Por este motivo, a lo largo del manual mostraremos el ejemplo de una arquitectura de tres capas.



Estructura de una aplicación de 3 capas

1.1.1. Concepto de Framework

En el mundo de la informática, un **framework** es una estructura de **componentes software** en la cual otros proyectos de software pueden ser organizados y desarrollados. Los frameworks sirven para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

1.1.2. Características del Framework Spring

Spring es un contenedor **ligero**. Normalmente las aplicaciones basadas en EJB´s se dice que usan un contenedor pesado, ya que, es necesario la utilización de servidores de aplicaciones (JBoss, Bea Web Logic, WepSphere Application Server) para desplegar dichas aplicaciones, sin embargo, la utilización de contenedores ligeros permite que las aplicaciones se desplieguen en Servidores de Servlets como Tomcat.

Las empresas exigen cada vez más que el software que se realice sea más **seguro, fiable, escalable, manejable y económico**. Spring nos permite tener un soporte sobre el cual desarrollar nuestras aplicaciones de esta manera, en consecuencia, la elección de este Framework nos dotará de todas estas características.

Además, Spring se caracteriza por:

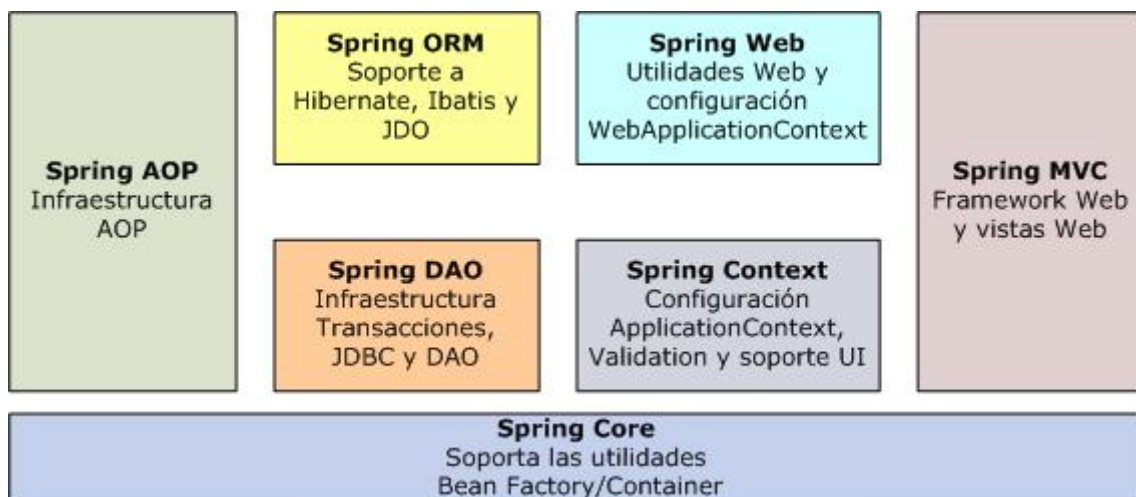
- Proporcionarnos **soluciones sencillas** a problemas usuales.
- **No intrusivo**. La integración entre las diversas capas y los componentes de las mismas se configura de manera declarativa en ficheros XML, y se utiliza mediante el empleo de **IoC (Inversión de Control) o Inyección de Dependencia**, gestionado por la factoría de Beans, como veremos en el próximo tema.
- **Ahorra trabajo**. El desarrollador se centra en lo que es puramente negocio y no en problemas adyacentes.
- **Fácil de usar**.
- Es gratuito. **Código libre**.
- Promueve la **reutilización de componentes** mediante el empleo de interfaces: el **desacoplo entre las diversas capas de la aplicación** promociiona buenas

prácticas (mejora **calidad** y **claridad** del código). Gracias al desacoplo entre las capas podemos acceder a los servicios desde la web, desde web services o desde dispositivos móviles sin modificar el código.

- Como hemos mencionado en el anterior punto, se basa en el empleo de interfaces. Por este motivo, nos permite modificar la clase que la implementa, gracias a lo cual conseguimos el **desacoplo entre tecnologías**. Por ejemplo, declaramos una clase **UsuarioDAO** que la implementa una clase **UsuarioDAOJDBC**, pero si en el futuro decidimos utilizar **Hibernate** como framework de persistencia, la única modificación que deberíamos realizar sería crear una clase que implemente **UsuarioDAO**, llamada **UsuarioDAOHibernate**, y modificar el fichero de configuración normalmente llamado **application-context.xml**, el cual trataremos en más detalle en el próximo capítulo.
- **Sigue evolucionando**, desarrollando nuevas funcionalidades y nuevos componentes para la integración con Web Service, WorkFlow, etc.

1.2. MÓDULOS DEL FRAMEWORK DE SPRING

Spring Framework está compuesto por siete módulos, que a su vez se corresponden con siete librerías independientes, como muestra la siguiente figura:



Como acabamos de mencionar, cada uno de los módulos se corresponde con una librería java independiente. Por este motivo, sólo deberemos utilizar las librerías necesarias para

nuestra aplicación. No obstante, en la mayoría de las aplicaciones utilizaremos sólo la librería **Spring.jar**, la cual contiene a todas las demás.

En los siguientes apartados describiremos cuál es la utilidad de cada uno de los módulos.

1.2.1. Spring Core

El módulo **Core** es la parte fundamental del framework. Proporciona funcionalidades de inyección de dependencias y de gestión como contenedor de beans, así como un patrón de acceso a beans estilo factoría. El contenedor gestiona las instancias de los beans configurados en el framework.

Este módulo lo desarrollaremos con más detalle en el siguiente tema.

1.2.2. Spring Context

Suministra la manera de acceder a los Beans, es decir, las diversas formas para cargar el contexto de la aplicación (**ApplicationContext**), principalmente a través del contenedor de servlets, declarando el contexto de la aplicación en el fichero **web.xml**, también llamado **descriptor de despliegue**.

1.2.3. Spring DAO

Se basa en el **patrón de diseño DAO**, el cual proporciona una capa de abstracción de **JDBC**, gracias a la cual se produce un ahorro de tiempo y un código mucho más sencillo de mantener y eficiente, ya que no es necesario teclear el **código JDBC** para establecer una conexión en todos y cada uno de los accesos a la base de datos.

1.2.4. Spring AOP

Este módulo proporciona servicios empresariales de manera declarativa, especialmente, como reemplazo para los servicios declarativos de **EJB**. El más importante de dichos servicios es el **manejo declarativo de transacciones**, que está construido sobre la abstracción de transacciones de Spring.

Spring proporciona además otros servicios transversales como el envío de correos electrónicos y soporte para subir ficheros al servidor.

1.2.5. Spring ORM

El modulo **ORM** proporciona el soporte para la integración con otros frameworks de mapeo objeto-relacional (object-relational mapping), incluyendo Hibernate e iBatis.

Al utilizar el paquete ORM de Spring, podemos integrar nuestros DAOs con la gestión de transacciones declarativa de Spring.

1.2.6. Spring Web

Provee características básicas de integración orientadas a la web, inicialización de contextos mediante **ServletListeners** y un contexto de aplicación orientado a web.

Mediante el empleo de este módulo podremos integrar otros frameworks MVC más implantados en el mercado como Struts y WebWork.

1.2.7. Spring MVC

El módulo Spring MVC, como su propio nombre indica, nos proporciona una implementación del patrón de diseño MVC (Modelo - Vista – Controlador), en el Capítulo 4, mostraremos como se desarrollan aplicaciones utilizando este módulo.

1.3. FUNCIONALIDADES DE SPRING

- Configuración de aplicaciones con un enfoque sencillo.
- Uso de interfaces como base de la aplicación sin costo adicional.
- Utilización de **JDBC** de forma sencilla.
- Una alternativa sencilla a **EJB**, que permita obtener la mayoría de los servicios del mismo sin su complejidad.
- Conseguir todo lo anterior con clases tradicionales de Java (**POJO's**) y sin dependencia de ninguna API especial.
- Soporta **AOP** (Programación orientada a aspectos).

- Transacciones declarativas **JTA** a clases planas (No requiere **EJB** 's).

1.4. PREPARACIÓN DEL ENTORNO DE DESARROLLO

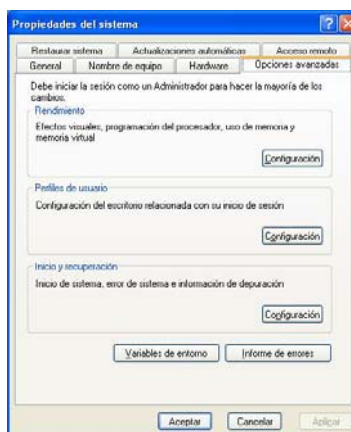
Para poder hacer los ejemplos prácticos que realizaremos a lo largo del manual como el ejercicio final, es necesario que instalemos los siguientes productos de software libre, los cuales se pueden descargar desde Internet:

- **JAVA**. Máquina virtual de Java, junto con las APIs de desarrollo. Recomendamos el empleo de la versión **j2sdk 1.4.2_12**, que se puede descargar gratuitamente desde la web de Sun Microsystems en la siguiente dirección:
<http://java.sun.com/j2se/1.4.2/download.html>

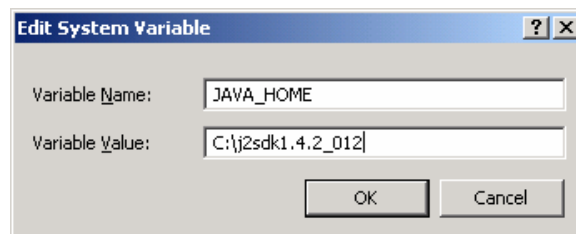
Una vez instalada, no debemos olvidarnos de configurar la variable de entorno **JAVA_HOME** apuntando a la ruta donde hemos instalado el **j2sdk**, y posteriormente le añadiremos esta ruta a la variable de entorno **path**.

Seguidamente, ilustraremos el procedimiento a seguir:

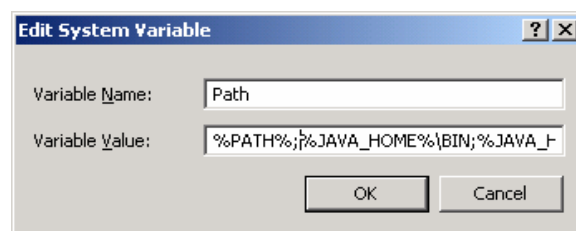
1. Instalamos **j2sdk 1.4.2_12** en el directorio **C:\j2sdk1.4.2_12**, pudiéndose instalar en cualquier otro directorio. Por comodidad indicamos esta ruta, ya que posteriormente nos resultará más fácil configurar la variable de entorno **JAVA_HOME**.
2. Accedemos a las propiedades del sistema pulsando con el botón derecho del ratón sobre el acceso directo de **Mi PC**, y pulsamos en la pestaña de **Opciones avanzadas**. Como resultado nos aparecerá la siguiente pantalla:



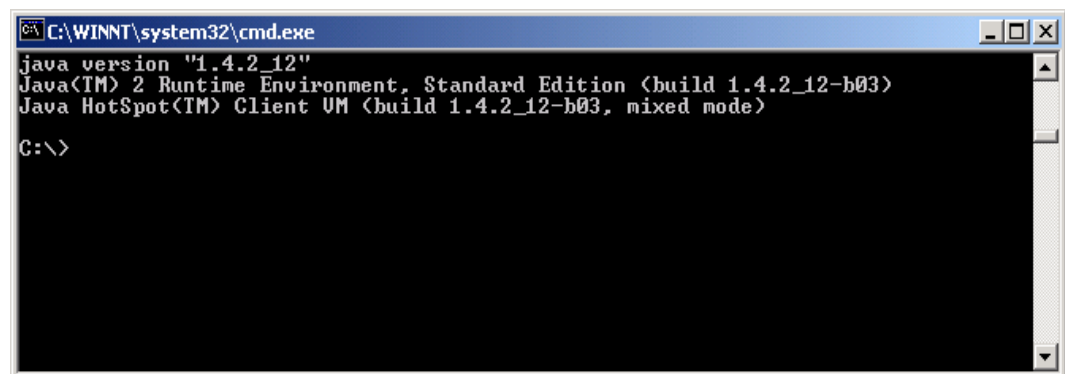
3. Pulsamos sobre el botón **Variables de entorno**.
4. Añadimos la variable **JAVA_HOME** como variable de sistema, como se indica a continuación:



5. Posteriormente, añadimos a la variable **path** la variable **JAVA_HOME** que acabamos de definir.



6. Por último, para comprobar que el proceso ha sido correcto, ejecutaremos en una ventana MS-DOS el comando **java -version**, debiendo aparecer la versión de java configurada como aparece a continuación. En el caso de que no aparezca, volveremos a realizar los pasos anteriores.



1.4.1. Descarga e instalación del IDE Eclipse Web Tools

Una vez configurado el **j2sdk**, instalaremos el entorno de desarrollo integrado de java (**Integrated Development Environment, IDE**).

El **IDE** que hemos elegido es una extensión de la plataforma **Eclipse**, denominada **Eclipse Web Tools Project (WTP)**, el cual proporciona una serie de herramientas para el desarrollo de aplicaciones J2EE Web. Entre las herramientas que ofrece destacamos:

- Editores HTML, CSS, JSP, JavaScript, XML.
- Herramientas para manejo de bases de datos.
- Herramientas para gestionar Servidores Web.
- Soporte para EJB y Web Service

Para poder descargarnos el software, accederemos a la siguiente dirección:
<http://www.eclipse.org/webtools>

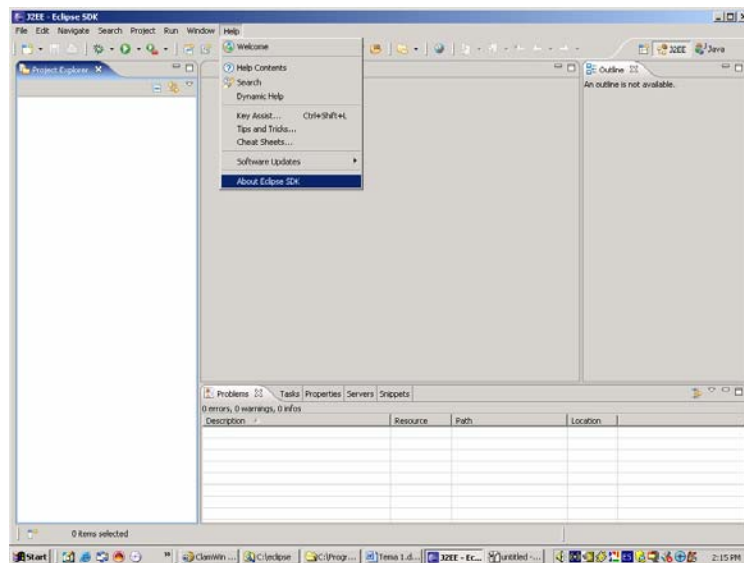
Una vez descargado el software, sólo necesitamos descomprimir el fichero **.zip** en la carpeta que deseemos. Ej. **C:\Eclipse**.

Por último, al ejecutar **Eclipse**, veremos que utiliza la versión **Eclipse 3.1** o superior, ya que Eclipse ha sacado al mercado recientemente la **versión 3.2**.

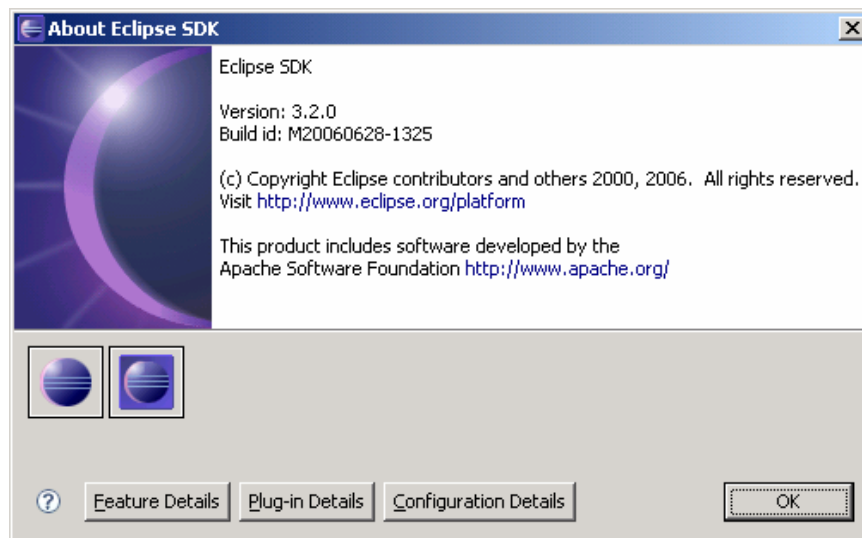
1.4.2. Configuración de los plug-ins a utilizar

Como hemos mencionado en el apartado anterior, en lugar de descargarnos la distribución de **Eclipse**, hemos bajado la distribución **Eclipse Web Tools**, ya que nos proporciona un conjunto de plug-ins que nos ayudarán para el desarrollo de aplicaciones web.

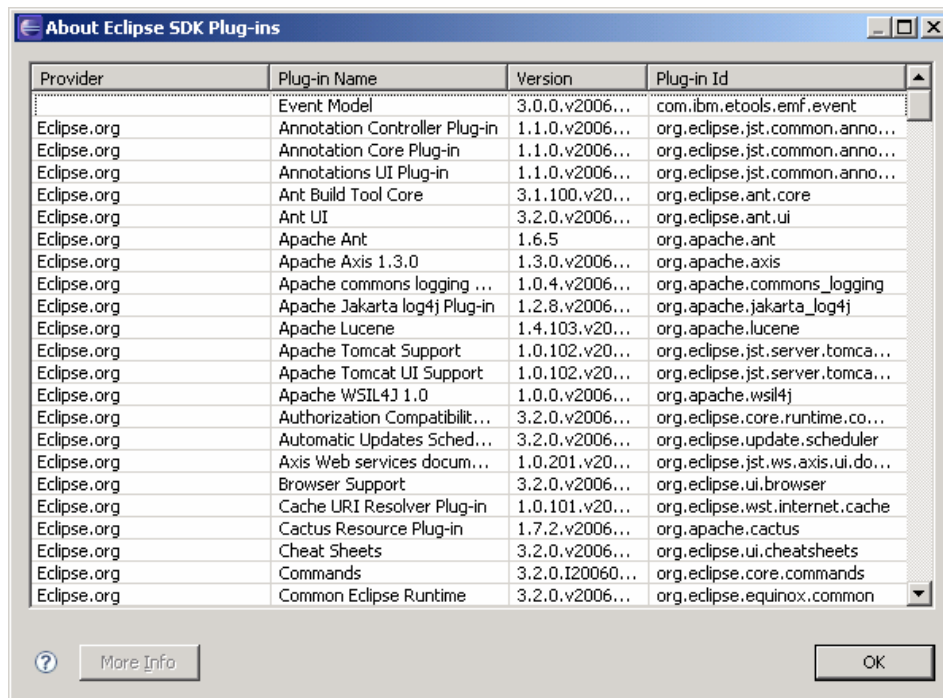
Para poder consultar los plug-ins que aporta la plataforma **Web Tools**, pulsaremos sobre la opción del menú de Eclipse **Help → About Eclipse SDK**, como se muestra en la siguiente imagen.



En dicha pantalla, podemos comprobar que nos aparecen instalados los paquetes relativos a **Eclipse** y **Eclipse Web Tools**.



Por último, si pulsamos sobre **Plug- in Details**, nos aparecerá una pantalla en la que podremos consultar la lista de plug-ins instalados que proporciona la distribución.

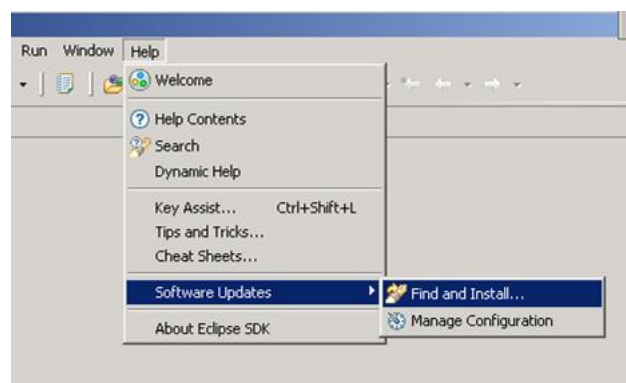


1.4.2.1. Instalación del plug-in SpringIDE

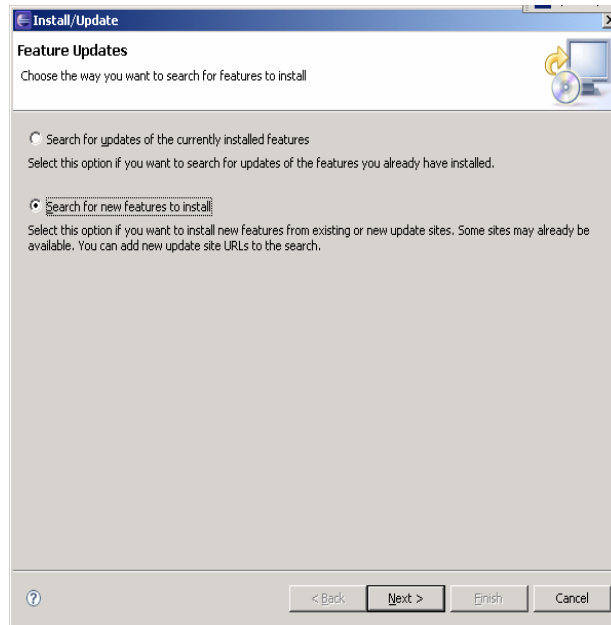
Para conseguir que los proyectos que realicemos en **Eclipse** se integren con **Spring**, el equipo de Spring nos proporciona este plug-in, el cual podemos descargar de: <http://springide.org/updatesite>

La forma de instalar este plug-in es descargar la última versión de la URL anterior y descomprimirla en una carpeta temporal, por ejemplo **C:\SpringIDE**.

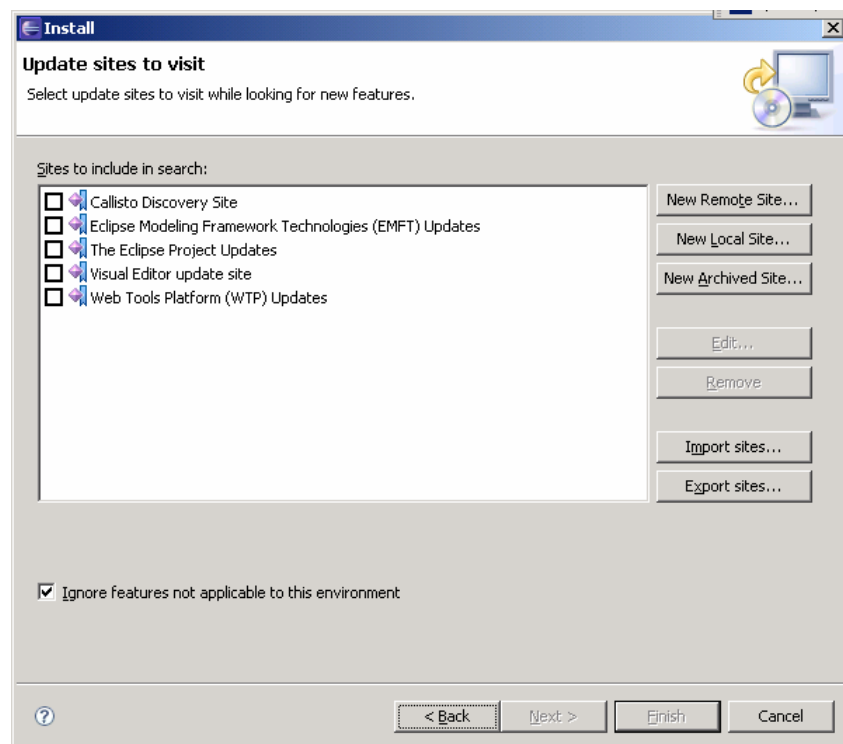
Una vez hecho esto, nos dirigiremos a la barra de menú del eclipse y seleccionaremos las opciones que se muestran en la siguiente ruta **Help→Software Updates→Find and Install**.



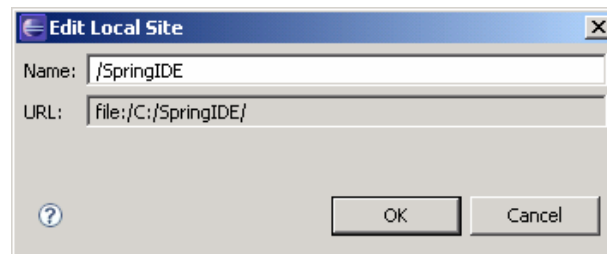
Como resultado, nos aparecerá la siguiente pantalla, en la cual señalaremos que queremos instalar un plug-in nuevo, tal y como muestra la imagen, y pulsaremos **Next**.



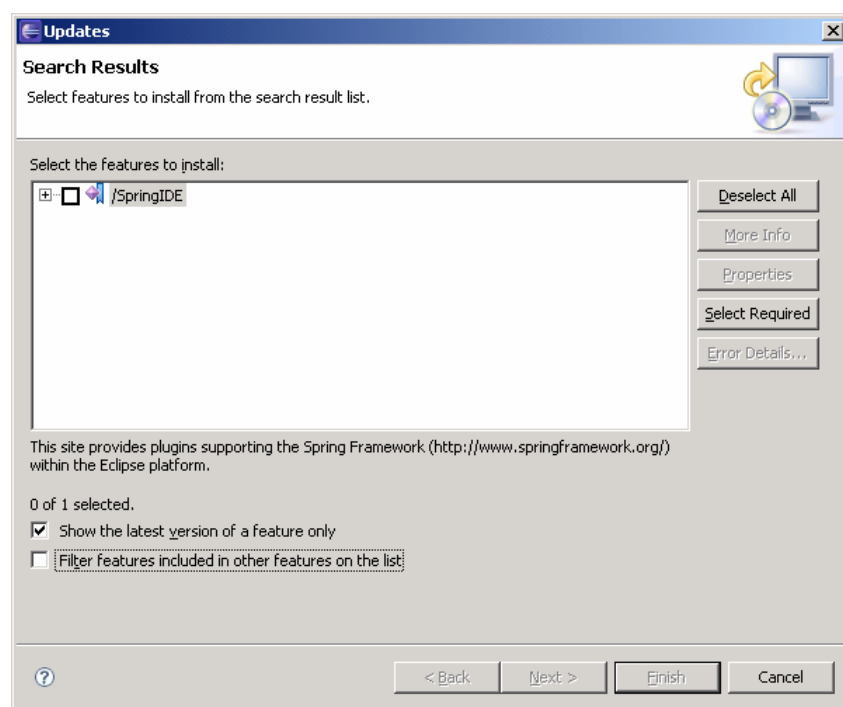
A continuación, aparecerá la siguiente pantalla, donde haremos clic en **New Local Site**.



Después de pulsar dicho botón, aparecerá una pantalla del explorador para indicar dónde se encuentra el plug-in. Tras seleccionarlo, aparecerá la siguiente pantalla:



Seguidamente, se mostrará la pantalla anterior con una nueva opción **/SpringIDE** marcada y pulsaremos **Finalizar**, presentándose la siguiente pantalla, en la que marcamos la opción y pulsamos **Next**.



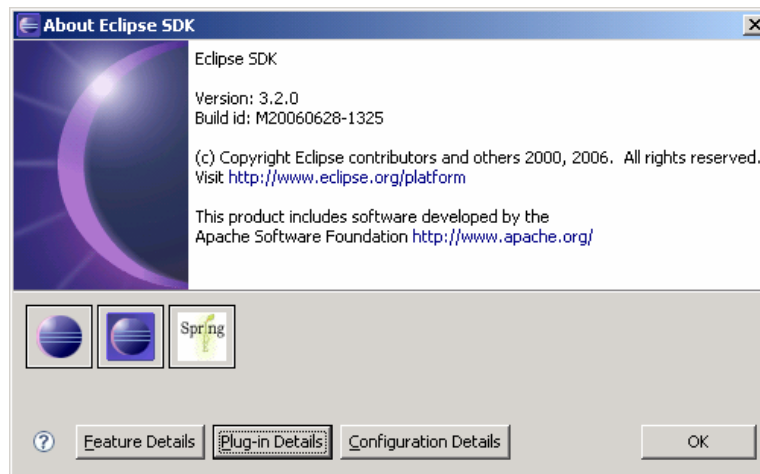
A continuación, nos aparecerá una pantalla de aceptación de la licencia, aceptamos dicha licencia y pulsamos de nuevo **Next**.

Finalmente, se muestra una ventana que indica el plug-in que queremos instalar y pulsamos **Finalizar**.

Cuando realicemos esta última acción, es posible que nos aparezca algún **Warning**. Si ocurre esto, pulsamos en **Instalar** y una vez realizada la instalación, nos aparecerá una

ventana de aviso indicándonos la necesidad de reiniciar, pulsamos **Aceptar** y el plug-in estará correctamente instalado.

Por último, para poder comprobar que se ha instalado correctamente, deberemos pulsar de nuevo sobre la opción del menú de Eclipse **Help → About Eclipse SDK**. Comprobaremos que aparece el icono de **Spring**.



1.4.3. Librerías de Spring

Para bajar las librerías de Spring, descargaremos desde la página de descargas la última versión de producción del software de Spring, la **versión 1.2.8**, eligiendo la opción con dependencias: <http://prdownloads.sourceforge.net/springframework/spring-framework-1.2.8-with-dependencies.zip?download>

Una vez descargado, lo descomprimiremos en nuestro ordenador y observaremos la estructura de directorios que genera. A continuación, comentaremos la información de las carpetas que componen esta distribución salvo las de testeo:

1. **dist**: librerías que componen el framework Spring.
2. **docs**: documentación de Spring, el API de Spring, la guía de referencia, aplicación del patrón MVC que proporciona Spring paso a paso, al igual que el API de las librerías de **tags** que proporciona Spring MVC.
3. **lib**: conjunto de librerías ajenas a Spring que se utilizarán en los ejemplos.

4. **samples:** contiene algunos ejemplos en los que se emplean los distintos módulos de Spring.
5. **src:** código fuente de todos los módulos de Spring.

1.4.4. Instalación del software necesario

Una vez configurado el entorno de desarrollo base, instalaremos, como material de apoyo, una base de datos y un servidor web, para poder realizar los ejemplos que se desarrollaremos a lo largo del manual.

1.4.4.1. Instalación del servidor Apache Tomcat

Para poder desplegar las aplicaciones que realicemos en el transcurso del manual, necesitaremos un servidor de aplicaciones web. No obstante, como en los ejemplos que vamos a realizar no intervienen Enterprise Java Beans, ni web Services, utilizaremos el contenedor de Servlets y JSP Tomcat 5.0, que encontraremos en la siguiente dirección:

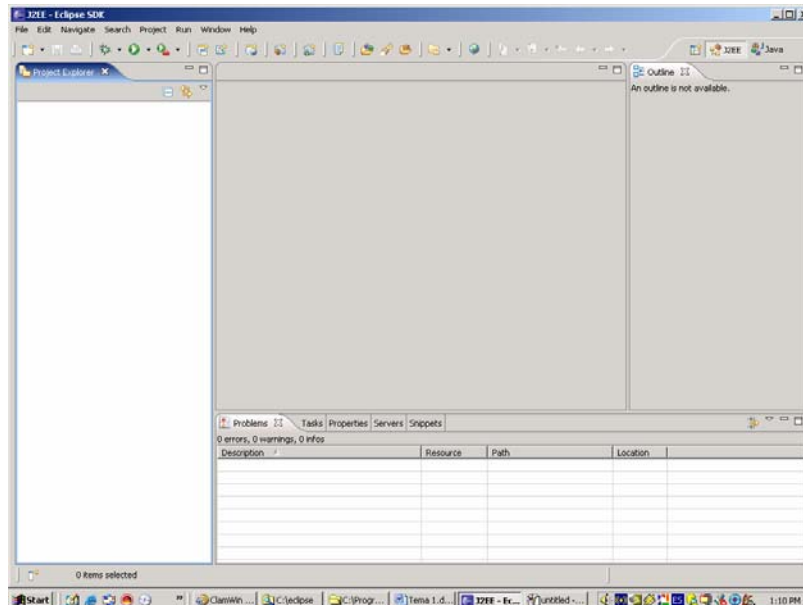
<http://mirrors.hostalia.com/apache/tomcat/tomcat-5/v5.0.28/bin/jakarta-tomcat-5.0.28.exe>

Una vez instalado, lo arrancamos y ejecutamos desde nuestro navegador <http://localhost:8080>. Seguidamente, observamos que aparece la siguiente pantalla.

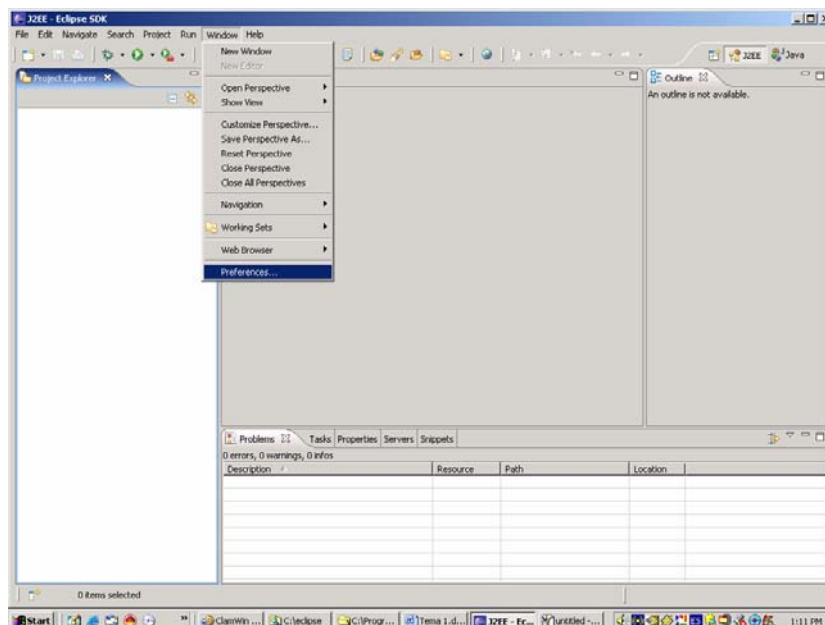


A continuación, mostraremos los pasos a seguir para configurar un servidor en Eclipse.

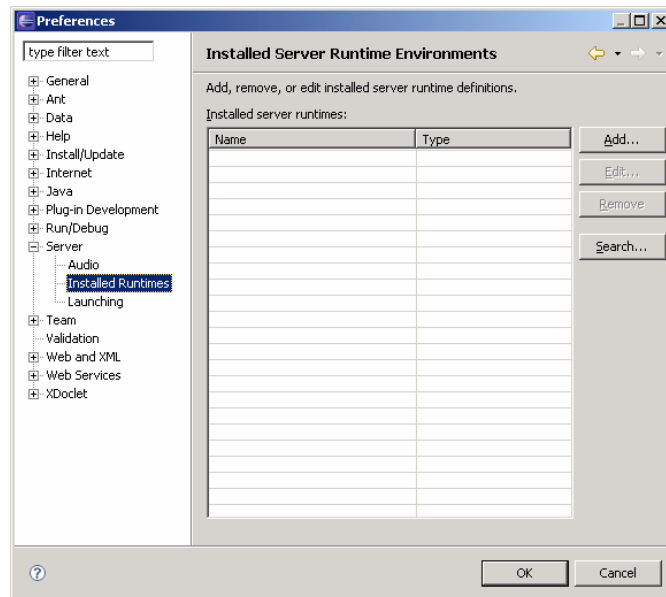
1. Ejecutamos el eclipse.



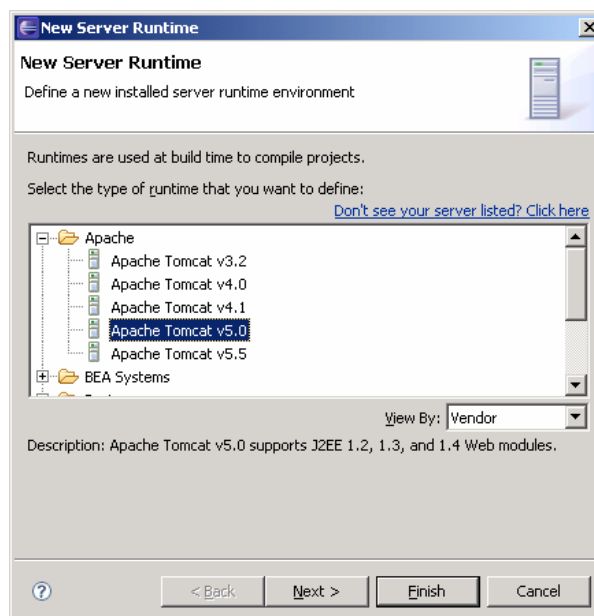
2. Desde la barra de herramientas de Eclipse, seleccionamos la opción **Window→Preferences...**



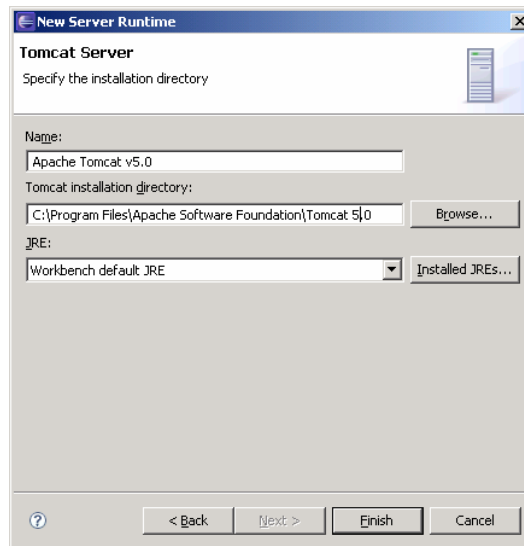
3. En la lista de la izquierda, seleccionamos: **Server→Installed Runtimes**.



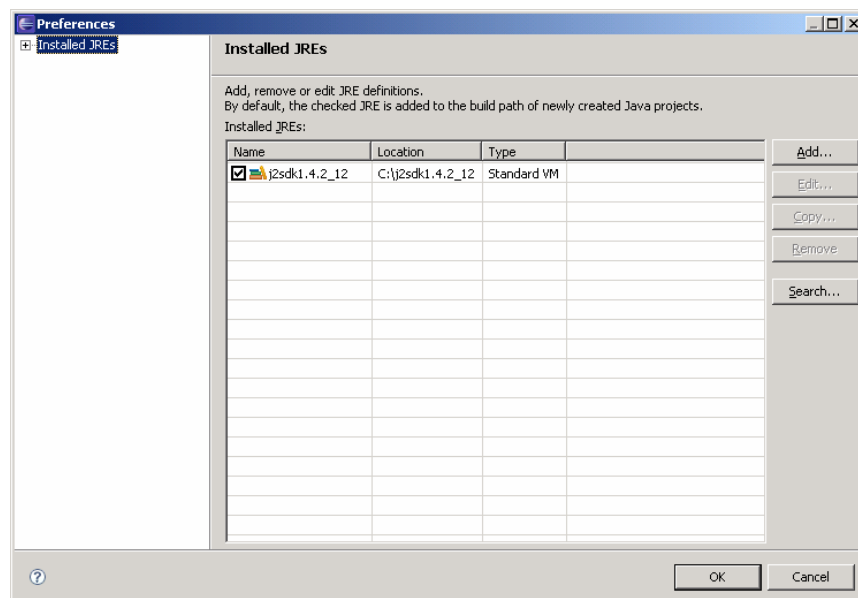
4. Como la primera vez que accedemos no hay ningún servidor configurado, debemos añadir uno. Para realizar esta operación, pulsamos **Add**, seleccionamos el servidor que hemos instalado y pulsamos **Next**.



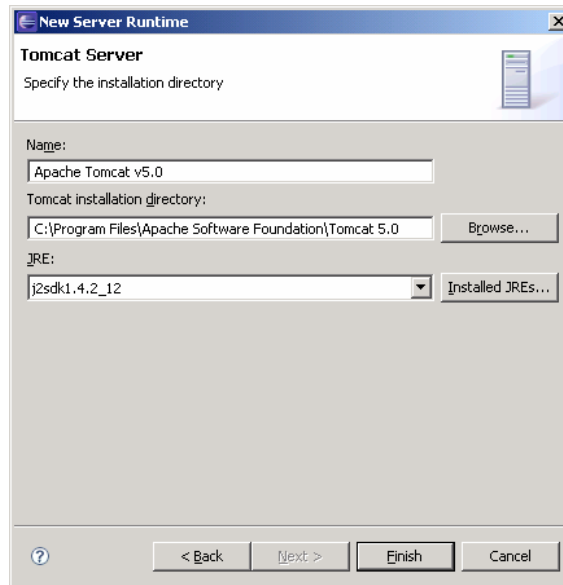
5. Seguidamente, se muestra la siguiente ventana que es donde debemos especificar la ruta donde hemos instalado nuestro servidor. Para poder establecer dicha ruta pulsaremos sobre **Browse**.



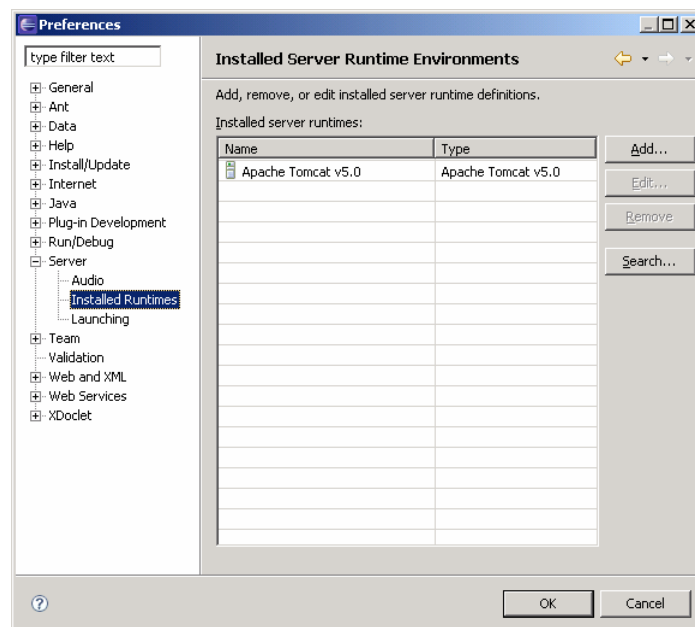
6. Como el **JRE** (Java Runtime Environment) asociado al servidor no es el que hemos instalado, pulsamos sobre la opción **Installed JREs**. Nos aparecerá la siguiente pantalla donde seleccionaremos el **JRE** que hemos instalado y pulsaremos **OK**.




- Una vez realizado esto, nos devolverá a la pantalla anterior y deberemos seleccionar el JRE.



- Por último, pulsamos **Finish** y surge la pantalla con los servidores configurados. Pulsamos **Ok** para finalizar la instalación.



Una vez instalado el servidor, para comprobar que se ha realizado correctamente, desde la pantalla de **Eclipse**, pulsamos sobre cualquiera de las siguientes opciones  de

Eclipse para arrancar el **Tomcat**. El primer botón indica que arrancaremos Tomcat en modo depuración, mientras que el segundo lo hará en modo normal.

1.4.4.2. Instalación de MySQL

A lo largo del manual, mostraremos como configurar en **Spring** la conexión con una base de datos a través de **JDBC** y, finalmente, en la práctica de ejemplo, realizaremos la gestión de una librería. Como dicha información será almacenada en una base de datos, será necesaria la instalación de una.

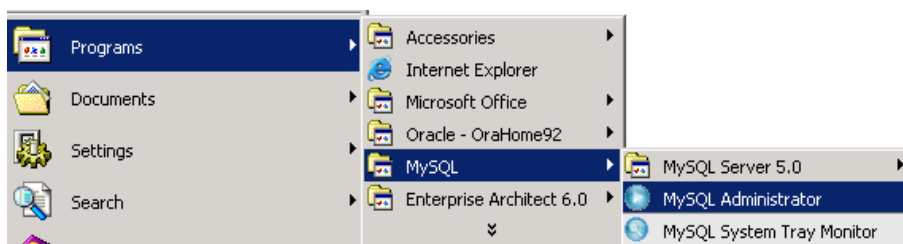
En nuestro caso, hemos elegido la utilización de MySQL, como gestor de base de datos, al tratarse de un software de código abierto y ser uno de los gestores de bases de datos más utilizado en el mercado.

Descargaremos el instalador de la base de datos desde la siguiente dirección: <http://dev.mysql.com/downloads/>

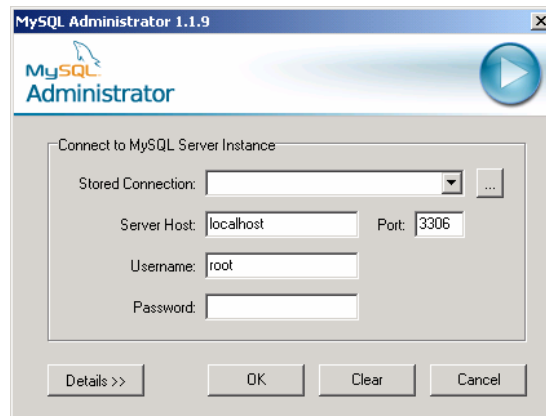
En dicho link, también encontraremos la documentación de referencia de las distintas versiones de base de datos, así como otros productos que nos facilitarán el trabajo, tales como **MySQL Administrator**, **MySQL Tools** y los **Drivers** para conectarnos desde nuestra aplicación.


En primer lugar, comenzaremos por descargarnos el instalador de la base de datos, en nuestro caso hemos escogido la versión 5.0 por ser la última versión estable del producto. Seguidamente, seleccionaremos todos los valores por defecto y, posteriormente, procederemos de la misma forma con el software **MySQL Administrator**, el cual, como su propio nombre indica, nos ayudará a administrar la base de datos instalada, para la creación de usuarios, de esquemas o catálogos y consultar el **log** principalmente.

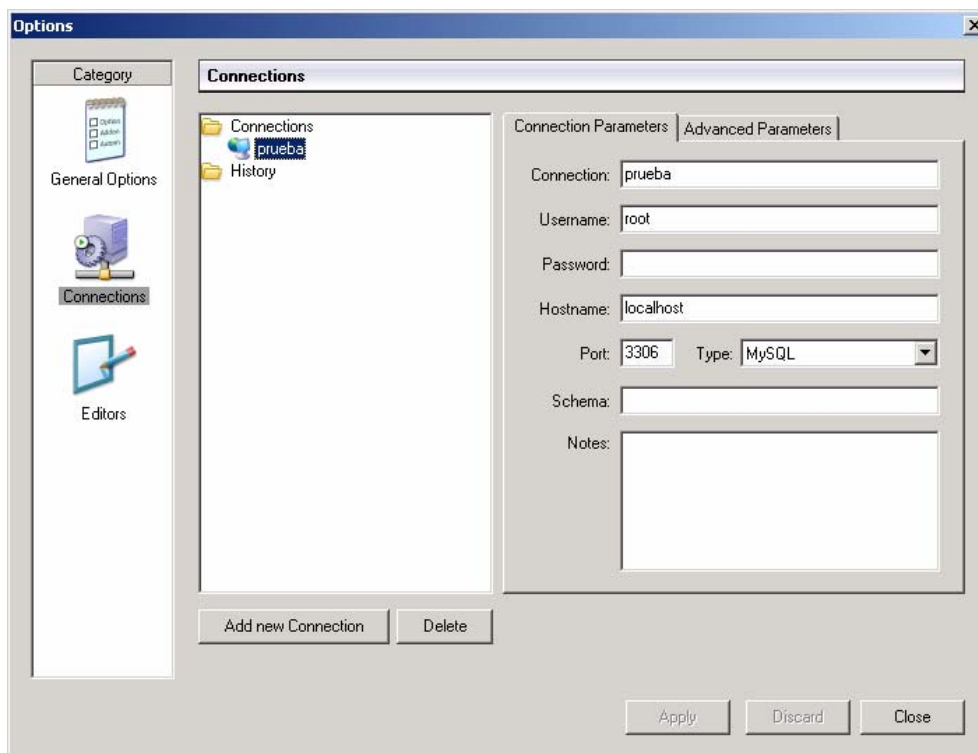
Para realizar estas operaciones debemos configurar una conexión, siguiendo los siguientes pasos:



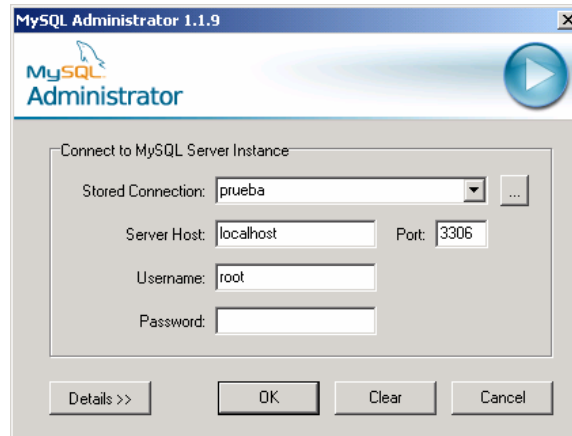
1. **Arrancar el programa MySQL Administrador** desde el menú de inicio:



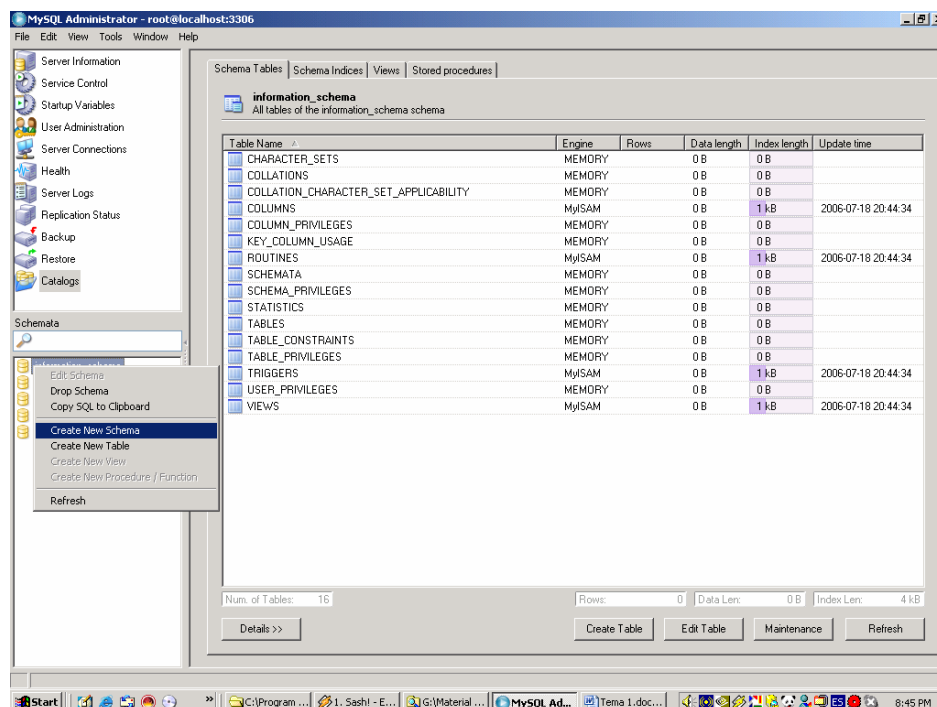
2. **Crear una conexión.** Para ello, pulsamos el botón  y realizamos una nueva conexión, indicando los parámetros de ésta, en nuestro caso con los datos por defecto al instalar MySQL, como se muestra a continuación:



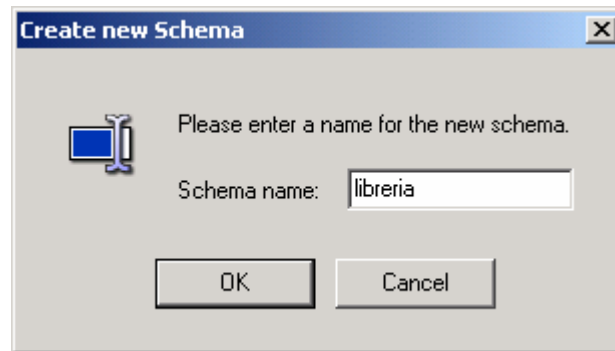
- Una vez creada la conexión, pulsamos **OK**.



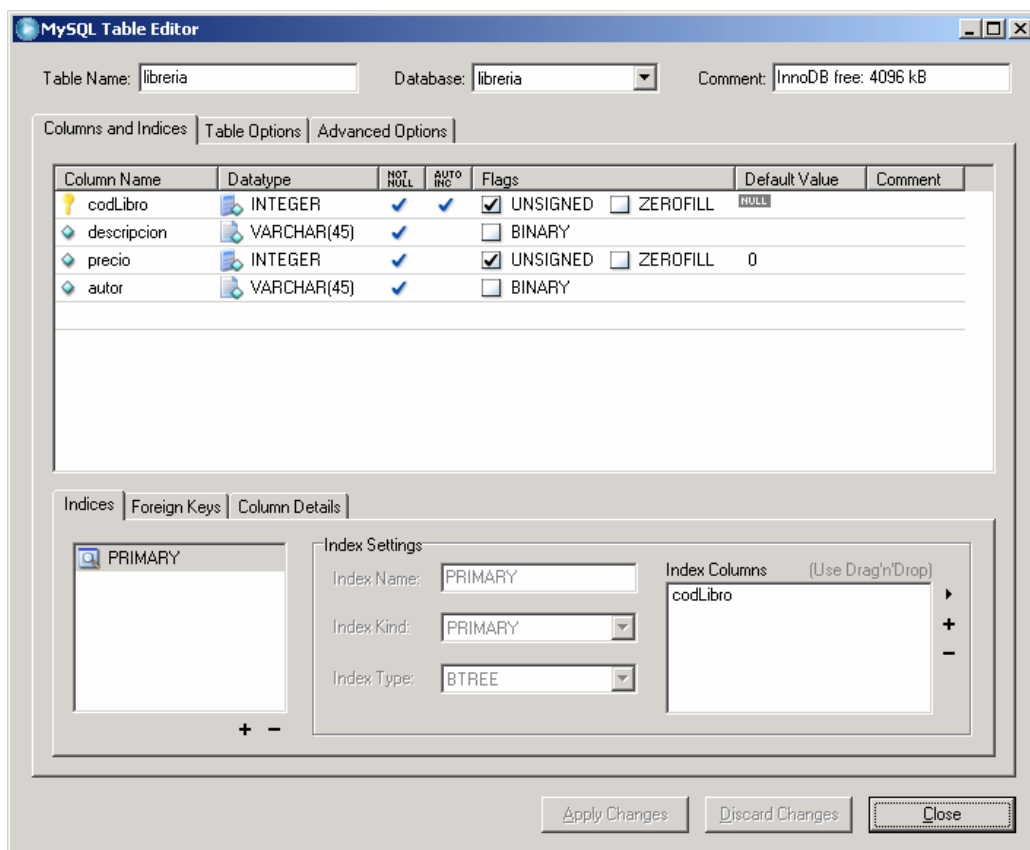
- En este momento, nos aparece la pantalla principal desde la cual administraremos nuestra base de datos, seleccionaremos la opción **Catalogs** del menú superior izquierdo y, en la parte inferior, haremos clic con el botón derecho en **Create New Schema**, para crear un nuevo esquema en la base de datos.



- Seguidamente, introducimos el nombre del esquema. Para la aplicación que realizaremos al tratarse de una librería, hemos seleccionado **libreria** como nombre para nuestro esquema.



- Tras crear el esquema, establecemos la tabla que utilizaremos para ilustrar nuestro ejemplo, la cual también hemos llamado **libreria**.



- **Spring** es un framework ligero.
- Spring se basa en el empleo de **POJO** ¹s (clases tradicionales de Java) y de interfaces.
- Spring es un framework compuesto por un conjunto de **módulos**.
- Spring proporciona **aplicaciones seguras, fiables, escalables, manejables y económicas**.

¹ A lo largo del manual podemos también utilizar la palabra BEANS para referirnos a los POJO ¹s

2.1. INTRODUCCIÓN.....	29
2.2. DEFINICIÓN DE DEPENDENCIA DE INYECCIÓN.....	29
2.3. LA FACTORÍA DE BEANS Y EL CONTEXTO DE LA APLICACIÓN	31
2.3.1. BeanFactory	32
2.3.2. ApplicationContext	33
2.3.2.1. Inicializar el contenedor de Beans	35
2.3.3. Definición de Bean.....	36

2.1. INTRODUCCIÓN

En este tema, nos vamos a centrar en los componentes de los dos módulos principales que constituyen el núcleo central de Spring (**Spring Core** y **Spring Context**), de los cuales ya realizamos una breve introducción en el primer capítulo.

Además, haremos especial hincapié en la utilización del patrón de diseño **Inversión de Control (IoC)**, y, más concretamente, **Dependencia de Inyección (ID)**, patrón en el que está basado la lógica de Spring.

2.2. DEFINICIÓN DE DEPENDENCIA DE INYECCIÓN

El concepto de **Inversión de Control (IoC)** es un **patrón** de diseño que permite la escritura de un código mucho más limpio y sencillo, facilitando el desacoplamiento entre los componentes de la arquitectura y la definición declarativa (frente a la definición programática) de las relaciones entre ellos.

El **modelo IoC de Dependencia de Inyección** se basa en el **principio de Hollywood** – “**No me llames, ya te llamo yo**” –. Es decir, las clases de una aplicación no necesitan buscar e instanciar las clases de las que dependen. El control se invierte y es un componente contenedor el que asume la responsabilidad de la definición de dichas dependencias y la gestión de las mismas en tiempo de ejecución.

La utilización de **IoC** permite la escritura de código mucho más limpio y sencillo, y facilita el desacoplamiento entre los componentes de la aplicación.

Por ejemplo, el siguiente código es una **Action de Struts** que tiene una dependencia de la clase **LibreriaService**.

```
public class ObtenerListaLibrosAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        String forward = "ok";

        LibreriaService libreriaService = SvcFactory.libreriaService();
```

```
Collection listaLibros=(Collection)libreriaService.getLibros();

request.setAttribute("listaLibros",listaLibros);
return mapping.findForward(forward);

}
}
```

El código tiene que ser “consciente” de esta dependencia y debe localizar e instanciar un objeto de la clase **LibreriaService**.

Por otra parte, la siguiente versión del mismo código implementa el **modelo IoC**.

```
public class ObtenerListaLibrosAction extends Action {
    /** Logger for this class and subclasses */

    protected final Log logger = LogFactory.getLog(getClass());
    //Dependencia de Inyección utilizando inyección de setter, el método más común
    //utilizado por los desarrolladores java
    private LibreriaService libreriaService;

    public void setLibreriaService(LibreriaService libreriaService) {
        this.libreriaService = libreriaService;
    }

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        String forward = "ok";
        Collection listaLibros=(Collection)libreriaService.getLibros();

        request.setAttribute("listaLibros",listaLibros);
        return mapping.findForward(forward);
    }
}
```


A diferencia del ejemplo anterior, éste no contiene ningún tipo de búsqueda en la **Factoría**, ni instanciación del objeto **LibreriaService**. Esto lo hace más claro y, debido a que el código no contiene dependencias, más sencillo de probar.

Spring se encarga, de forma declarativa, de gestionar las dependencias entre los distintos componentes y asegurar la disponibilidad de los objetos cuando sean necesarios.

Asimismo, Spring nos facilita varios tipos **Dependencia de Inyección**. A continuación, mostramos los más utilizados:

- **Inyección de Interfaces.** Esta forma de Dependencia de Inyección es poco utilizada por los desarrolladores en Spring. En ella, el contenedor es el encargado de implementar los métodos relativos a esas interfaces en tiempo de ejecución.
- **Inyección de setter.** La Dependencia de Inyección se consigue mediante los métodos setter de los JavaBeans. Es el método más utilizado por los desarrolladores de Spring.

```
private LibreriaService libreriaService;  
  
public void setLibreriaService(LibreriaService libreriaService) {  
    this.libreriaService = libreriaService;  
}
```

Ej. Inyección setter

- **Inyección en el Constructor.** La Dependencia de Inyección se produce al pasarle las clases que utiliza como parámetros al constructor.

2.3. LA FACTORÍA DE BEANS Y EL CONTEXTO DE LA APLICACIÓN

Como adelantamos en el anterior tema, al explicar los diversos módulos que componen Spring:

- **Spring Core** es el módulo principal de Spring (en algunos libros lo podemos encontrar como el **núcleo** o el **corazón** de Spring). Por este motivo, debemos entender que este módulo lo componen los elementos esenciales de Spring.

Dicho módulo proporciona:

- Funcionalidades de **Dependencia de Inyección**, mediante la implementación del patrón de diseño **Inversión de Control**, y las de gestión como contenedor de beans.
- Un patrón de acceso a beans estilo factoría en lugar de utilizar el patrón **ServiceLocator** (localizador de servicios).

El contenedor es el encargado de gestionar las instancias de las clases configuradas en el framework.

- **Spring Context**. Este módulo nos proporciona una capa superior al módulo **Core**, que nos hace mucho más amigable el trabajo.

2.3.1. BeanFactory

Hay dos maneras mediante las cuales los clientes pueden utilizar las funcionalidades que proporciona el Framework de Spring:

- La factoría de beans **BeanFactory**. Es una representación Java del contenedor de beans utilizado por Spring, la cual aloja la información de todos los beans de Spring y permite al usuario instanciar dichos beans y utilizarlos.
- Utilizar el contexto a través del **ApplicationContext**. Normalmente, los desarrolladores no operan con la clase anterior, sino que la factoría de Beans está encapsulada en el **ApplicationContext** del módulo **Context**.

El **ApplicationContext** se encuentra por encima del **BeanFactory** y hereda de ella todas las funcionalidades de ésta.

La factoría de Beans es en realidad quien nos **instancia, configura y maneja los beans**.

Por último, aunque, como ya hemos mencionado, los desarrolladores de **Spring** rara vez utilizan directamente la **Factoría de Beans**, el equipo que ha desarrollado Spring nos proporciona una jerarquía de interfaces para crear nuestra propia **Factoría de Beans**.

A continuación, mostramos algunas de las interfaces que proporciona **Spring**:

- **BeanFactory**
- **HierarchicalBeanFactory**
- **ListableBeanFactory**
- **AutowireCapableBeanFactory**
- **ConfigurableBeanFactory**
- **XMLBeanFactory**

2.3.2. ApplicationContext

Como ya hemos resaltado en el apartado anterior, **ApplicationContext** es una **extensión** más amigable de **BeanFactory**.

El **ApplicationContext** sólo se instancia una vez y es usada por los programadores para:

- **Obtener las instancias de los beans.** La forma más sencilla de hacerlo es obtener una instancia de un Bean que se encuentra definida en el fichero **ApplicationContext**, mediante **IoC** como se muestra a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/
spring-beans.dtd">
<!-- - Application context definition for "springapp" DispatcherServlet. -->
<beans>
  <bean id="controladorSpring"
class="com.formacion.tienda.web.ControladorSpring">
    <property name="libreriaService">
      <ref bean="libreriaSvc"/>
    </property>
  </bean>
</beans>
```

- **Localizar definición de beans.** Por ejemplo, podemos encontrar la definición de un bean que utilizaremos para crear el objeto **DataSource**, el cual comentaremos con más detalle en el tema de persistencia de datos.

```
<!-- ===== DataSource ===== -->
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
        <value>jdbc:mysql://localhost:3306/libreria</value>
    </property>
    <property name="username">
        <value>root</value>
    </property>
    <property name="password">
        <value></value>
    </property>
</bean>
```

- **Acceder a mensajes internacionalizados estilo I18N.** A continuación, mostramos cómo configurar nuestro fichero para conseguir la internacionalización de la aplicación.

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename"><value>resourceBundle</value></property>
</bean>
```

- **Cargar múltiples contextos.** La aplicación que estamos desarrollando tiene una funcionalidad bastante limitada, no obstante, debemos tener en cuenta que en las aplicaciones que nos encontramos en el mundo real, se puede definir una cantidad de bean muy elevada. Debido a ello, Spring nos permite cargar múltiples contextos, cada uno de los cuales se encargará de una funcionalidad definida. Así, por ejemplo:
 - Los beans relativos a seguridad se suelen alojar en un fichero: **application-context-seguridad.xml**.
 - Los beans de persistencia se suelen alojar en un fichero: **application-context-persistencia.xml**.

- **Propagar eventos** para beans en los cuales implementamos la interfaz **ApplicationListener**.

2.3.2.1. Inicializar el Contenedor de Beans

Spring nos proporciona diversas formas de inicializar el contenedor de Beans. Así, como ya mencionamos, podemos realizar la inicialización del Contenedor implementando cualquiera de las interfaces **BeanFactory**.

Normalmente esto no lo suelen realizar los desarrolladores, ya que Spring nos facilita otros mecanismos para inicializar el contexto, utilizando cualquiera de las siguientes maneras:

- **ClassPathXmlApplicationContext**. Carga el fichero de contexto desde el **classpath** de la aplicación Web, es decir, **WEB-INF/classes** o **WEB-INF/lib**.

A continuación, mostramos un ejemplo de inicialización, indicando que el fichero de configuración del contexto se encuentra en: **WEB-INF/classes/com/prueba/applicationContext.xml**

```
ApplicationContext appContext =  
new ClassPathXmlApplicationContext("com/prueba/applicationContext.xml");  
BeanFactory factory = (BeanFactory) appContext;
```

En el caso de tener varios ficheros de configuración, deberíamos modificar el anterior fragmento de código para cargar todos los ficheros de configuración, como se muestra seguidamente:

```
String[] paths = {"/WEB-INF/applicationContext*.xml"};  
ApplicationContext appContext =  
New ClassPathXmlApplicationContext(paths);  
BeanFactory factory = (BeanFactory) appContext;
```

- **FileSystemXmlApplicationContext**. Carga el fichero de contexto desde un fichero del sistema.

```
ApplicationContext appContext =  
new FileSystemXmlApplicationContext (path);  
BeanFactory factory = (BeanFactory) appContext;
```

Donde **path** puede ser un **String** que representa la ruta absoluta o relativa donde encontramos los ficheros de configuración.

- **XmlWebApplicationContext**. Carga el fichero de contexto inicializado por el **ContextLoaderListener**. Para cargar el contexto de esta forma, necesitamos configurar un **Listener** en el fichero **web.xml**. El **Listener** inicializaría el contexto de la aplicación al arrancar la instancia del servidor de Aplicaciones.

- Fragmento del fichero **web.xml**:

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

2.3.3. Definición de Bean

En Spring, un **bean** es cualquier clase instanciable, es decir no debe ser un Java Bean puro. Por **JavaBean Puro** entendemos aquél que sigue las siguientes convenciones:

- La clase debe ser **serializable**, es decir, capaz de salvar persistentemente y de restablecer su estado.
- Debe tener un **constructor sin argumentos**.
- Sus **propiedades** deben ser **accesibles** mediante métodos **get** y **set** que siguen una convención de nomenclatura estándar.
- Debe contener determinados **métodos de manejo de eventos**.

Para definir un bean en Spring, simplemente debemos declararlo en el fichero o los ficheros (en el caso de aplicaciones complejas que se declaren muchos beans, se suele separar por funcionalidades o capas) de configuración de Spring (normalmente, **application-context.xml**), aunque para aplicaciones que utilizan **Spring MVC**, se suele utilizar el nombre de la aplicación – **servlet.xml**.

Para definir un bean, debemos proporcionar toda la información necesaria para que la factoría sea capaz de instanciar dicho bean. La definición de un bean, en ocasiones, contiene una dependencia con otros beans. Por ejemplo, en la definición de un bean que representa el controlador de Spring que veremos en el tema 4 de este manual, podemos comprobar que

depende de otro bean llamado **libreríaService** que hace referencia a un bean llamado **libreríaSvc**.

```
<bean id="controladorSpring" class="com.formacion.tienda.web.ControladorSpring">  
    <property name="libreriaService">  
        <ref bean="libreriaSvc"/>  
    </property>  
</bean>
```

Por último, también es necesario decir que para definir correctamente los Beans, Spring nos proporciona un fichero llamado **spring-beans.dtd**, el cual nos indica todos los atributos y las reglas necesarias para que el fichero esté bien generado.

- **Dependencia de Inyección** es la implementación del patrón de diseño **Inversión de Control (IoC)**, el cual se basa en el **principio de Hollywood**. “No me llames, ya te llamo yo”.
- Spring nos proporciona tres **formas de Dependencia de Inyección**:
 - **Inyección de Interfaces.**
 - **Inyección de setter.**
 - **Inyección en el Constructor.**
- Spring nos permite obtener las **instancias** a los objetos, de dos formas diferentes:
 - Implementando una **Factoría de beans** por medio de alguna de las interfaces expuestas en el punto 2.3.1 de este capítulo.
 - Utilizando el fichero que nos proporciona **Spring** para definir los beans y cargándolos en el **Contexto**, capa por encima de la factoría.
- **Bean** en Spring es toda clase que se desee, no debe cumplir los estándares para los JavaBeans.

3.1. INTRODUCCIÓN.....	41
3.2. PATRÓN DE DISEÑO DAO.....	41
3.3. PAQUETES QUE PROPORCIONA SPRING PARA EL SOPORTE JDBC	42
3.4. DATASOURCE	44
3.4.1. Mapeo de un DataSource utilizando Spring	46
3.4.2. Mapeo de un DataSource utilizando Spring y JNDI	47
3.4.3. Mapeo de un DataSource utilizando el paquete Apache Commons dbcp.....	47
3.4.4. Mapeo de un DataSource mediante el empleo fichero .properties	48
3.5. UTILIZACIÓN DE JDBC CON SPRING	51
3.5.1. Operaciones utilizando JdbcTemplate	55
3.5.2. Utilización de la clase JdbcDaoSupport	56

3.1. INTRODUCCIÓN

En el presente capítulo mostraremos como Spring proporciona soporte para trabajar con las distintas bases de datos mediante JDBC.

Comenzaremos por nombrar el patrón de diseño **DAO** y, una vez entendamos en qué se basa dicho patrón, veremos como Spring lo utiliza.

Por último, mostraremos cómo configurar un **DataSource** en Spring, y cómo utilizar las clases que proporciona Spring, **JdbcTemplate** y **JdbcDaoSupport**, para operar con la base de datos.

3.2. PATRÓN DE DISEÑO DAO

El **patrón DAO (Data Access Object)**, es el patrón de diseño más utilizado por los desarrolladores para separar la capa de persistencia y la lógica de negocio, ya que está demostrado que aporta unas mejoras en cuanto a claridad del código generado y reutilización de los componentes.

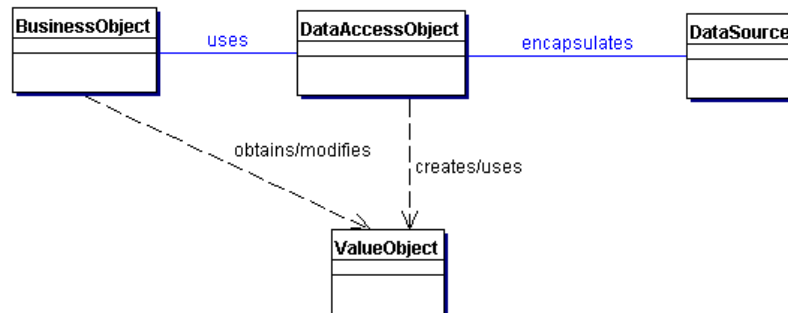
El **patrón DAO** oculta completamente los detalles de la implementación de las estructuras que almacenan los datos de nuestra aplicación, ya se encuentren dichos datos en una base de datos, en ficheros o en cualquier otro soporte que empleemos para este fin.

Podremos consultar más información sobre este patrón de diseño y muchos otros en la página oficial de SUN:

<http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>

3 Configuración y manejo de persistencia con Spring JDBC

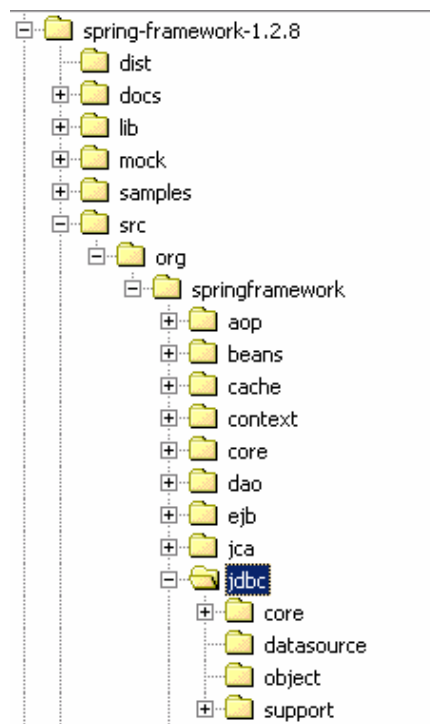
La siguiente figura muestra el **diagrama de clases** que representa las relaciones para el **patrón DAO**:



3.3. PAQUETES QUE PROPORCIONA SPRING PARA EL SOPORTE JDBC

El **paquete JDBC**, que proporciona la distribución de Spring, se divide en los siguientes cuatro paquetes: **core**, **datasource**, **object** y **support**.

En el explorador de Windows puede verse la organización de paquetes de la distribución de Spring, en la que hemos resaltado el **paquete jdbc**.



Estructura de la distribución de Spring

3 Configuración y manejo de persistencia con Spring JDBC

A continuación, mostraremos la **funcionalidad** de cada uno de estos paquetes:

- **org.springframework.jdbc.core**. Son las clases centrales de Spring ODBC. Éstas son:
 - **JdbcTemplate**
 - **JdbcDaoSupport**

En los apartados finales de este tema mostraremos en qué se basan y cuál es su empleo.

- **org.springframework.jdbc.datasource**. Contiene tanto un conjunto de clases que nos facilitarán el acceso a los **datasource**, como implementaciones básicas de **datasource**.
- **org.springframework.jdbc.object**. Son las clases que nos proporcionan el soporte para realizar consultas sobre la base de datos, operaciones de modificaciones y ejecución de procedimientos almacenados.
- **org.springframework.jdbc.support**. Se encarga del tratamiento de las excepciones de tipo **SQLException** y proporciona soporte para las clases de los paquetes **jdbc.core** y **jdbc.object**.

Las clases de este paquete nos facilitan el uso de JDBC y reducen los errores más comunes al utilizar JDBC, ya que:

- Nos simplifica el manejo de errores. Al estar encapsulado, no es necesario añadir bloques de excepciones tipo **try/catch/finally** en el código de las aplicaciones, principalmente para cerrar las conexiones, ya que Spring lo hace por nosotros.
- Presenta excepciones en el código de la aplicación en una jerarquía genérica de **excepciones de tipo unchecked**, habilitando a las aplicaciones para capturar excepciones de acceso a datos sin depender de JDBC.

3.4. DATASOURCE

Un **DataSource** es una interfaz que se encuentra en el paquete **javax.sql.DataSource** de **J2SDK**, la cual se emplea como factoría de objetos **Connection**.

Gracias al empleo de un objeto **DataSource**, nos ahorraremos mucho tiempo y esfuerzo, y controlaremos mejor los posibles errores que se producen al dar de alta una conexión.

Cuando se comenzó a utilizar **jdbc** en los proyectos, debíamos escribir el **Driver** y la **cadena de conexión con la base de datos**, en todos y cada uno de los accesos que realizásemos desde la aplicación a la base de datos, como se muestra en la siguiente clase. Ésta la hemos realizado para comprobar la conexión con la base de datos.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ProbarConexion {

    public ProbarConexion() {
        Consultar();
    }

    private void Consultar(){
        try{
            Class.forName("com.mysql.jdbc.Driver");
            Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/libreria", "root", "");
            Statement st=con.createStatement();
            ResultSet rs=st.executeQuery("select * from libreria");
            while (rs.next()){
                String cadena=Integer.toString(rs.getInt(1));
                cadena=cadena+rs.getString(2);
                cadena=cadena+ Integer.toString(rs.getInt(3));
                cadena=cadena+rs.getString(4);
                System.out.println(cadena);
            }
        }
    }
}
```

```
} catch (SQLException ex) {  
    ex.printStackTrace();  
} catch (ClassNotFoundException e) {  
    e.printStackTrace();  
} finally {  
    try {  
        con.close();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
}  
  
public static void main(String[] args) {  
    ProbarConexion pc= new ProbarConexion();  
}  
}
```

Ej. Acceso jdbc sin utilización de DataSource

No obstante, el objeto **DataSource** se configura una única vez y, posteriormente, se **registra**, normalmente al inicializar la aplicación.

El empleo de **DataSource** está muy extendido en desarrollos web, pudiéndose registrar, normalmente, de varias formas. A continuación, indicaremos algunas de ellas y, posteriormente, mostraremos varias formas de realizarlo en Spring:

- Un objeto que implementa la interfaz **DataSource** se puede registrar mediante el empleo de **JNDI (Java Naming and Directory Interface)**.
- **Struts** también nos permite la configuración de un objeto **DataSource** configurado en el fichero **struts-config.xml**. Indicamos esta posibilidad, ya que, Spring proporciona los medios para utilizar **Struts** como **patrón MVC**.

– **Spring** nos permite dos formas de creación de un **DataSource**:

- **Programando una clase**, en la cual utilizaremos un objeto de la clase **BasicDataSource**, que no es más que una clase que implementa la interfaz **DataSource**, y le añadimos los parámetros necesarios, como se muestra a continuación:

```
BasicDataSource ds = new BasicDataSource();
ds.setDriverClassName("com.mysql.jdbc.Driver");
ds.setUrl("jdbc:mysql://localhost:3306/libreria");
ds.setUsername("root");
ds.setPassword("");
```

- Spring nos permite la configuración de un **DataSource** de una manera muy simple en el fichero **application-context.xml**. Suele ser la opción más utilizada y, además, se puede integrar perfectamente con la opción **JNDI**, como veremos en los distintos ejemplos de configuración de un **DataSource** en el fichero de configuración de **Spring** que mostraremos en los siguientes apartados.

3.4.1. Mapeo de un DataSource utilizando Spring

La forma más sencilla que nos proporciona Spring para configurar un **DataSource**, es utilizar la clase **org.springframework.jdbc.datasource.DriverManagerDataSource**, a la que le pasaremos como **property**s los datos básicos de una conexión, como mostramos en el siguiente recuadro:

```
<!-- ===== DataSource ===== -->
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
    <value>jdbc:mysql://localhost:3306/libreria</value>
  </property>
  <property name="username">
    <value>root</value>
```



```

    </property>
    <property name="password">
        <value></value>
    </property>
</bean>

```

Ej. Simple mapeo **DataSource** en **Spring**.

3.4.2. Mapeo de un DataSource utilizando Spring y JNDI

JNDI es un estándar **J2EE** para acceder y nominar directorios. Spring nos proporciona una clase **org.springframework.jndi.JndiObjectFactoryBean** que nos da soporte a esta tecnología. Es ampliamente utilizado en aplicaciones comerciales.

```

<bean id="myDataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/ dataSource " />
</bean>

```

Ej. Mapeo de **DataSource** utilizando **Spring** y **JNDI**.

3.4.3. Mapeo de un DataSource utilizando el paquete Apache Commons dbcp

Otra alternativa que proporciona **Spring**, es la configuración de un **DataSource** fuera del contenedor de J2EE, utilizando la clase **org.apache.commons.dbcp.BasicDataSource**, que pertenece al proyecto [Apache's Jakarta Commons DBCP](#). Mediante esta clase, podremos establecer algunas propiedades adicionales:

```

<!-- ===== Pool de Conexiones ===== -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
    <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
        <value>jdbc:mysql://localhost:3306/libreria</value>
    </property>
    <property name="username">
        <value>root</value>
    </property>

```

```
<property name="password">
    <value></value>
</property>
<property name="maxActive">
    <value>10</value>
</property>
<property name="maxWait">
    <value>5000</value>
</property>
<property name="defaultReadOnly">
    <value>false</value>
</property>
</bean>
```

Ej. Mapeo **DataSource** utilizando **commons dbcp**.

3.4.4. Mapeo de un DataSource mediante el empleo fichero .properties

Finalmente, mostraremos la posibilidad de configurar un **DataSource** mediante el empleo de ficheros de propiedades. Con esto conseguimos que, si deseamos modificar alguna propiedad de la configuración del DataSource, sólo tengamos que modificar el fichero **.properties** (en nuestro caso **dao.properties**).

Los ficheros **.properties** son ficheros de texto plano en los que por cada línea se establece el nombre de una propiedad y su valor asociado, separados por un signo de igual (=). Además, son almacenados con el nombre **xxxx.properties**, siendo **xxxx** el nombre del fichero que nosotros queramos definir.

Es importante tener en cuenta que, en ocasiones, una aplicación puede tener varios ficheros **.properties**, por ejemplo: **dao.properties** (establece las propiedades de la base de datos), **mail.properties** (establece las propiedades del servidor de mail), **file.properties** (establece las propiedades necesarias para subir ficheros al servidor).

A continuación, mostraremos un ejemplo de fichero de propiedades, en el que estableceremos las propiedades de nuestro **DataSource**.

```
driver= com.mysql.jdbc.Driver
url= jdbc:mysql://localhost:3306/libreria
username= root
```

3 Configuración y manejo de persistencia con Spring JDBC

```
password=  
maxActive = 10  
maxWait= 5000  
defaultReadOnly= false
```

Ej Fichero **dao.properties**

Para que los ficheros de propiedades sean accesibles en Spring, necesitamos crear un Bean utilizando la clase:

```
org.springframework.beans.factory.config.PropertyPlaceholderConfigurer.
```

Gracias a esta clase, Spring es capaz de localizar todos los ficheros de propiedades para, posteriormente, poder trabajar con ellos.

A continuación, mostraremos un ejemplo de Bean que realiza lo anteriormente mencionado. En dicho ejemplo, aunque hemos utilizado la etiqueta de **XML list**, en realidad podríamos eliminarla. No obstante, en caso de querer cargar otro fichero de propiedades, sólo necesitaríamos añadir otro valor entre las etiquetas **list**, por ejemplo: **<value>WEB-INF/conf/mail.properties</value>**.

```
<bean id="propertyConfigurer"  
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
  <property name="locations">  
    <list>  
      <value>WEB-INF/conf/dao.properties</value>  
    </list>  
  </property>  
</bean>
```

Ej. Método de Spring para localizar los ficheros de propiedades.

3 Configuración y manejo de persistencia con Spring JDBC

Una vez cargado el fichero de propiedades en Spring, únicamente debemos crear nuestro **DataSource** de cualquiera de las maneras anteriormente mencionadas, y utilizar los valores almacenados en el fichero de propiedades. A continuación, mostraremos cómo se emplean los valores:

```
<!-- ===== Pool de Conexiones ===== ->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
    <property name="driverClassName">
        <value>${driver}</value>
    </property>
    <property name="url">
        <value>${url}</value>
    </property>
    <property name="username">
        <value>${username}</value>
    </property>
    <property name="password">
        <value>${password}</value>
    </property>
    <property name="maxActive">
        <value>${maxActive}</value>
    </property>
    <property name="maxWait">
        <value>${maxWait}</value>
    </property>
    <property name="defaultReadOnly">
        <value>${defaultReadOnly}</value>
    </property>
</bean>
```

Ej. Mapeo **DataSource** utilizando fichero **dao.properties**

3.5. UTILIZACIÓN DE JDBC CON SPRING

Una vez configurado el **DataSource**, estudiaremos cómo Spring utiliza el **patrón DAO** para trabajar con la capa de persistencia.

Comenzaremos por observar un fragmento del fichero **aplicación-context.xml** en el cual se define un bean para la **capa DAO** utilizando **Spring**:

```
<bean id="librosDao" class="com.formacion.tienda.dao.GestionLibrosDAOJDBC">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
</bean>
```

Una vez definido el bean en el fichero de configuración de **Spring**, siguiendo con la filosofía de este framework, los siguientes pasos que debemos seguir son:

- La creación de una **interfaz**, en nuestro caso la llamaremos **GestionLibrosDAO**, en la que definiremos los métodos que debemos implementar.

```
package com.formacion.tienda.dao;

import java.util.List;

import com.formacion.tienda.dto.Libro;
import com.formacion.tienda.dto.LibroPk;

public interface GestionLibrosDAO {
    public List getListaLibros();
    public Libro getLibro(LibroPk libroPk);
    public void guardarLibro(Libro libro);
    public void modificarLibro(Libro libro);
    public void borrarLibro(Libro libro);
}
```

- Seguidamente, realizamos una primera versión de la clase que implementa dicha interfaz, en la que mostraremos los conocimientos aprendidos en el segundo tema sobre **Inversión de Control (IoC)**.

3 Configuración y manejo de persistencia con Spring JDBC

Como hemos observado en la definición del bean **librosDao**, que hemos realizado anteriormente, indicamos que este bean utilizará otro bean llamado **datasource**, el cual instanciará mediante **IoC**.

```
private DataSource ds;           //declaración del objeto datasource
public void setDataSource(DataSource ds) {
    this.ds = ds;                //Por IoC cargamos el bean llamado datasource
}
```

Una vez hayamos cargado por **IoC** el bean **datasource**, utilizaremos este objeto para establecer la conexión con la base de datos y operar utilizando **JDBC**.

```
package com.formacion.tienda.dao;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import com.formacion.tienda.dto.Libro;
import com.formacion.tienda.dto.LibroPk;

public class GestionLibrosDAOJDBC implements GestionLibrosDAO {
    /** Logger for this class and subclasses */
    protected final Log log = LogFactory.getLog(getClass());

    private DataSource ds;

    Connection con;

    public void setDataSource(DataSource ds) {
        this.ds = ds;
    }
}
```

```
}

public List getListaLibros() {

    List listaLibros=new ArrayList();
    log.info("Entro en obtener libros");
    try{
        con=ds.getConnection();
        Statement st=con.createStatement();
        ResultSet rs=st.executeQuery("select * from libreria");
        while (rs.next()){
            Libro libro=new Libro();
            libro.setCodigoLibro(rs.getInt(1));
            libro.setDescripcionProducto(rs.getString(2));
            libro.setPrecioProducto(rs.getInt(3));
            libro.setAutor(rs.getString(4));
            listaLibros.add(libro);
        }
    }catch(SQLException ex){
        ex.printStackTrace();
    } finally{
        try{
            con.close();
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
    return listaLibros;
}
}
```

Ej. Implementación sin utilizar las clases que nos facilita **Spring**

A continuación, mostraremos un ejemplo más refinado con la clase que nos proporciona Spring **JdbcTemplate**. En dicho ejemplo, comprobaremos que Spring encapsula las operaciones de creación de **conexión** del **Statement**. De esta manera, posteriormente no tendremos que formatear el **ResultSet**, para almacenar la información en un objeto de tipo **List**, ya que, la clase **JdbcTemplate** tiene ya implementado los métodos que realizan estas operaciones como veremos en el siguiente apartado de este tema.

3 Configuración y manejo de persistencia con Spring JDBC

Una de las características más importantes que queremos remarcar es que no hace falta encapsular el código que accede a la base de datos mediante la clase **JdbcTemplate** entre bloques **try/catch/finally**, en los que deberíamos capturar las excepciones y cerrar las conexiones, ya que, estas operaciones las gestiona Spring por nosotros.

```
package com.formacion.tienda.dao;

import java.util.ArrayList;
import java.util.List;

import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.jdbc.core.JdbcTemplate;

import com.formacion.tienda.dto.Libro;
import com.formacion.tienda.dto.LibroPk;

public class GestionLibrosDAOJDBC implements GestionLibrosDAO {
    /** Logger for this class and subclasses */
    protected final Log log = LogFactory.getLog(getClass());
    private DataSource ds;

    public void setDataSource(DataSource ds) {
        this.ds = ds;
    }

    public List getListaLibros() {
        List listaLibros=new ArrayList();
        log.info("Entro en obtener libros");
        JdbcTemplate jt = new JdbcTemplate(ds);
        String sql="select * from libreria";
        listaLibros=jt.queryForList(sql);
        return listaLibros;
    }
}
```

Ej. Implementación utilizando **JdbcTemplate**

3.5.1. Operaciones utilizando JdbcTemplate

JdbcTemplate es la clase principal del modulo **core** de Spring. Esta clase nos simplifica el uso de **JDBC** y nos permite evitar algunos errores comunes de programación, como dejar abiertas las conexiones, sufriendo como consecuencia la caída del servidor y el cese del funcionamiento de la aplicación.

Además, ofrece tres **constructores**:

JdbcTemplate();	<p>Constructor por defecto, al cual no se le proporciona ningún parámetro una vez declarado: JdbcTemplate jt=new JdbcTemplate();</p> <p>Deberemos de decirle cuál es el DataSource que deseamos utilizar: jt.setDataSource(ds);</p>
JdbcTemplate(DataSource ds);	<p>Constructor que recibe como parámetro el DataSource.</p> <p>JdbcTemplate jt = new JdbcTemplate(ds);</p>
JdbcTemplate(DataSource ds, boolean lazyInit);	<p>Constructor que recibe como parámetros:</p> <ul style="list-style-type: none"> – El DataSource definido. – Un tipo boolean (true o false), el cual indica si utilizaremos el lazy (perezoso). El tipo trabaja de la siguiente manera: <ul style="list-style-type: none"> • false. Trabaja de forma normal, es decir, al traer un objeto de la base de datos, si tiene otros objetos relacionados también los traerá. Como ejemplo si tenemos una tabla usuario, y otra relacionada usuarios_rol a través de FK (Foreign Key), obtendremos el usuario, y la lista de roles asociado al usuario en cuestión. • true. Utilizaremos el modo perezoso, es decir, utilizando el mismo juego de tablas anteriormente mencionado, si obtenemos el usuario, sólo nos devolverá el usuario, y luego mediante el empleo de la sesión, obtendremos los roles asociados a dicho usuario.

3 Configuración y manejo de persistencia con Spring JDBC

A continuación, veremos algunas de las **operaciones** que podemos realizar con la instancia del objeto **JdbcTemplate**:

Operaciones de consulta	<code>int queryForInt(String sql)</code>
	<code>int queryForInt(String sql, Object[] args)</code>
	<code>long queryForLong(String sql)</code>
	<code>long queryForLong(String sql, Object[] args)</code>
	<code>Object queryForObject(String sql, Class requiredType)</code>
	<code>Object queryForObject(String sql, Class requiredType, Object[] args)</code>
	<code>List queryForList(String sql)</code>
	<code>List queryForList(String sql, Object[] args)</code>
Operaciones de modificación	<code>int update(String sql)</code>
	<code>int update(String sql, Object[] args)</code>
	<code>int update(String sql, Object[] args, int[] argTypes)</code>
Operaciones de inserción y eliminación utilizando la cláusula execute	<code>void execute(String sql)</code>
Operaciones Batch	<code>int batchUpdate(String[] sql)</code>

3.5.2. Utilización de la clase JdbcDaoSupport

JdbcDaoSupport es una clase que proporciona la distribución de Spring. Dicha clase la debemos extender en nuestra clase.

```
package com.formacion.tienda.dao;

import java.util.List;

import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.RowMapperResultReader;
import org.springframework.jdbc.core.support.JdbcDaoSupport;

import com.formacion.tienda.dto.Libro;
import com.formacion.tienda.dto.LibroPk;

public class GestionLibrosDAOSupportJdbc extends JdbcDaoSupport implements
    GestionLibrosDAO {

    /**
     * Método que devuelve la lista de libros
     */
}
```

```

    */
    public List getListaLibros() {
        return getJdbcTemplate().queryForList("select * from libreria");
    }
    /**
    * Método que obtiene el detalle de un libro
    */
    public Libro getLibro(LibroPk libroPk) {
        RowMapper rowMapper = new LibroRowMapper();
        List listaLibro=getJdbcTemplate().query("select * from libreria where
codLibro=?",new Object[]{ Integer.toString(libroPk.getCodLibro())},
        new RowMapperResultReader(rowMapper,1));
        Libro libro=(Libro)listaLibro.get(0);
        return libro;
    }
    /**
    * Método que inserta un libro
    */
    public void guardarLibro(Libro libro) {
        getJdbcTemplate().execute("insert into libreria
values("+Contador()+"','"+libro.getDescripcion()+"','"+libro.getPrecio()+"','"+libro.getAutor()+"");");
    }
    /**
    * Método que modifica los valores de un libro
    */
    public void modificarLibro(Libro libro) {
        getJdbcTemplate().update("Update libreria set descripcion= ? , precio= ? ,autor= ? where
codLibro= ? ", new Object[] { libro.getDescripcion(),Integer.toString(libro.getPrecio()),
libro.getAutor() , Integer.toString(libro.getCodLibro())});
    }
    /**
    * Método que elimina un libro
    */
    public void borrarLibro(Libro libro) {
        getJdbcTemplate().execute("delete from libreria where
codLibro="+libro.getCodLibro());
    }
}

```

3 Configuración y manejo de persistencia con Spring JDBC

```
/*
 * Método privado que utilizamos para obtener el código de un libro nuevo
 */
private int Contador(){
    int contador=0;
    contador=getJdbcTemplate().queryForInt("select codLibro from libreria ORDER BY
codLibro DESC");
    contador+=1;
    return contador;
}
}
```

- **Spring** se basa en el empleo del patrón de **diseño DAO**.
- **Spring** nos proporciona los siguientes **paquetes** para facilitarnos el trabajo con base de datos: **core**, **datasource**, **object** y **support**.
- **Spring** permite la creación de los objetos **DataSource** para conectarnos a la base de datos de varias formas.
 - Mediante el fichero **struts-config.xml** de **struts**.
 - Implementando **JNDI**.
 - En una clase instanciar un objeto **BasicDataSource**.
 - Diversas formas de configuración en el fichero de configuración de **Spring**.
- **Spring** nos proporciona dos **clases** que nos facilitan mucho el trabajo.
 - **JdbcTemplate**
 - **JdbcDaoSupport**

4.1. INTRODUCCIÓN A SPRING MVC	63
4.2. PATRÓN DE DISEÑO MVC	63
4.2.1. Ventajas del patrón MVC	64
4.3. PATRÓN DE DISEÑO MVC UTILIZANDO SPRINGMVC	64
4.3.1. Ventajas adicionales del patrón MVC utilizando SpringMVC	64
4.3.2. DispatcherServlet	65
4.3.3. WebApplicationContext	67
4.3.4. Controladores	68
4.3.5. ViewResolvers	70
4.3.6. Tags utilizadas en aplicaciones realizadas con Spring	71
4.3.6.1. Tags de Spring	71
4.3.7. JSTL CORE	72

4.1. INTRODUCCIÓN A SPRING MVC

A lo largo del manual, estamos comprobando que el framework **Spring** cubre casi todas las necesidades de la programación moderna en aplicaciones web. Es por esta razón que no podía faltar la implementación del patrón **MVC (Model- View- Controller)** o **Modelo Vista y Controlador**, patrón por excelencia en las aplicaciones web, principalmente desde la aparición de frameworks como **Struts**, con más de cinco años de historia.

4.2. PATRÓN DE DISEÑO MVC

El **patrón de diseño MVC** separa la aplicación web en tres **módulos**:

- **Controlador**. Se encarga de manejar las peticiones realizadas desde la interfaz de usuario, reacciona a la entrada del usuario y obtiene de la capa del modelo, los datos relativos a la entrada de datos.
- **Modelo**. Representa los objetos que interaccionan con los datos almacenados, tanto en base de datos como en ficheros, y los elementos de la **capa de negocio o servicio**.
- **Vista**. Parte visual del modelo **MVC**, es decir el usuario interacciona directamente con elementos de esta capa.

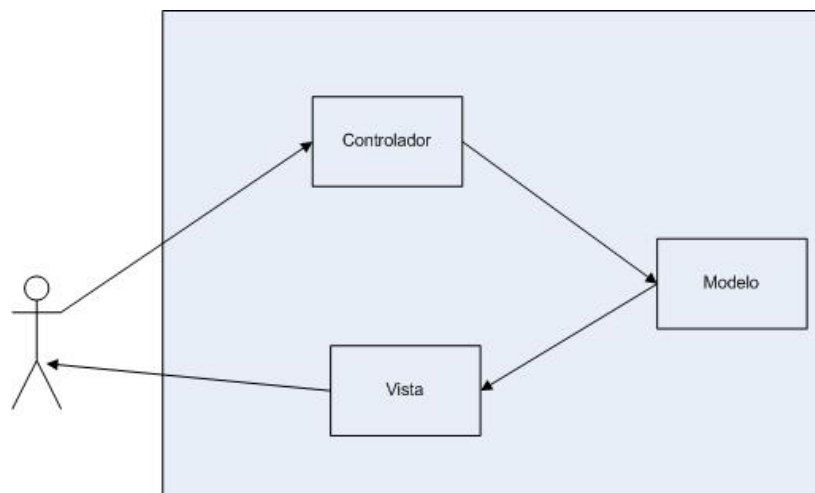


Diagrama MVC

4.2.1. Ventajas del patrón MVC

Algunas de las principales **ventajas** de utilizar el MVC incluyen:

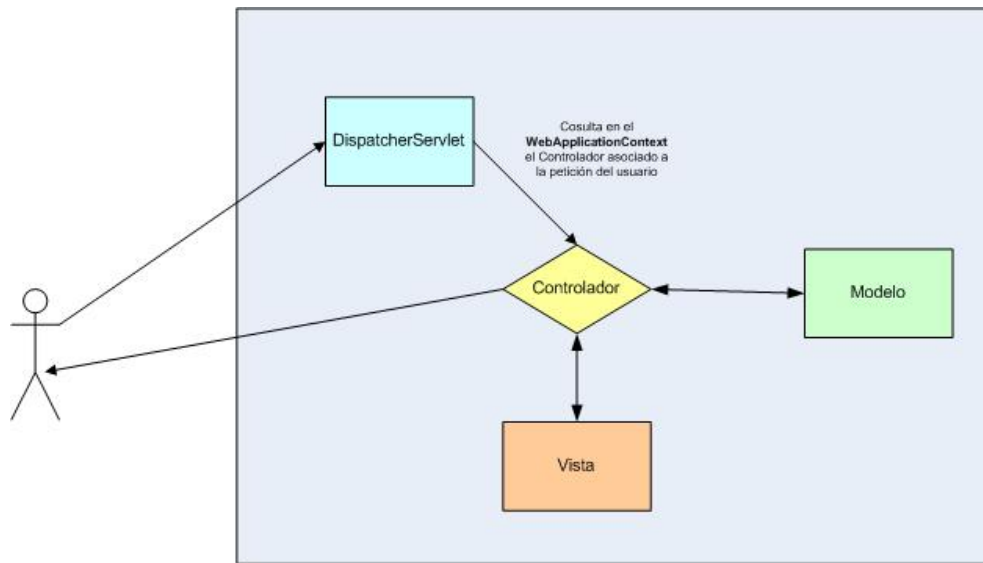
- **Un buen código que implementa el patrón MVC no debe ser intrusivo.** Es decir, los diferentes módulos deben ser lo más autónomos posibles, es decir, la vista, el controlador y el modelo deben estar claramente separados.
- **Alta reutilización y adaptabilidad.** Diferentes funcionalidades pueden acceder a las mismas capas del **MVC** y se pueden implementar tanto para navegadores web (**http** o **HTTPS**) como para navegadores inalámbricos (**WAP**).
- **Rápido desarrollo.** El tiempo de desarrollo puede reducirse considerablemente porque los diseñadores pueden trabajar en la vista, los programadores más familiarizados con el acceso datos pueden hacerlo en el modelo y otros expertos en el desarrollo de aplicaciones, en el controlador.
- **Bajo coste de mantenimiento.** La separación de presentación y lógica de negocio también hace más fácil mantener y modificar una aplicación web.

4.3. PATRÓN DE DISEÑO MVC UTILIZANDO SPRINGMVC

Al igual que **Struts** y la mayoría de los frameworks web realizados en Java, **Spring** utiliza una única instancia del **servlet** que maneja todas las peticiones. Dicho **servlet** es el encargado de indicar al controlador la petición realizada por el usuario, el controlador obtiene los datos del modelo y, posteriormente, resuelve el tipo de vista que se le devolverá al usuario.

4.3.1. Ventajas adicionales del patrón MVC utilizando SpringMVC

- **Proporciona la renderización en distintos formatos.** Modificando el bean **viewResolver** del fichero de configuración de **Spring**, podemos obtener diversos formatos de salida: **JSP**, **Velocity**, **XSLT**, **Tiles**, **jasperreports**, **freeMarker**.
- **Implementa una amplia serie de controladores que se ajustan a las necesidades del desarrollador.**



Patrón MVC utilizado en Spring

En los próximos apartados, describiremos los distintos componentes que componen el módulo **SpringMVC**.

4.3.2. DispatcherServlet

El **servlet** que utiliza **SpringMVC** se llama **DispatcherServlet**. En la distribución de Spring, lo encontramos en el paquete web

org.springframework.web.servlet.DispatcherServlet.

Para las personas familiarizadas con Struts, diremos que su funcionalidad es semejante al **servlet ActionServlet** utilizado por éste, siendo su función manejar las peticiones realizadas por el usuario.

Las aplicaciones web que desarrollemos utilizando **Spring MVC** tienen una serie de características:

- Podemos declarar varios **DispatcherServlet**, teniendo asociado un **WebApplicationContext** cada uno de ellos.
- Una aplicación de este tipo siempre debe tener asociado al menos un **WebApplicationContext**.

- Cuando utilicemos la configuración por defecto del **servlet**, **DispatcherServlet**, debemos tener un fichero **XML** en el cual definimos los beans. Dicho fichero lo encontraremos en el directorio **/WEB-INF** de nuestra aplicación y se suele nombrar por **(nombre del servlet)-servlet.xml**. En el ejemplo de aplicación que mostraremos en el tema 5, lo hemos nombrado **springapp-servlet.xml**.

Por otra parte, **DispatcherServlet** consulta en el fichero de configuración de **Spring** (en el ejemplo **springapp-servlet.xml**), los mapeos relativos a cada controlador, indicando cuál debemos utilizar para cada petición del usuario.

Por último, al igual que pasaba con **Struts**, debemos configurar el servlet **DispatcherServlet** en el fichero **web.xml**. Asimismo, además de declarar el servlet, es necesario indicar el mapeo que utiliza **SpringMVC** ("***.htm**"), de la misma manera que **Struts** utiliza como mapeo "***.do**". Todo ello indica que las URL finalizadas en "**.htm**" serán manejadas por el **servlet DispatcherServlet**.

```
<servlet>
  <servlet-name>
    Springapp
  </servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>springapp</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

Declaración del servlet **DispatcherServlet** y su mapeo en el fichero **web.xml**

4.3.3. WebApplicationContext

El **DispatcherServlet** iniciará el **WebApplicationContext**, que contiene todas las relaciones de los beans y es el responsable de manejar la lógica de la aplicación, obteniendo de él las declaraciones de algunos beans implementados por el equipo de desarrollo de **Spring**.

A continuación, mostramos una breve descripción de algunos de estos **beans**:

- **HandlerMapping**: responsable de determinar el controlador apropiado que debe resolver la petición. Normalmente utilizaremos como criterio de decisión la URL.

```
<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/listaLibros.htm">
        controladorSpring
      </prop>
      <prop key="/detalleLibro.htm">
        controladorFormLibro
      </prop>
    </props>
  </property>
</bean>
```

- **ViewResolver**. Es el encargado de establecer el formato de la vista que será devuelta al usuario. Por ejemplo: **JSP**, **Velocity**, **XSLT**, **Tiles**, **jasperreports**. Lo estudiaremos en más detalle en el siguiente apartado.

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass">
    <value>
      org.springframework.web.servlet.view.JstlView
    </value>
  </property>
  <property name="prefix">
```

```
<value>/WEB-INF/jsp/</value>
</property>
<property name="suffix"><value>.jsp</value></property>
</bean>
```

- **LocaleResolver.** Es el encargado de resolver el **locale** y se utiliza principalmente para establecer el lenguaje en el que se devolverá la vista correspondiente en aplicaciones que utilizan multilenguaje.
- **MultipartResolver.** Proporciona el soporte para subir ficheros al servidor, vía **HTTP**. Es una implementación de **FileUpload**.
- **HandlerExceptionResolvers.** Ofrece soporte para capturar las excepciones.
- **Controller.** Concede soporte para gestionar la lógica interna de nuestra aplicación y el acceso a la lógica de negocio apropiada para cada controlador. Lo estudiaremos en más detalle cuando expliquemos los controladores.
- **HandlerInterceptor.** Proporciona la capacidad de capturar las peticiones **HTTP**. Se suele utilizar en algunos servicios comunes a la mayoría de aplicaciones web, como puede ser la seguridad.

4.3.4. Controladores

Los **Controllers** o **controladores** son los encargados de recibir los eventos que se envían desde la interfaz de usuario, acceden a los datos y proporcionan una vista como respuesta.

Son el elemento principal en el planteamiento de **SpringMVC**. Por esta razón, se ha desarrollado un interfaz **Controller** que devuelve un objeto **ModelAndView**, el cual podemos implementar en nuestra aplicación, como se muestra en la siguiente clase de la aplicación de ejemplo.

Por otro lado, aunque implementar la interfaz **Controller** es una solución, el equipo de desarrollo de **Spring** nos ha proporcionado una serie de clases con la mayoría de las funcionalidades que necesitamos para desarrollar nuestras aplicaciones que nos facilitan mucho el trabajo.

Los **tipos de controladores** se dividen en tres **conjuntos**:

- **AbstractController**. Clase abstracta que implementa la interfaz **Controller** que proporciona algunos servicios básicos como proporcionar soporte a **caché**.
- **MultiActionController**. Permite agregar varios **Action** a un controlador, agrupando la funcionalidad de éstos. Para los programadores familiarizados con **Struts**, comentaremos que es similar a utilizar un **DispatchAction**.
- **CommandController**. Proporciona la forma de interaccionar entre los datos recibidos del formulario, vía **HttpServletRequest**, y los objetos de datos.

Su función es similar al **ActionForm** de **Struts**. La única diferencia está en que actúan como controlador por sí mismo sin necesidad de utilizar un objeto **Action**.

A su vez, se ha implementado una serie de clase que heredan de **CommandController**:

- **AbstractCommandController**
- **AbstractFormController**
- **SimpleFormController**
- **AbstractWizardFormController**

En el próximo tema, mostraremos varios ejemplos de controladores. Así, por una parte, implementaremos la interfaz **Controller** para obtener la lista de libros y, por otra, extenderemos la clase **SimpleFormController**, gracias a la cual capturaremos los datos del formulario del detalle de cada libro. Por último, con los datos del **form** podremos **borrar, modificar e insertar un libro**, funciones básicas del mantenimiento de una entidad.

4.3.5. ViewResolvers

Como mencionamos en el apartado de 4.3.4., todos los controladores definidos en **Spring MVC** devuelven una instancia del objeto **ModelAndView**.

Spring proporciona un grupo de **ViewResolvers** u objetos utilizados para resolver el tipo de vista que el servidor debe servir al usuario. A continuación, mostramos el listado de **ViewResolvers**:

- **AbstractCachingViewResolver**. Clase abstracta que implementa **ViewResolver**. Cachea las vistas.
- **XmlViewResolver**. Implementación de **ViewResolver**, la cual utiliza como fichero de configuración un fichero **XML**, cuya cabecera debe utilizar el **DTD spring-beans.dtd**. El fichero por defecto lo debemos alojar en **/WEB-INF/views.xml**.
- **ResourceBundleViewResolver**. Implementación de **ViewResolver**. Ésta utiliza la definición de los beans en un **ResourceBundle**, normalmente definido en un fichero **.properties**. El nombre por defecto de este fichero es **views.properties**.
- **UrlBasedViewResolver**. Implementación de **ViewResolver** utilizada para resolver la vista a devolver mediante la URL.
- **InternalResourceViewResolver**. Subclase de **UrlBasedViewResolver** que soporta **InternalResourceView** (**JSPs**), y subclases como **JstlView** y **TilesView**.
- **VelocityViewResolver**. Subclase de **UrlBasedViewResolver** que soporta **VelocityView** (plantillas de **Velocity**).
- **FreeMarkerViewResolver**. Subclase de **UrlBasedViewResolver** que soporta **FreeMarkerView** (plantillas de **FreeMarker**).

4.3.6. Tags utilizadas en aplicaciones realizadas con Spring

Las aplicaciones basadas en **SpringMVC** se fundamentan en:

- El conjunto de **etiquetas** que nos proporciona **Spring**, utilizadas principalmente por los diversos **controladores** para: devolver los errores o mensajes, recoger y cargar parámetros de un formulario, o gestionar la navegación de la aplicación.
- Las **librerías de jstl (java standart tag lib)**, desarrolladas por **SUN**, que recogen un conjunto de funcionalidades que nos permiten eliminar los **scriptles** `<%...%>`, ampliamente utilizados en los comienzos del desarrollo de aplicaciones basadas en **jsp**.

4.3.6.1. Tags de Spring

Spring también tiene su propia librería de **tags** al igual que **Struts**. Así, proporciona una librería de **tag** simple pero muy potente que genera **HTML** sin mucho coste.

Para poder utilizar dicha librería, debemos declarar el archivo **spring.tld** en el fichero **web.xml**, para permitir que la aplicación interprete correctamente las **tags** de **Spring**.

```
<taglib-uri>/spring</taglib-uri>
<taglib-location>/WEB-INF/spring.tld</taglib-location>
</taglib>
```

Ejemplo de declaración del **tld** de **Spring** en el fichero **web.xml**

A continuación, mostramos un cuadro con las **tags** de **Spring**:

BindErrorsTag	Evalúa si un bean contiene errores.
BindTag	Soporta la evaluación de errores de bind para ciertos beans y evalúa las propiedades de los beans.
EscapeBodyTag	Tag utilizada para salir del cuerpo de la página HTML , es decir, entre las etiquetas <code><body></body></code>
HtmlEscapeTag	Proporciona la salida HTML por defecto para la página en la que nos encontremos.
MessageTag	Tag utilizada para mostrar los mensajes.

RequestContextAwareTag	Superclase para todas las clases que requieren un objeto RequestContext .
ThemeTag	Es utilizada para buscar un mensaje en el ámbito de la página.
TransformTag	Tag que emplearemos para transformar los valores proporcionados por el controlador.

Por último, una vez descritas las **tags** que nos proporciona **Spring**, mostraremos algunos ejemplos que hemos utilizado en nuestra aplicación.

```
//Se suele utilizar este fragmento de código para evaluar y mostrar los mensajes de error,
aunque podríamos utilizar la tag de Spring <spring:MessageTag>

<spring:bind path="libro.*">
  <c:if test="${not empty status.errorMessages}">
    <div class="error">
      <c:forEach var="error" items="${status.errorMessages}">
        <c:out value="${error}" escapeXml="false"/><br />
      </c:forEach>
    </div>
  </c:if>
</spring:bind>

//Como hemos comentado anteriormente, utilizamos la tag <spring:bind> para evaluar la
información de un bean.

<spring:bind path="libro.codLibro">
  <input type="hidden" name="codLibro"
    value="<c:out value="${status.value}"/>">
</spring:bind>
```

4.3.7. JSTL CORE

Estas etiquetas son muy útiles en las **JSP** porque sustituyen otras etiquetas algo más difíciles de usar, consiguiéndose así un código más limpio. En esta librería hay **etiquetas** con distintas utilidades:

- **Etiquetas de uso general:** manipulación de variables.

- **Out.** Es la más utilizada. Con esta etiqueta, se evalúa una expresión y se pinta el resultado en la página JSP.
 - **Set.** Establece el valor de una variable a una propiedad de un bean.
 - **Remove.** Elimina una variable que anteriormente se ha establecido con la etiqueta set.
 - **Match.** Etiqueta que captura excepciones que se pueden provocar en la página JSP.
- **Etiquetas condicionales.** Estas etiquetas ejecutan un condicional evaluando expresiones:
- **If.** Es el condicional típico **'if'** de cualquier lenguaje de programación. Muestra lo que existe entre las etiquetas de inicio (**<c:if>**) y de fin (**</c:if>**) si la expresión que se evalúa es verdadera.
 - **choose, when y otherwise.** Es el condicional **'if...else'** donde el inicio y el fin está limitado por las etiquetas **<c:choose>** y **</c:choose>**. El **'if'** corresponde a la etiqueta **<c:when>** y el **'else'** a **<c:otherwise>**.
- **Etiquetas que iteran una lista.** Estas etiquetas proporcionan un mecanismo para iterar sobre una amplia variedad de colecciones de objetos. Dos etiquetas muy flexibles están disponibles para llevar a cabo esto:
- **forEach.** Repite el contenido del cuerpo una vez por cada uno de los elementos de la colección.
- Esta colección puede ser una instancia de **Collection**, **Map**, **Iterator**, **Enumeration**, un **array** e, incluso, una cadena separada por comas.
- **forEachTokens.** Iteración sobre entidades definidas por el usuario que deben estar separadas por algún delimitador y encerradas en una cadena.
- **Etiquetas de url.** Se utilizan para importar y redireccionar a otras páginas. Éstas son:

- **Import.** La utilizamos para importar un trozo de código mediante una url relativa al contexto, a la página, a una url absoluta, etc.
- **redirect:** se usa para redireccionar la petición a una nueva página o a algún controlador para que realice alguna operación.

Todas estas bibliotecas tienen una peculiaridad, de la que a veces hacen uso otras etiquetas, que consiste en obtener el valor de las variables mediante el uso de lo que hace las veces de **scriptlet en Struts** (**`${variable}`**).

- **SpringMVC** es un módulo que proporciona la distribución de **Spring** y que implementa el **patrón de diseño MVC (Modelo vista y Controlador)**.
- **SpringMVC**, como la mayoría de los frameworks web realizados en java, utiliza un **servlet** que se encarga de procesar todas las peticiones. En el caso de SpringMVC, lo llamamos **DispatcherServlet**. Dicho servlet consulta en el fichero **(nombre del servlet)-servlet.xml**, el controlador adecuado.
- **Spring** proporciona una amplia gama de **controladores**, que se adecuan a la mayoría de los programadores.
- Los controladores devuelven un objeto **ModelAndView**, que devuelve la vista en el formato que deseemos: **JSP, Velocity, XSLT, Tiles, JasperReports, freeMarker**, etc.
- Para facilitar estas salidas, **SpringMVC** utiliza **los ViewResolvers**.
- **SpringMVC** ofrece un conjunto de librerías para soportar el manejo de errores, la salida de los datos y la navegación de la aplicación. Además, utiliza las librerías **JSTL** para organizar la información en la página principalmente.

5.1. INTRODUCCIÓN.....	79
5.2. APLICACIÓN DE EJEMPLO.....	79
5.2.1. Análisis de la aplicación.....	81
5.3. CREACIÓN DE UN NUEVO PROYECTO UTILIZANDO ECLIPSE	
WTP	83
5.4. ESTRUCTURA DE LA APLICACIÓN.....	87
5.4.1. Añadir al proyecto la naturaleza de proyecto Spring	88
5.4.2. Configuración del entorno	89
5.4.3. Bibliotecas necesarias en la aplicación.....	92
5.4.4. Spring.tld	94
5.4.5. Clases del modelo.....	97
5.4.6. Ficheros de propiedades de la aplicación	99
5.4.6.1. resourceBundle.properties	99
5.4.6.2. log4j.properties	100
5.4.7. Capa de presentación	100
5.4.7.1. Páginas JSP de la aplicación.....	101
5.4.7.2. Clase de control utilizando el patrón Spring	
MVC	105
5.4.8. Capa de servicio	109
5.4.9. Capa de persistencia.....	111
5.4.10. Descripción del despliegue de la aplicación	116
5.4.11. Fichero de definición de los beans.....	117
5.5. DESPLIEGUE DE LA APLICACIÓN	121

5.1. INTRODUCCIÓN

En este tema mostraremos cómo realizar una aplicación basada en **Spring**, utilizando el entorno de desarrollo que mostramos en el primer tema, es decir, emplearemos la **Plataforma Web Tools de Eclipse**.

Esta aplicación contendrá todo lo aprendido en los anteriores temas y será muy útil para que tengamos una visión general de **Spring**.

5.2. APLICACIÓN DE EJEMPLO

Para realizar la aplicación basada en Spring a la que hemos hecho referencia en el apartado anterior, aplicaremos lo aprendido en los temas anteriores. De esta forma:

- Realizaremos una aplicación que utiliza el **patrón MVC**, empleando el módulo **SpringMVC**.
- Dicha aplicación accederá a la base de datos para gestionar la persistencia de datos mediante el empleo de **JDBC**, utilizando el módulo **SpringDAO**.
- El control estará gestionado por la factoría de **Beans** y el contexto mediante el empleo del fichero **application-context.xml** aunque, siguiendo la convención de nombres de Spring, lo hemos nombrado **springapp-servlet.xml**. Para ello, se usarán los módulos que proporciona Spring: **Spring Core** y **Spring Context**.

La **secuencia de acciones** que se van a implementar en la aplicación son:

1. La aplicación mostrará el listado de libros que tiene la librería. En dicho listado aparecerán cuatro **campos**:

- **Código Libro**. Código que identifica al libro de forma única. Este campo representa la clave primaria de cada libro en la base de datos.

En el listado lo asociaremos a un **<a ef.> ** de HTML, gracias al cual podremos acceder a la pantalla de detalle de un libro que se encuentre en la base de datos.

- **Descripción**. Representa los títulos disponibles.

- **Precio.** Importe del libro. Para simplificar lo hemos representado utilizando un valor **int**, es decir, no tendrá parte decimal.
- **Autor.** Autor del libro.

Además, hemos añadido un botón llamado **Añadir Libro**, que nos redirigirá al formulario de **detalleLibro**, apareciendo todos los valores en blanco. Una vez cumplimentados los campos, daremos de alta un nuevo libro.

2. El usuario puede acceder a la pantalla de **detalleLibro** de dos formas diferentes:

- **Modo Inserción.** Desde el listado de libros, pulsamos el botón **Añadir Libro** y nos aparece la pantalla **detalleLibro** con todos los valores del formulario en blanco o con el valor cero, en caso del precio, al tratarse de un campo **int**.

A través de este modo, podemos rellenar todos los valores del formulario con los datos del libro que deseemos y pulsar sobre el botón **Insertar**. Para almacenar la información en la **base de datos**. Una vez hecho esto, el sistema nos redirigirá al listado de **Libros**, en el que podremos comprobar que aparece el libro que hemos añadido.

Por otro lado, en caso de no desear dar de alta un nuevo libro, añadiremos un botón **Volver Lista**, el cual nos devolverá al listado de libros sin insertar ningún valor en la **base de datos**.

- **Modo Detalle.** Desde el listado de libros, pulsamos sobre cualquiera de los enlaces de la primera columna del listado y nos aparecerá el detalle del libro seleccionado.

Desde esta pantalla podremos realizar las siguientes operaciones sobre el libro seleccionado:

- **Modificar.** Nos permite modificar todos los campos del libro salvo el código del libro, el cual utilizamos como clave primaria en la base de datos.

Para realizar esta operación, modificamos el campo o los campos que deseemos del libro y pulsamos el botón **modificar**. Seguidamente, el sistema nos redirigirá a la pantalla principal.

- **Borrar.** Si queremos eliminar este libro, simplemente pulsamos el botón **borrar** y el sistema borrará el registro de la base de datos correspondiente al libro seleccionado y nos redirigirá a la pantalla principal.
- **Volver al listado de libros.** Si no deseamos realizar ninguna de las anteriores operaciones, con el libro seleccionado, pulsamos el botón **Volver Lista** para redirigirnos al listado de libros.

5.2.1. Análisis de la aplicación

En este apartado mostraremos el diagrama de secuencia de la aplicación que hemos realizado como ejemplo. En dicho diagrama, se muestran los elementos que vamos a necesitar en nuestra aplicación, aunque, como veremos en más detalle a lo largo del capítulo, no aparecen todos los elementos que intervienen en la aplicación como, por ejemplo, los ficheros de propiedades o los ficheros XML de confirmación.

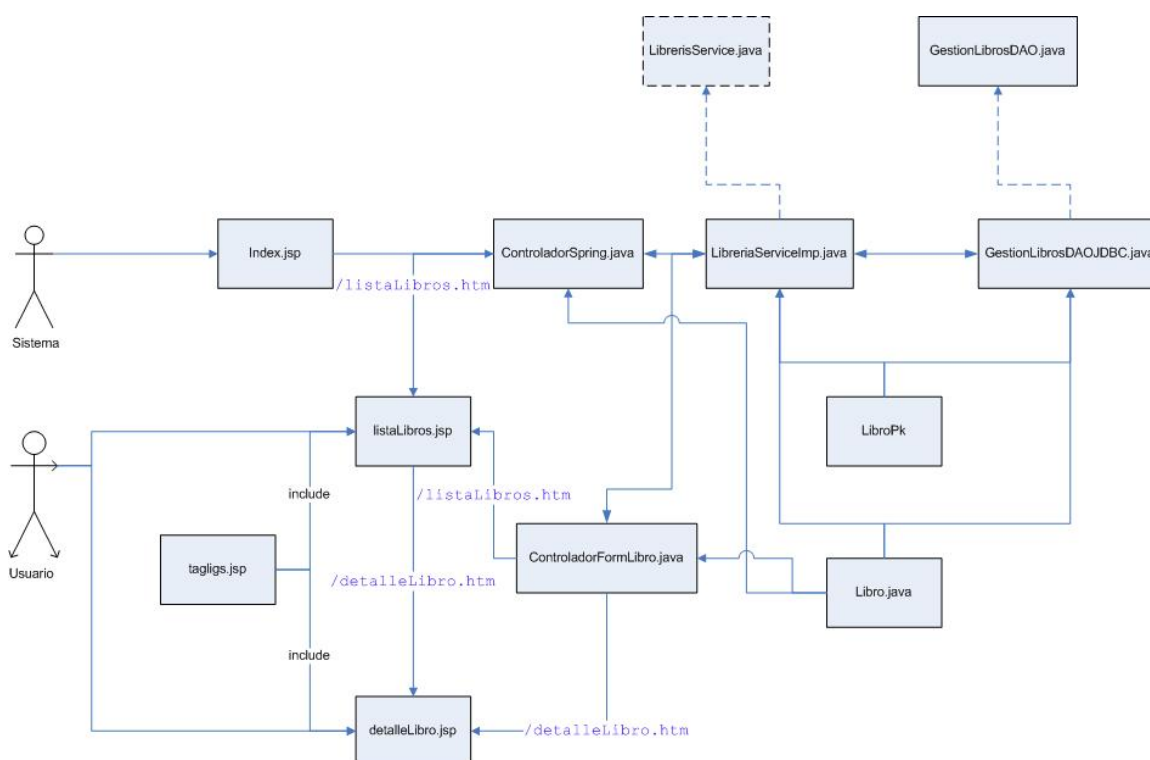


Diagrama UML de secuencia de la aplicación

Como se puede observar en el diagrama de secuencia anterior, hemos representado las diferentes capas que intervienen en la aplicación. A continuación, describiremos dichas capas y los componentes que se alojan en cada una de estas:

- **Capa de Presentación.** Normalmente, en esta capa se alojan los elementos visuales de la aplicación: las páginas **jsp**, **html** y **pdf** (en caso de generar informes), así como las clases que realizan la función de control del **patrón MVC**, es decir, **Taglibs.jsp**, **index.jsp**, **listaLibros.jsp**, **detalleLibro.jsp**, **ControladorSpring.java** y **ControladorFormLibro.java**.

En algunas referencias encontramos que los elementos que ejercen el control del **patrón MVC**, en nuestro caso **ControladorSpring.java** y **ControladorFormLibro.java**, no se alojan en esta capa sino en una nueva capa llamada **aplicación**.

- **Capa de Servicio.** En esta capa se alojan los componentes que gestionan la lógica de negocio de la aplicación.

En nuestro ejemplo, en esta capa tenemos la **interfaz LibreriaService.java** y la clase que implementa dicha interfaz **LibreriaServiceImpl.java**.

- **Capa DAO.** Aquí se alojan las operaciones entre la aplicación y la base de datos.

En el ejemplo que hemos realizado encontramos la interfaz **GestionLibrosDAO.java** y la clase que implementa dicha interfaz **GestionLibrosDAOJDBC.java**.

- **Modelo de datos.** No es una capa en sí, más bien son las clases java que representan el modelo de la base de datos. Dichas clases son utilizadas por todas las capas anteriormente mencionadas.

En nuestro ejemplo indicamos que las dos clases del Modelo de datos son:

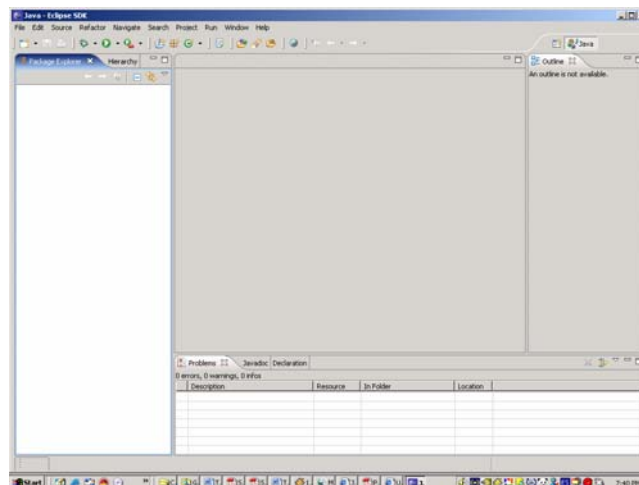
- **Libro.java.** Es una representación en java de todas las columnas de la tabla **librería**.
- **LibroPk.** Esta clase es una representación en java de las columnas que forman la clave primaria de la tabla **librería**. En nuestro caso, sólo contendrá un valor **codLibro**.

Finalmente, para realizar las uniones necesarias entre los componentes, crearemos el fichero de configuración de Spring, **springapp-servlet.xml**, y el archivo **web.xml**, necesario en cualquier aplicación web, para indicarnos cómo se debe de desplegar una aplicación.

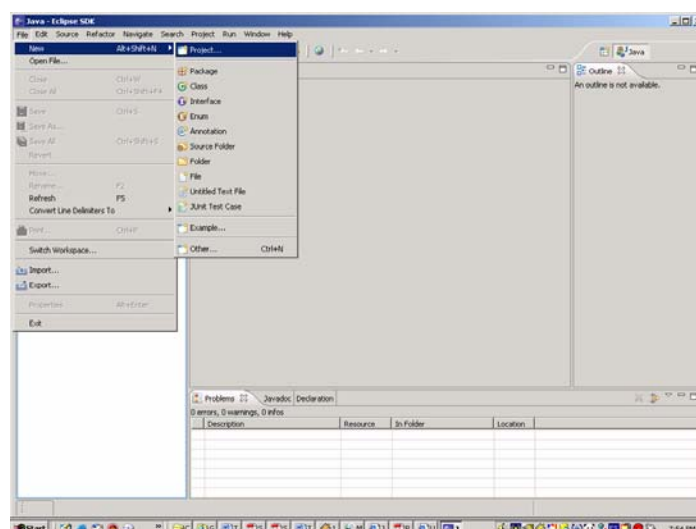
5.3. CREACIÓN DE UN NUEVO PROYECTO UTILIZANDO ECLIPSE WTP

Vamos a comenzar definiendo un nuevo proyecto y las propiedades del mismo paso a paso:

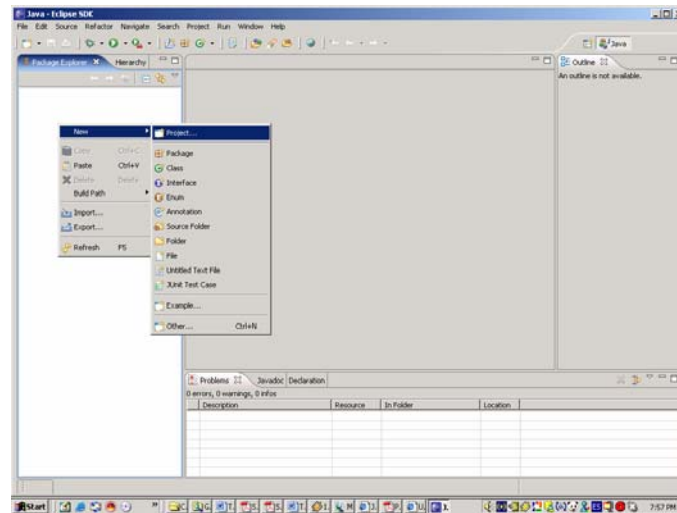
1. Arrancamos el eclipse:



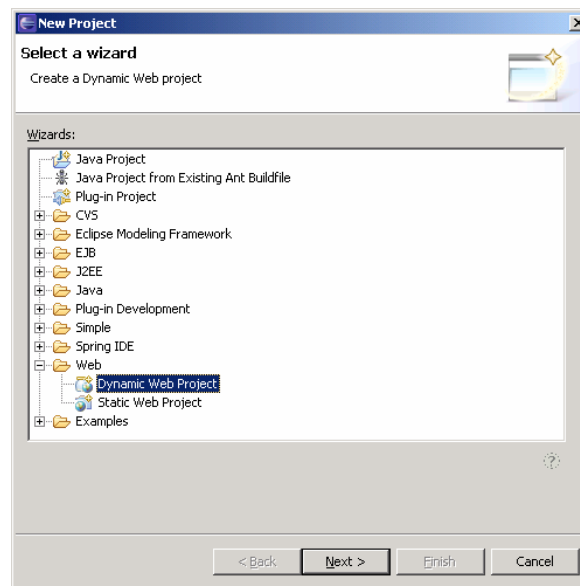
2. Creamos el proyecto. Para realizar esta operación, seguimos la siguiente ruta:
File→New→Project...



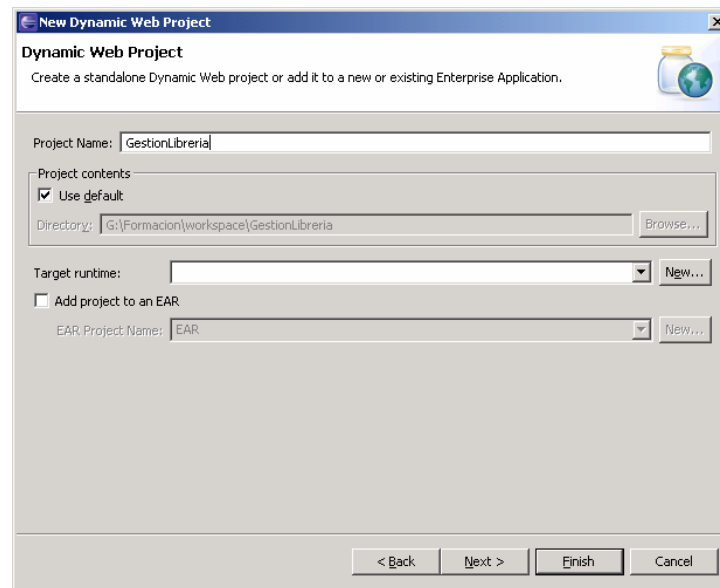
También es posible realizar esta operación, pulsando con el botón derecho en el **Explorador de paquetes** de la vista **Java** y seleccionando **New→Project...**



3. Seleccionamos la opción **Dynamic Web Project**, como muestra la siguiente captura, y pulsamos **Next**.

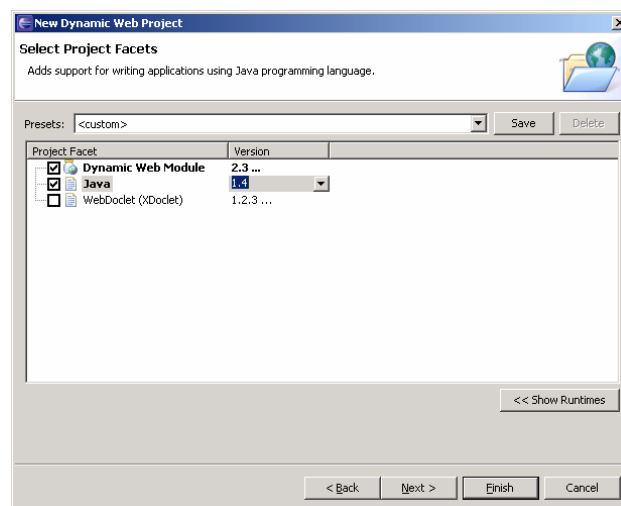


4. Seguidamente, introducimos el nombre del proyecto. Como nosotros vamos a realizar una gestión simple de una librería, hemos elegido el nombre **GestionLibreria** y pulsamos **Next**.

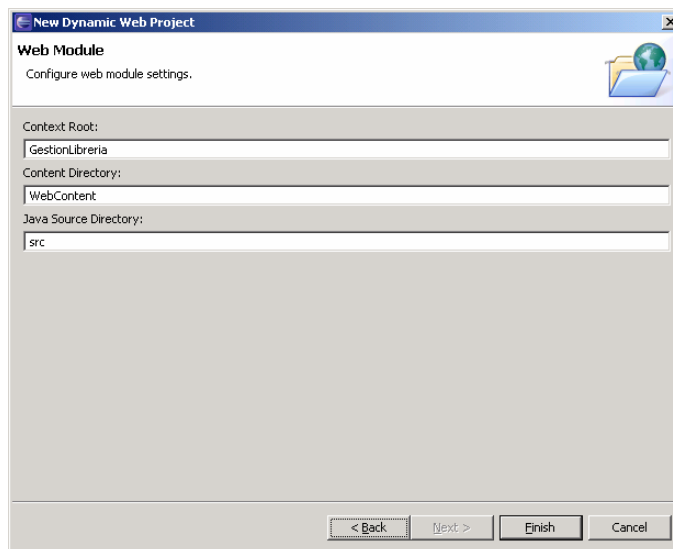


5. Elegimos las siguientes versiones de las propiedades de nuestro proyecto:

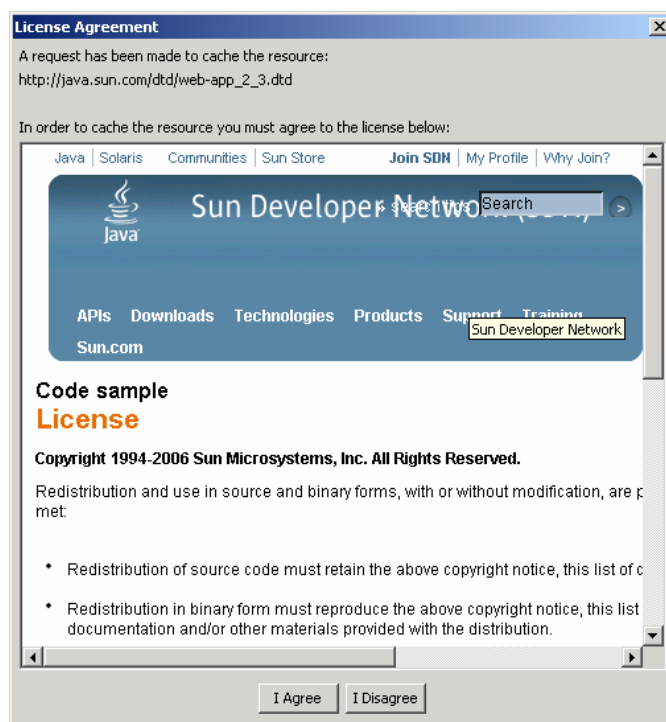
- **Dynamic web Module** → **2.3**
- **Java** → **1.4**



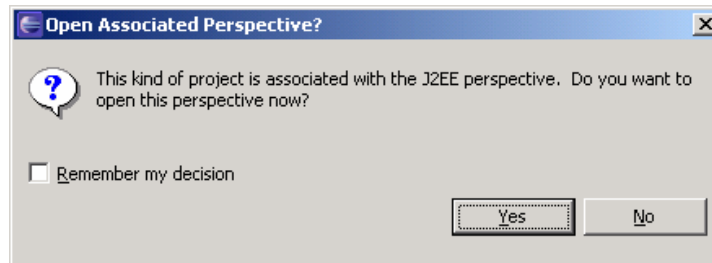
6. Pulsamos **Next**



7. Pulsamos **I Agree** para realizar la confirmación de la licencia para utilizar el **dtd**.



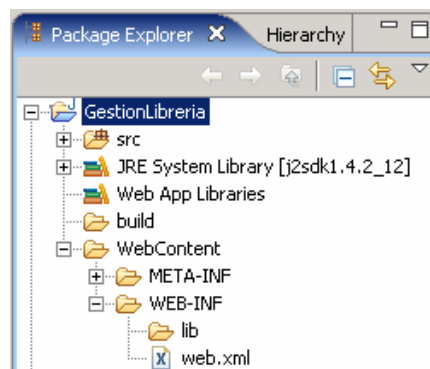
8. Por último, Eclipse nos pregunta si deseamos cambiar de perspectiva. Pulsaremos **No** y continuaremos trabajando con la perspectiva **Java**.



5.4. ESTRUCTURA DE LA APLICACIÓN

En este apartado explicaremos cómo crear la estructura de carpetas que contendrá nuestra aplicación. Dicha estructura es importante para las diversas fases del desarrollo y posterior despliegue en un servidor de nuestra aplicación.

A continuación, podemos observar la estructura de carpetas que hemos realizado en el anterior proceso de creación de un proyecto web dinámico.



Donde:

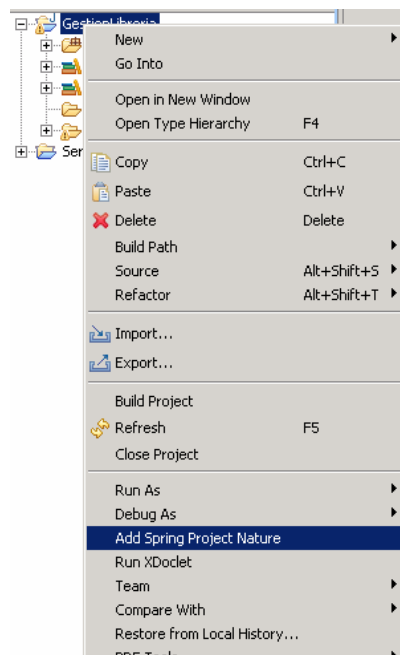
- **Src.** En esta carpeta es donde guardamos nuestros ficheros fuente de Java.
- **JRE System Library.** Se crea automáticamente y contiene los archivos **.jar** necesarios para un proyecto Java para la versión de **JRE** que hemos seleccionado en la creación de nuestro proyecto, en nuestro caso **jdk1.4.2_12**.

- **Web App Libraries.** En esta carpeta encontramos las librerías propias del servidor de aplicaciones, como **servlet.jar**.
- **build.** Esta carpeta se ha creado porque vamos a utilizar el despliegue de la aplicación en un servidor **Tomcat**. A lo largo del tema la eliminaremos, ya que, nuestra aplicación se puede desplegar en cualquier servidor que tengamos instalado.
- **WebContent.** Esta carpeta es la carpeta raíz del despliegue. En ella, se alojarán las páginas de seguridad que deseemos que sean visibles y contendrá la carpeta **WEB-INF**.
- **WEB-INF.** En ella reside la configuración de una aplicación web y encontramos las librerías que necesita la aplicación, las clases compiladas java y las páginas **jsp**, entre otros muchos componentes. Su función es servir para el despliegue de la aplicación.

5.4.1. Añadir al proyecto la naturaleza de proyecto Spring

Como recordaremos, en el primer tema, mostramos cómo instalar el plug-in **SpringIDE**, que proporciona soporte Spring a nuestras aplicaciones. Este plug-in lo podemos utilizar de dos formas:

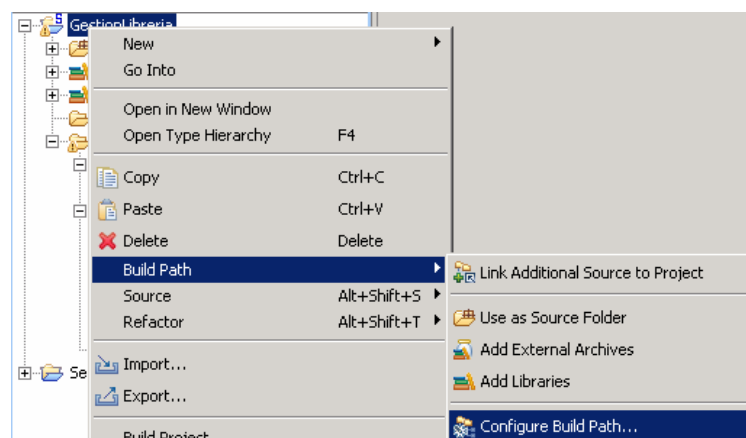
- **Al crear un nuevo proyecto.** Nos ofrece la posibilidad de crear un proyecto nuevo Spring.
- **Añadir Soporte Spring al proyecto web que hemos generado.** Ésta es la opción que hemos elegido. A continuación, describiremos cómo realizar esta operación:
 1. Nos posicionamos sobre el proyecto, en la vista de **Explorador de paquetes**, y pulsamos con el botón derecho del ratón.
 2. Seleccionamos **Add Spring project Nature**, como muestra la siguiente captura.



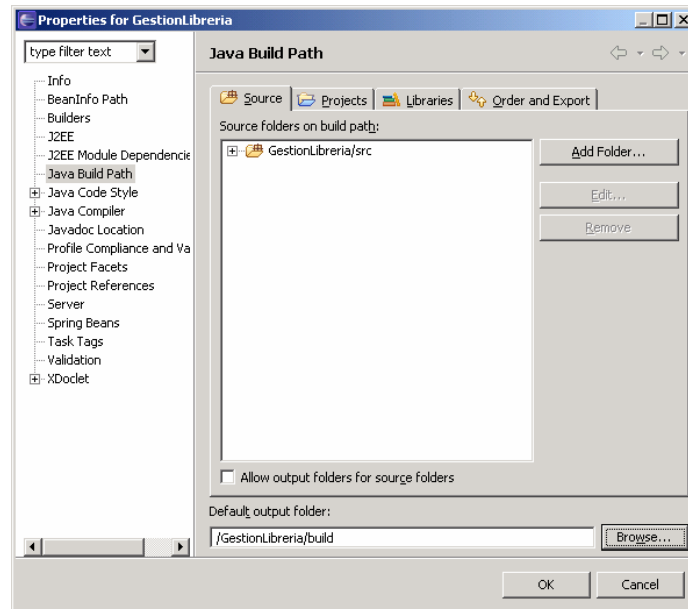
5.4.2. Configuración del entorno

Comenzaremos por establecer el **Path** de construcción de nuestra aplicación. Para realizar esta operación:

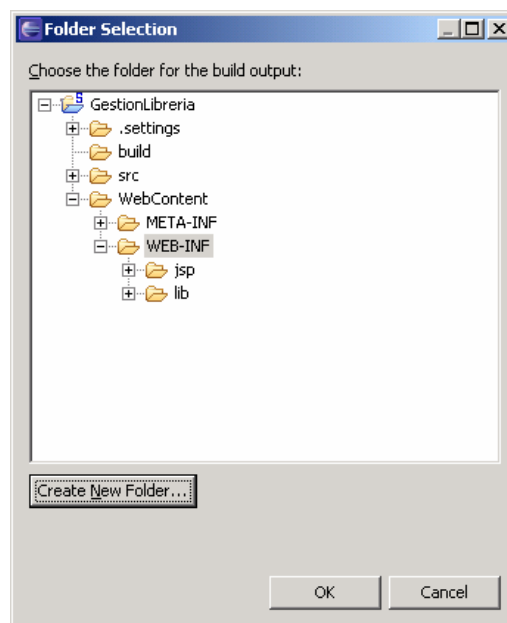
- Pulsamos con el botón derecho del ratón sobre el nombre del proyecto y seleccionamos el menú desplegable **Build Path** → **Configure Build Path...**



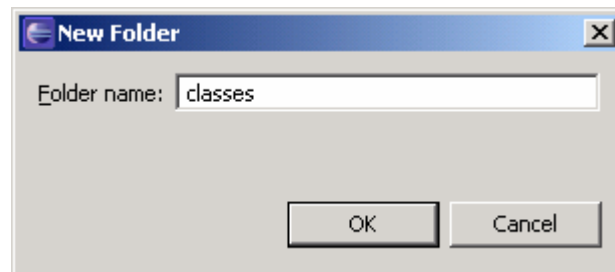
- Seguidamente, seleccionamos la pestaña **Source** y pulsamos sobre **Browse...** para modificar el fichero de salida.



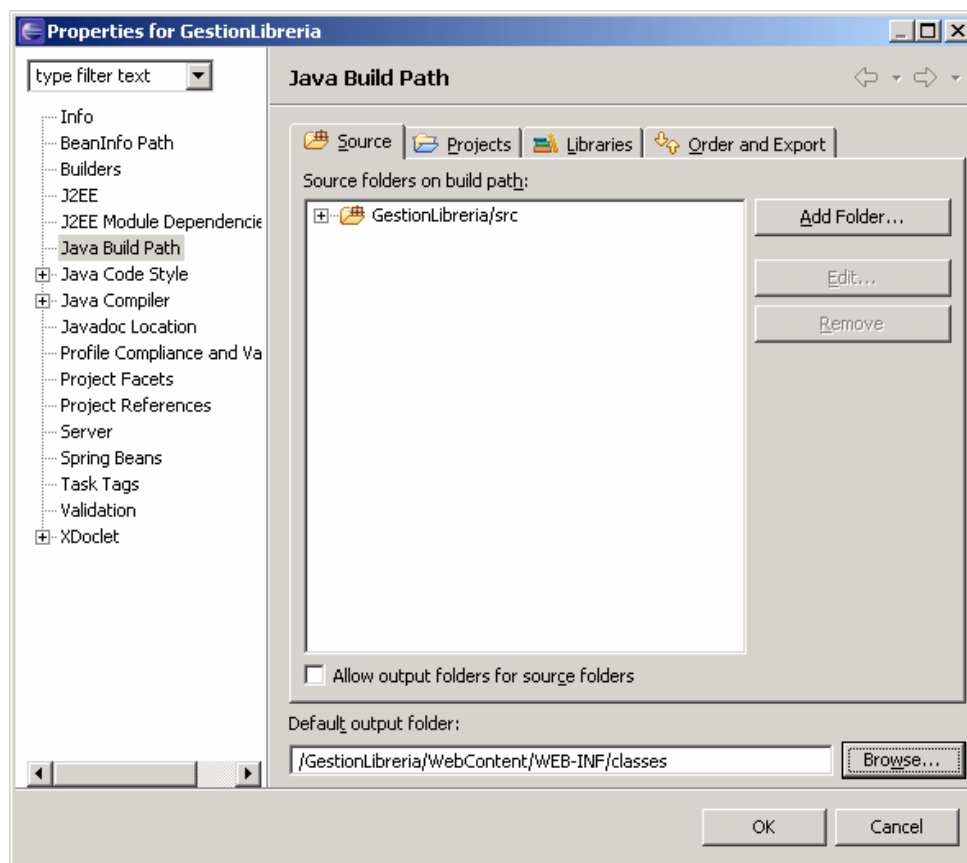
- Navegamos por las carpetas del proyecto hasta situarnos en **WEB-INF** y pulsamos sobre el botón **Create New Folder...**



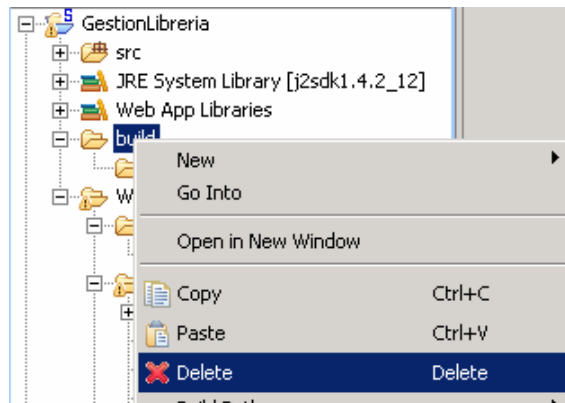
- Añadimos una nueva carpeta llamada **classes**, donde alojaremos las clases Java compiladas, y pulsaremos **OK**.



- A continuación, volvemos a la pantalla anterior, pero indicando que la nueva ruta de salida de las clases compiladas es: **/GestionLibreria/WebContent/WEB-INF/classes** y pulsamos **OK**.



- Por último, una vez hemos modificada la carpeta que alojará las clases compiladas, procedemos a eliminar la carpeta **build**.

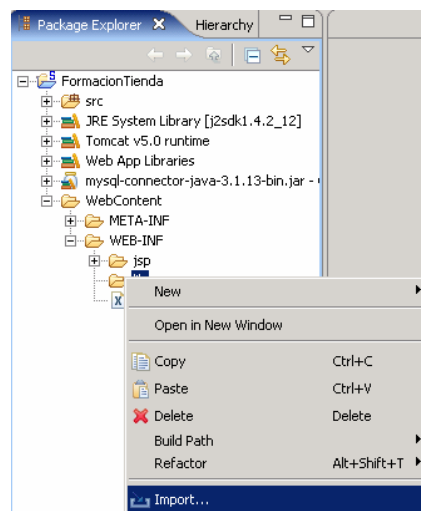


5.4.3. Bibliotecas necesarias en la aplicación

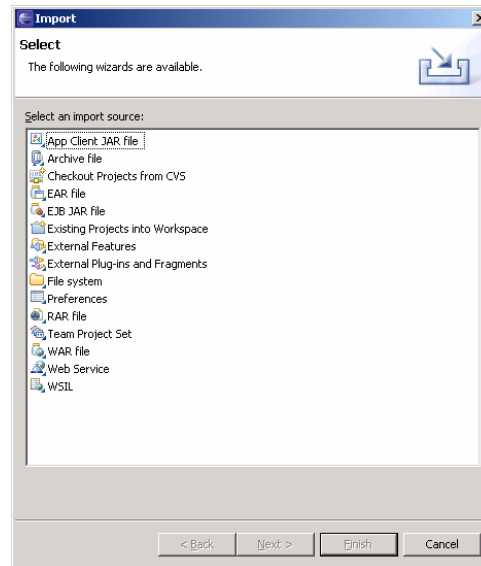
La aplicación que estamos realizando necesita añadir un conjunto de librerías que nos proporcionarán las funcionalidades de **Spring** y el acceso a la base de datos **MySQL** que hemos definido en el primer tema.

En primer lugar, comenzaremos por añadir las librerías de **Spring**. Para importar los archivos de biblioteca, seguiremos los siguientes pasos:

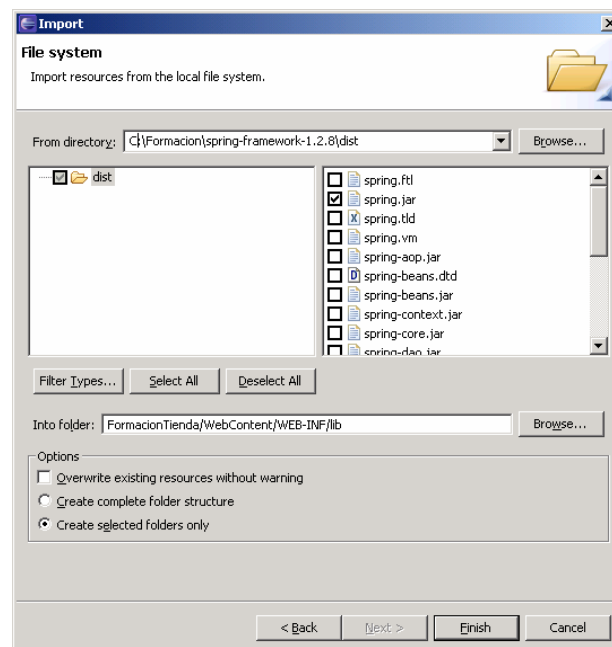
1. Nos situamos sobre la carpeta **WEB-INF/lib**, hacemos clic con el botón derecho del ratón y pulsamos la opción “**Import...**”.



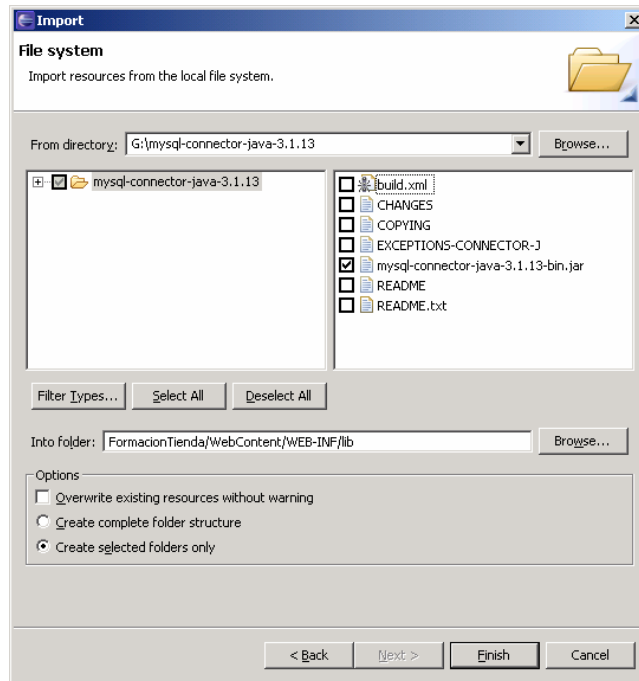
2. Seleccionamos **"File system"** y pulsamos el botón **"Next"**.



3. Pulsamos el botón **Browse...**, seleccionamos la ruta **C:\Formacion\spring-framework-1.2.8\dist** donde, como indicamos en el primer tema, se encuentran las librerías de **Spring**, y habilitamos la casilla de la librería **spring.jar**. Por último, hacemos clic en **"Finish"**.



4. Finalmente, realizamos la misma operación para añadir las librerías del **conector JDBC de MySQL**.

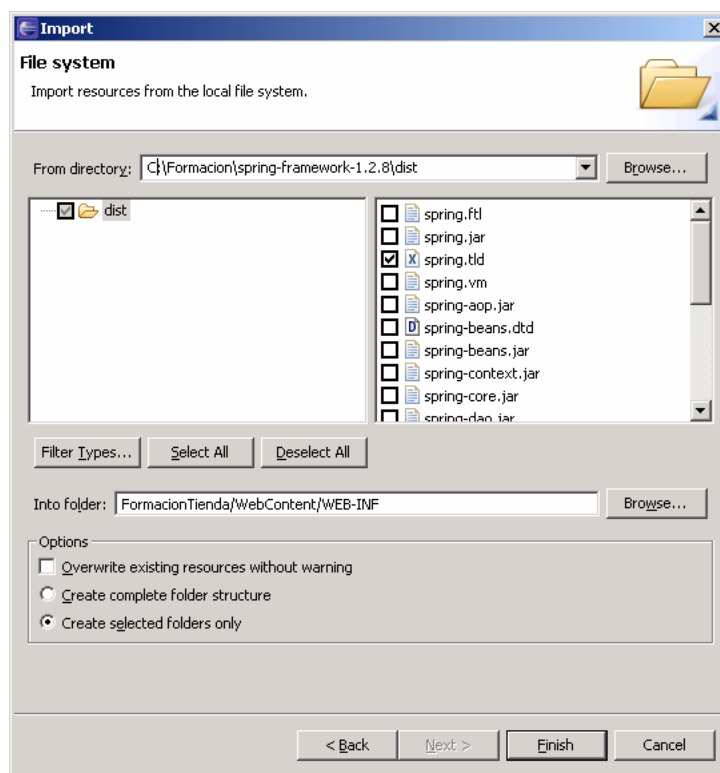


5.4.4. Spring.tld

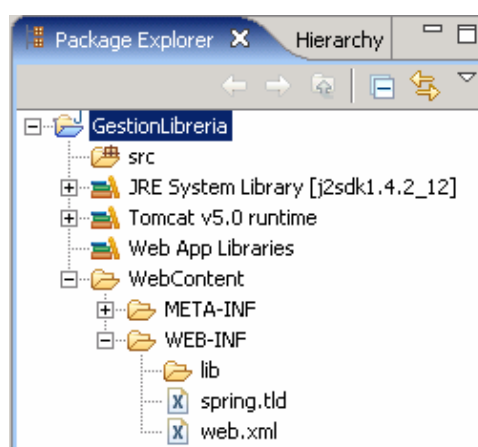
Una vez importadas las librerías, importaremos también los archivos descriptores de etiquetas de **Spring** para hacer uso de éstas en nuestras páginas **JSP's**.

Para ello, tenemos que seguir los siguientes pasos:

1. Nos situamos sobre la carpeta **WEB-INF**, hacemos clic con el botón derecho del ratón y pulsamos la opción **Import...**, de la misma forma que realizamos para añadir los **lib** de Spring.
2. Seguidamente, seleccionamos "**File system**" y pulsamos el botón "**Next**".
3. Por último, pulsamos el botón **Browse...**, seleccionamos la ruta **C:\Formacion\spring-framework-1.2.8\dist**, habilitamos la casilla de la librería **spring.tld** y hacemos clic en "**Finish**".



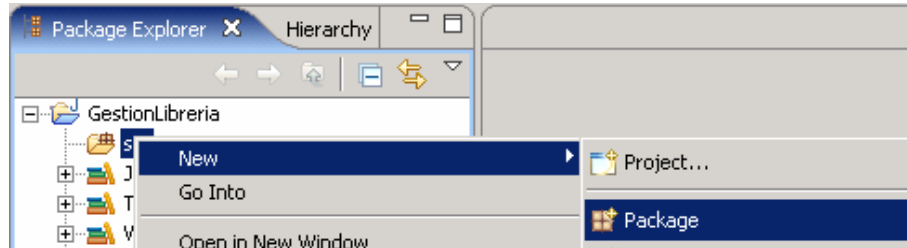
Una vez finalizados todos los pasos anteriores, obtendremos la estructura de carpetas que se muestra a continuación, así como la carpeta **GestionLibreria→WebContent→WEB-INF→classes** que está oculta. Ésta sería la carpeta base de la aplicación, a partir de la cual desarrollaremos lo lógico mostrada en el diagrama de secuencias del tercer punto de este tema.



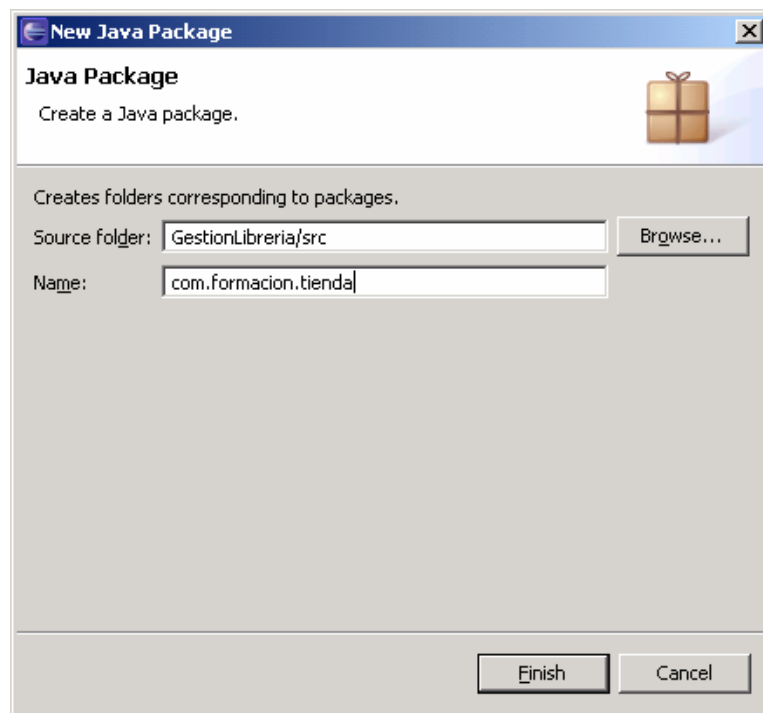
Por otro lado, una vez tenemos la estructura base del proyecto, comenzaremos a generar las clases que vamos a utilizar.

Para ello, crearemos una estructura de paquetes en la carpeta **src**, haciendo que todas nuestras clases cuelguen de ésta. Para realizar esta acción, seguiremos los siguientes pasos:

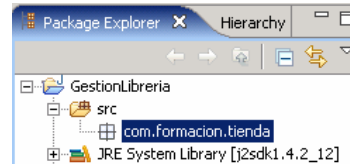
1. Seleccionamos la carpeta **src**, pulsamos con el botón derecho del ratón y seleccionamos **New→Package**, como muestra la siguiente captura.



2. Insertamos el nombre de nuestro paquete **com.formacion.tienda**



3. Como resultado obtenemos la siguiente estructura:



5.4.5. Clases del modelo

Las **clases del modelo** son utilizadas para transferir la información entre las diversas capas de nuestra aplicación. Por este motivo, es prioritaria su creación.

Lo primero que haremos será crear un nuevo paquete llamado **com.formacion.tienda.dto** donde alojaremos las clases de modelo.

Una vez creado el paquete, lo seleccionaremos y pulsaremos con el botón derecho **New→Class**. Una vez realizado esto, nos aparecerá una pantalla donde nos solicitan el nombre de la clase, escribimos **Libro**, y pulsamos **Finish**.

Esta clase representa la estructura almacenada en la base de datos que realizamos en el primer tema. Va a constar únicamente de sus cuatro atributos (**codLibro**, **descripción**, **precio** y **autor**), junto con sus métodos **get's** y **set's**, ya que vamos a tener una clase propia para la lógica de negocio. El código del **JavaBean** es el que se muestra a continuación:

```
package com.formacion.tienda.dto;
public class Libro {
    private int codLibro;
    private String descripcion;
    private int precio;
    private String autor;

    /**
     * @return
     */
    public int getPrecio() {
        return precio;
    }
    /**
```

```

    * @param precio
    */
    public void setPrecio(int precio) {
        this.precio = precio;
    }
    /**
    * @return
    */
    public String getDescripcion() {
        return descripcion;
    }
    /**
    * @param descripcionProducto
    */
    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }
    /**
    * Metodo que devuelve el código del producto
    * @return int, codigo del producto
    */
    public int getCodLibro() {
        return codLibro;
    }
    /**
    * Metodo que proporciona el código del producto
    * @param codigoProducto
    */
    public void setCodLibro(int codLibro) {
        this.codLibro = codLibro;
    }
    public String getAutor() {
        return autor;
    }
    public void setAutor(String autor) {
        this.autor = autor;
    }
}

```

Libro.java

En el mismo paquete, crearemos otra clase de modelo llamada **LibroPk**, que se corresponderá con un **JavaBean**, de un único campo **codLibro**, junto con su método **get** y **set**. Esta clase se utilizará para obtener el detalle de un objeto **Libro** y coincide con el campo clave primaria de la tabla **librería**.

```
package com.formacion.tienda.dto;

public class LibroPk {
    private int codLibro;

    public int getCodLibro() {
        return codLibro;
    }

    public void setCodLibro(int codLibro) {
        this.codLibro = codLibro;
    }
}
```

LibroPk.java

5.4.6. Ficheros de propiedades de la aplicación

Otra acción que debemos realizar es la definición de dos ficheros de propiedades, que se suelen alojar en la carpeta **src**, en los proyectos web. Estos ficheros son **resourceBundle.properties** y **log4j.properties**, los cuales describiremos en los siguientes apartados.

5.4.6.1. **resourceBundle.properties**

Este fichero, también llamado **I18N** (en inglés, **internationalization** está compuesta de dieciocho letras), se utiliza para conseguir la **internacionalización** de las aplicaciones. Es decir, en este fichero de propiedades, se inician los mensajes de texto de la aplicación que utilizamos desde nuestras **JSP**'s.

```
title=Gestión Librería
heading=Listado de libros
Libro.borrado=Se ha eliminado:
```

resourceBundle.properties

5.4.6.2. log4j.properties

Fichero de propiedades que utiliza el **log4j.jar** de Apache. Es la forma más fácil y normal que encontraremos para realizar el **log** de la aplicación.

Aunque este fichero de propiedades viene con la configuración por defecto de Apache, debemos indicar que es muy potente y que, si se configura adecuadamente, es muy útil para consultar errores o fallos en la lógica de nuestra aplicación.

```
# For JBoss: Avoid to setup Log4J outside $JBOSS_HOME/server/default/deploy/log4j.xml!
# For all other servers: Comment out the Log4J listener in web.xml to activate
Log4J.log4j.rootLogger=INFO, stdout, logfile
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - <%m>%n
log4j.appender.logfile=org.apache.log4j.RollingFileAppender
log4j.appender.logfile.File=/Users/trisberg/jakarta-tomcat-
5.0.28/logs/springapp.loglog4j.appender.logfile.MaxFileSize=512KB
# Keep three backup files.
log4j.appender.logfile.MaxBackupIndex=3
# Pattern to output: date priority [category] - message
log4j.appender.logfile.layout=org.apache.log4j.PatternLayout
log4j.appender.logfile.layout.ConversionPattern=%d %p [%c] - %m%n
```

Log4Java.properties

Una vez tenemos las clases del modelo y la definición de ficheros de propiedades generales de la aplicación, explicaremos la aplicación siguiendo su estructura de capas.

5.4.7. Capa de presentación

La capa de presentación es la más cercana al usuario y está compuesta por **páginas jsp, html, imágenes, hojas de estilo (CSS), javaScript**. En nuestro caso, nosotros hemos alojado en esta capa las funciones que realizar la función **Controlador** del patrón **MVC**.

En los siguientes apartados, crearemos todos los componentes de esta capa, indicando claramente su funcionalidad y dónde se alojan dentro de la estructura de carpetas.

5.4.7.1. Páginas JSP de la aplicación

Comenzaremos creando en el directorio **WEB-INF** una carpeta que llamaremos **jsp**, donde alojaremos la mayoría de nuestras **páginas jsp**.

Para ello, seleccionaremos la carpeta **jsp** que acabamos de crear, y seguiremos los siguientes pasos para crear las páginas JSP:

- En primer lugar, pulsamos con el botón derecho del ratón y seleccionamos **New→Other**.
- Seguidamente, nos aparecerá una pantalla donde seleccionaremos **Web→JSP** y pulsaremos **Next**.
- Finalmente, daremos nombre a la **página jsp**, en nuestro caso **taglibs.jsp**, y pulsaremos **Finish**.

```
<%@ page session="false"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
<%@ taglib prefix="spring" uri="/spring" %>
```

taglibs.jsp

Esta página web recoge las librerías que vamos a utilizar en la aplicación y será incluida por todas las demás páginas web que desarrollaremos. Dichas páginas son las que se muestran seguidamente:

1. **Index.jsp** es la página inicial de la aplicación y se aloja directamente en **WebContent**.

Su única misión es redirigir a la página principal (en nuestro caso, **listaLibros.jsp**), aunque primero accede al **ControladorSpring.java** que obtiene el listado de libros. Como resultado, nos redirigirá a **/listaLibros.htm**. Esta transformación de **listaLibros.jsp** a **listaLibros.htm** la realizamos en el fichero de configuración de **Spring springapp-servlet.xml** (lo iremos viendo a lo largo del ejemplo).

```
<%@ include file="/WEB-INF/jsp/taglibs.jsp"%>
<c:redirect url="/listaLibros.htm" />
```

Index.jsp

2. **listaLibros.jsp**. Se trata de la página que utilizamos para mostrar el listado de libros, parte central de nuestra aplicación de ejemplo.

En esta página, debemos remarcar varios aspectos:

- Utilización de la etiqueta de **tags fmt**, principalmente, para mostrar los mensajes del **resourceBundle.properties**. Ejemplo:

```
<fmt:message key="title"/> -> muestra Gestión Librería
```

- Utilización de la etiqueta de **tags core** para mostrar el resultado de las variables y realizar la lógica de presentación de la página jsp:

```
<c:out value="${libro.descripcion}"/>
```

- Por último, veremos como **Spring** gestiona las llamadas utilizando el método **get** al analizar la clase **ControladorSpring.java**, a través de botones o enlaces:

```
<button onclick="location.href='detalleLibro.htm'">Añadir Libro</button>
```

```
<%@ include file="/WEB-INF/jsp/taglibs.jsp"%>
<html>
<head>
    <title><fmt:message key="title"/></title>

</head>
<body>
    <center><h3><fmt:message key="heading"/></h3></center>

    <table align="center">
        <tr bgcolor="#CCFFCC">
            <td>&nbsp;</td><td>Descripción</td><td>Precio</td><td>Autor</td>
        </tr>

        <c:forEach items="${model.libros}" var="libro" varStatus="contador">
```



```

        <c:choose>
        <c:when test="${contador.count % 2 == 0}"><tr
bgcolor="#99CCFF"></c:when>
        <c:otherwise><tr bgcolor="#FFFF99"></c:otherwise>
        </c:choose>
        <td><a href="detalleLibro.htm?codLibro=<c:out
value="${ libro.codLibro}"/>"><c:out value="${ libro.codLibro}"/></a></td>
        <td><c:out value="${ libro.descripcion}"/></td>
        <td><c:out value="${ libro.precio}"/> &euro; </td>
        <td><c:out value="${ libro.autor}"/></td>
        </tr>
    </c:forEach>
    <tr>
    <td colspan="4" align="right"><button
onclick="location.href='detalleLibro.htm'">Añadir Libro</button></td>
    </tr>
</table>

</body>
</html>

```

listaLibros.jsp

3. **detalleLibro.jsp**. Página en la que realizaremos las operaciones de alta modificación y eliminación de libros.

- En esta página veremos como Spring gestiona las llamadas utilizando el método **post** al analizar la clase **ControladorFormSpring.java** a través del **form**:

```

<form method="post" action="<c:url value="/detalleLibro.htm"/>"
onsubmit="return onFormSubmit(this)">

```

- Utilización de las **tags** propias de **Spring**:

```

<spring:bind path="libro.*">

```

```
<%@ include file="/WEB-INF/jsp/taglibs.jsp"%>
<title>Detalle Libro</title>
<spring:bind path="libro.*">
  <c:if test="${not empty status.errorMessages}">
    <div class="error">
      <c:forEach var="error" items="${status.errorMessages}">
        <c:out value="${error}" escapeXml="false"/><br />
      </c:forEach>
    </div>
  </c:if>
</spring:bind>
<H1>Detalle libro</H1>
<form method="post" action="<c:url value="/detalleLibro.htm"/>"
onsubmit="return onFormSubmit(this)">
<spring:bind path="libro.codLibro">
<input type="hidden" name="codLibro" value="<c:out value="${status.value}"/>">
</spring:bind>
<table bgcolor="#334B76;">
<tr>
<th>Descripción: </th>
<td>
<spring:bind path="libro.descripcion">
<input type="text" name="descripcion" value="<c:out value="${status.value}"/>">
  <span class="fieldError"><c:out value="${status.errorMessage}"/></span>
</spring:bind>
</td>
</tr>
<tr>
<th>Precio: </th>
<td>
<spring:bind path="libro.precio">
<input type="text" name="precio" value="<c:out value="${status.value}"/>">
  <span class="fieldError"><c:out value="${status.errorMessage}"/></span>
</spring:bind>
</td>
</tr>
<tr>
<th>Autor: </th>
<td>
```

```
<spring:bind path="libro.autor">
<input type="text" name="autor" value="<c:out value="\${status.value}"/>">
  <span class="fieldError"><c:out value="\${status.errorMessage}"/></span>
</spring:bind>
</td>
</tr>
<tr>
<td colspan="2" align="center">
<c:if test="\${empty param.codLibro}">
<input type="submit" class="button" name="save" value="Insertar"/>
</c:if>
<c:if test="\${not empty param.codLibro}">
<input type="submit" class="button" name="update" value="Modificar"/>
<input type="submit" class="button" name="delete" value="Borrar"/>
</c:if>
<input type="submit" class="button" name="listado" value="Volver Lista"/>
</td>
</tr>
</table>
</form>
<html:javascript formName="libro"/>
```

detalleLibro.jsp

5.4.7.2. Clase de control utilizando el patrón Spring MVC

En la aplicación de ejemplo hemos querido aplicar lo aprendido en el tema 4 de este manual. Por este motivo, en este apartado, mostraremos una primera clase que implementa la interfaz **Controller** y, posteriormente, utilizaremos otra clase que necesita recoger los parámetros de formulario, para lo que extenderemos la clase **SimpleFormController**.

```
package com.formacion.tienda.web;

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

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

```
import javax.servlet.http.HttpServletResponse;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

import com.formacion.tienda.svc.LibreriaService;

public class ControladorSpring implements Controller {

    /** Logger for this class and subclasses */

    protected final Log logger = LogFactory.getLog(getClass());
    private LibreriaService libreriaService;

    public LibreriaService getLibreriaService() {
        return libreriaService;
    }

    public void setLibreriaService(LibreriaService libreriaService) {
        this.libreriaService = libreriaService;
    }

    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        Map miModelo = new HashMap();
        miModelo.put("libros", getLibreriaService().getLibros());

        return new ModelAndView("listaLibros", "model", miModelo);
    }
}
```

Controlador Spring.java

```
package com.formacion.tienda.web;

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

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.SimpleFormController;
import org.springframework.web.servlet.view.RedirectView;

import com.formacion.tienda.dto.Libro;
import com.formacion.tienda.dto.LibroPk;
import com.formacion.tienda.svc.LibreriaService;

public class ControladorFormLibro extends SimpleFormController {

    protected final Log logger = LogFactory.getLog(getClass());
    private LibreriaService libreriaService;

    public LibreriaService getLibreriaService() {
        return libreriaService;
    }

    public void setLibreriaService(LibreriaService libreriaService) {
        this.libreriaService = libreriaService;
    }

    /* (non-Javadoc)
     * @see
     org.springframework.web.servlet.mvc.SimpleFormController#onSubmit(javax.servlet.http.HttpS
     rvletRequest, javax.servlet.http.HttpServletResponse, java.lang.Object,
     org.springframework.validation.BindException)
     */
    public ModelAndView onSubmit(HttpServletRequest request,
                                HttpServletResponse response,
                                Object command, BindException errors)
    
```

```

throws Exception {

    Libro libro = (Libro) command;
    if (request.getParameter("delete") != null) {
        libreriaService.borrarLibro(libro);
        request.getSession().setAttribute("message",
        getMessageSourceAccessor()
        .getMessage("Libro.borrado",
        new Object[] { libro.getCodLibro() +
        ' ' + libro.getDescripcion() + ' ' + libro.getPrecio() + ' ' +
        libro.getAutor()}));
    } else if (request.getParameter("save") != null) {
        libreriaService.guardarLibro(libro);

    } else if (request.getParameter("update") != null) {
        libreriaService.modificarLibro(libro);
    } else if (request.getParameter("listado") != null) {
        return new ModelAndView(new RedirectView("listaLibros.htm"));
    }

    return new ModelAndView(getSuccessView());
}

protected Object formBackingObject(HttpServletRequest request)
throws ServletException {
    String codLibro = request.getParameter("codLibro");
    if ((codLibro != null) && !codLibro.equals("")) {
        LibroPk pk=new LibroPk();
        String strCodLibro=request.getParameter("codLibro");
        pk.setCodLibro(Integer.parseInt(strCodLibro));
        return libreriaService.getLibro(pk);
    } else {
        return new Libro();
    }
}
}

```

Controlador FormSpring.java

5.4.8. Capa de servicio

La **capa de servicio** es la que establece la lógica e indica las operaciones de negocio de la aplicación.

En nuestro ejemplo, al tratarse de una práctica sencilla, actuará como una simple **fachada** entre la capa de presentación y la capa de persistencia, aunque es una capa muy importante en aplicaciones más complejas. Por ejemplo, en el caso de que subiéramos al servidor un fichero con un listado de libros para cargar en la base de datos, la capa de servicio sería la encargada de validar el tamaño y en formato del fichero.

Las **operaciones** de negocio de la aplicación que podemos encontrar en esta capa son:

1. **LibreriaService**. Interfaz que indica los métodos que realizará la capa de servicio.

```
package com.formacion.tienda.svc;

import java.util.List;

import com.formacion.tienda.dto.Libro;
import com.formacion.tienda.dto.LibroPk;

public interface LibreriaService {
    public List getLibros();
    public Libro getLibro(LibroPk libPk);
    public void guardarLibro(Libro libro);
    public void modificarLibro(Libro libro);
    public void borrarLibro(Libro libro);
}
```

LibreriaService.java

2. **LibreriaServiceImpl**. Clase que implementa los métodos definidos en la interfaz **LibreriaService**.

```
package com.formacion.tienda.svc;

import java.util.List;

import com.formacion.tienda.dao.GestionLibrosDAO;
import com.formacion.tienda.dto.Libro;
import com.formacion.tienda.dto.LibroPk;

public class LibreriaServiceImpl implements LibreriaService {

    private GestionLibrosDAO gld;

    private List listaLibros;

    public void setGestionLibrosDao(GestionLibrosDAO gld) {
        this.gld = gld;
    }

    public List getLibros() {
        listaLibros=gld.getListaLibros();
        return listaLibros;
    }

    public Libro getLibro(LibroPk libPk) {
        return (Libro)gld.getLibro(libPk);
    }

    public void guardarLibro(Libro libro) {
        gld.guardarLibro(libro);
    }

    public void modificarLibro(Libro libro) {
        gld.modificarLibro(libro);
    }
}
```



```

    }

    public void borrarLibro(Libro libro) {
        gld.borrarLibro(libro);
    }
}

```

LibreriaServiceImpl.java

5.4.9. Capa de persistencia

Esta capa agrupa todos los componentes cuya misión es permitir almacenar de forma persistente la información de la aplicación en un repositorio. Normalmente, este repositorio es una base de datos, como en el ejemplo que estamos desarrollando.

Los **componentes** que constituyen la capa de persistencia son los que se describen seguidamente:

1. **Dao.properties.** Fichero de propiedades de la base de datos.

Este fichero lo cargaremos en el fichero de configuración de **Spring** y, con los datos cargados en el contexto, estableceremos las propiedades del **DataSource**.

```

db.driver=com.mysql.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/libreria
db.username=root
db.password=

```

Dao.properties

2. **GestionLibrosDAO.** Interfaz en la que indicamos todas las operaciones que debemos realizar sobre la tabla **libros** en la base de datos y que implementaremos en la clase **GestionLibrosDAOJDBC**.

```

package com.formacion.tienda.dao;

import java.util.List;

import com.formacion.tienda.dto.Libro;
import com.formacion.tienda.dto.LibroPk;

```

```
public interface GestionLibrosDAO {
    public List getListaLibros();
    public Libro getLibro(LibroPk libroPk);
    public void guardarLibro(Libro libro);
    public void modificarLibro(Libro libro);
    public void borrarLibro(Libro libro);
}
```

GestionLibrosDAO.java

3. **GestionLibrosDAOJDBC**. Clase que implementa los métodos definidos en la interfaz **GestionLibrosDAO**.

En esta clase hemos querido mostrar todo lo aprendido en el tercer tema del manual. Por esta razón, hemos realizado algunos métodos utilizando Spring sólo para obtener el **DataSource** por **IoC**. Éstos son:

- public Libro getLibro(LibroPk libroPk)
- public void guardarLibro(Libro libro)
- public void borrarLibro(Libro libro)

No obstante, hemos utilizado la clase que nos proporciona Spring **JdbcTemplate**, en los siguientes métodos:

- public List getListaLibros()
- public String getDescripcionLibro(LibroPk libroPk)
- public void modificarLibro(Libro libro)

```
package com.formacion.tienda.dao;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
```

```
import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.jdbc.core.JdbcTemplate;

import com.formacion.tienda.dto.Libro;
import com.formacion.tienda.dto.LibroPk;

public class GestionLibrosDAOJDBC implements GestionLibrosDAO {
    /** Logger for this class and subclasses */
    protected final Log log = LogFactory.getLog(getClass());
    private DataSource ds;
    Connection con;
    public void setDataSource(DataSource ds) {
        this.ds = ds;
    }
    public DataSource getDs() {
        return ds;
    }

    public List getListLibros() {
        List listaLibros=new ArrayList();
        log.info("Entro en obtener libros");
        JdbcTemplate jt = new JdbcTemplate(ds);
        String sql="select * from libreria";
        listaLibros=jt.queryForList(sql);
        return listaLibros;
    }

    /**
     * Método que devuelve la descripcion del libro utilizando la clase JdbcTemplate
     */

    public String getDescripcionLibro(LibroPk libroPk){
        JdbcTemplate jt=new JdbcTemplate(ds);
        return (String)jt.queryForObject("select descripcion from libreria where codLibro = ?",new
```

```

Object[] {new Integer(libroPk.getCodLibro())},String.class);
    }

    public Libro getLibro(LibroPk libroPk) {
        Libro libro=new Libro();

        try{
            con=ds.getConnection();
            Statement st=con.createStatement();
            ResultSet rs=st.executeQuery("select * from libreria where
codLibro="+libroPk.getCodLibro());
            while (rs.next()){

                libro.setCodLibro(rs.getInt(1));
                libro.setDescripcion(rs.getString(2));
                libro.setPrecio(rs.getInt(3));
                libro.setAutor(rs.getString(4));

            }
        }catch(SQLException ex){
            ex.printStackTrace();
        }
        return libro;
    }

    public void guardarLibro(Libro libro) {
        try{
            con=ds.getConnection();
            Statement st=con.createStatement();
            String sql="insert into libreria
values("+Contador()+"','"+libro.getDescripcion()+"','"+libro.getPrecio()+"','"+libro.getAutor()+"');";
            st.execute(sql);
        }catch(SQLException ex){
            ex.printStackTrace();
        }

    }

    public void modificarLibro(Libro libro) {

```

```

        JdbcTemplate jt=new JdbcTemplate(ds);
        jt.update("Update libreria set descripcion= ? , precio= ? ,autor= ? where codLibro= ? ",
            new Object[] { libro.getDescripcion(),Integer.toString(libro.getPrecio()),
libro.getAutor() , Integer.toString(libro.getCodLibro())});

    }

    public void borrarLibro(Libro libro) {
        try{
            con=ds.getConnection();
            Statement st=con.createStatement();
            st.executeUpdate("delete from libreria where
codLibro="+libro.getCodLibro());
        }catch(SQLException ex){
            ex.printStackTrace();
        }

    }

    private int Contador(){
        int contador=0;
        try{
            con=ds.getConnection();
            Statement st=con.createStatement();
            ResultSet rs=st.executeQuery("select codLibro from libreria ORDER BY
codLibro DESC");

            rs.next();
            contador=rs.getInt(1);
            contador+=1;
        }catch(SQLException ex){
            ex.printStackTrace();
        }

        return contador;

    }

}

```

GestionLibrosDAOJDBC.java

5.4.10. Descripción del despliegue de la aplicación

En el fichero **web.xml**, indicaremos como se desplegará nuestra aplicación.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="Libreria">
    <display-name>GestionLibreria</display-name>
    <servlet>
        <servlet-name>springapp</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>springapp</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <taglib>

        <taglib-uri>/spring</taglib-uri>
        <taglib-location>/WEB-INF/spring.tld</taglib-location>
    </taglib>
</web-app>
```

web.xml

5.4.11. Fichero de definición de los beans

A continuación, mostraremos cómo hemos definido todos los beans que hemos utilizado en nuestra aplicación de ejemplo.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<!-- - Application context definition for "springapp" DispatcherServlet. -->

<beans>

    <bean id="controladorSpring" class="com.formacion.tienda.web.ControladorSpring" >

        <property name="libreriaService">
            <ref bean="libreriaSvc"/>
        </property>

    </bean>

    <bean id="controladorFormLibro"
class="com.formacion.tienda.web.ControladorFormLibro">

        <property name="commandName">
            <value>libro</value>
        </property>
        <property name="commandClass">
            <value>com.formacion.tienda.dto.Libro</value>
        </property>
        <property name="formView">
            <value>detalleLibro</value>
        </property>
        <property name="successView">
            <value>redirect: listaLibros.htm</value>
        </property>
        <property name="libreriaService">
            <ref bean="libreriaSvc"/>
        </property>

    </bean>

</beans>
```

```

    </bean>

    <bean id="urlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

        <property name="mappings">
            <props>
                <prop key="/listaLibros.htm">controladorSpring</prop>
                <prop key="/detalleLibro.htm">controladorFormLibro</prop>
            </props>
        </property>

    </bean>

    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">

        <property name="viewClass">
            <value>org.springframework.web.servlet.view.JstlView</value>
        </property>
        <property name="prefix">
            <value>/WEB-INF/jsp/</value>
        </property>
        <property name="suffix">
            <value>.jsp</value>
        </property>

    </bean>

    <!-- Como cargar un fichero de properties para la i18N-->

    <bean id="messageSource"
        class="org.springframework.context.support.ResourceBundleMessageSource">

        <property name="basename">
            <value>resourceBundle</value>
        </property>

    </bean>

```



```

<!-- Como cargar un fichero de properties para la BD-->

<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
abstract="false" singleton="true" lazy-init="default" autowire="default" dependency-
check="default">
    <property name="locations">
        <list>
            <value>/WEB-
INF/classes/com/formacion/tienda/dao/dao.properties</value>
        </list>
    </property>
</bean>

<!-- ===== DataSource ===== -->

<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
        <value>jdbc:mysql://localhost:3306/libreria</value>
    </property>
    <property name="username">
        <value>root</value>
    </property>
    <property name="password">
        <value></value>
    </property>
</bean>

<bean id="librosDao" class="com.formacion.tienda.dao.GestionLibrosDAOJDBC">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>

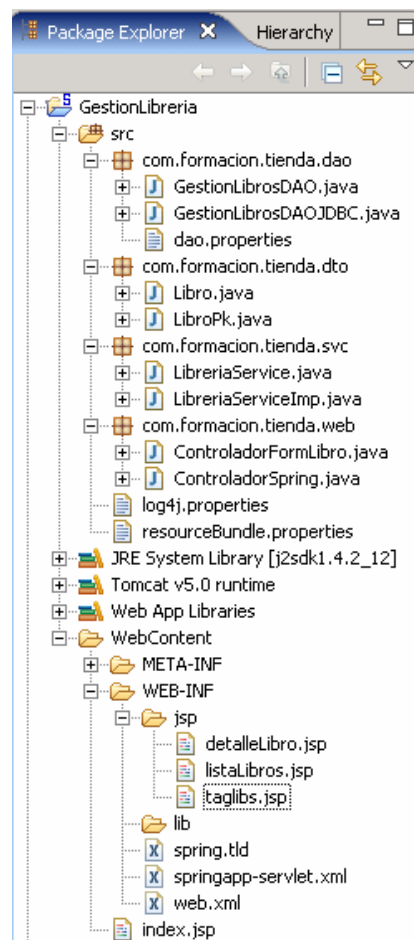
```

```
<bean id="libreriaSvc" class="com.formacion.tienda.svc.LibreriaServiceImp">
  <property name="gestionLibrosDao">
    <ref bean="librosDao"/>
  </property>
</bean>

</beans>
```

springapp-servlet.xml

Una vez creados todos los componentes de nuestra aplicación, la estructura final de ésta será la siguiente:

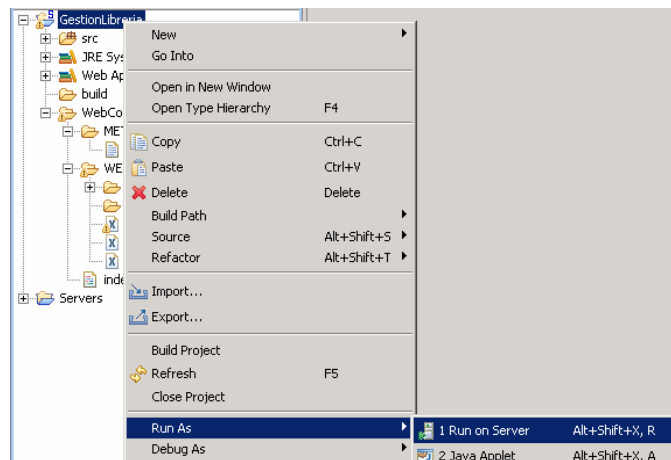


Estructura final de la aplicación **Gestión Librería**

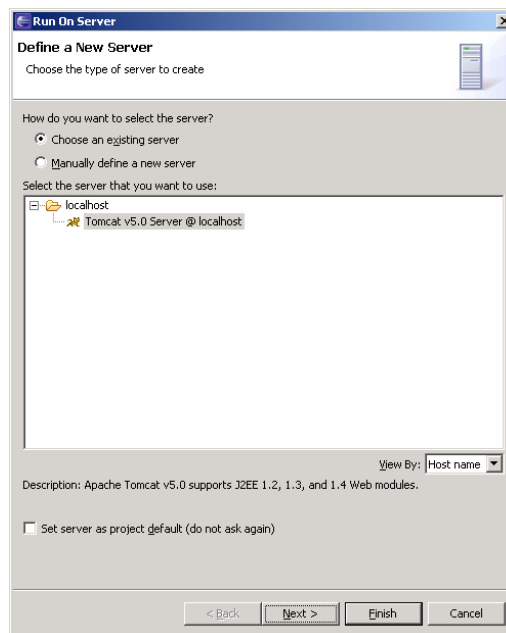
5.5. DESPLIEGUE DE LA APLICACIÓN

Finalizada la aplicación, la desplegaremos en el servidor de aplicaciones que configuramos en el primer tema. Para realizar esta operación, ejecutaremos los siguientes pasos:

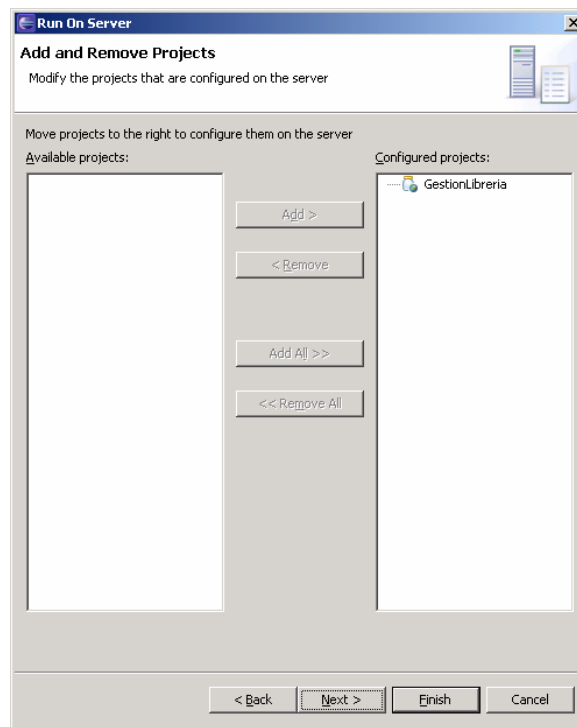
1. Seleccionamos el proyecto en el **explorador de paquetes** de Eclipse.
2. Pulsamos con el botón derecho del ratón y, en el menú que emerge, seleccionamos **Run AS→Run on Server**, como muestra la siguiente captura.




4. Seguidamente, seleccionamos el servidor **Tomcat** que configuramos en el primer tema y pulsamos **Next**.

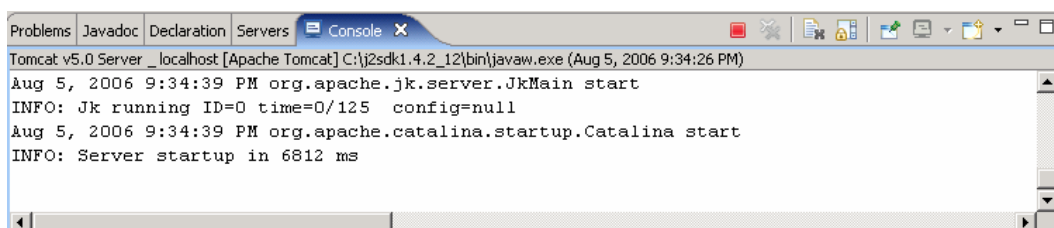


5. Una vez hecho esto, añadimos el proyecto que queremos desplegar en el servidor.
6. Por último, pulsamos **Finish** para dar por terminada la configuración del despliegue de la aplicación en el servidor de aplicaciones.

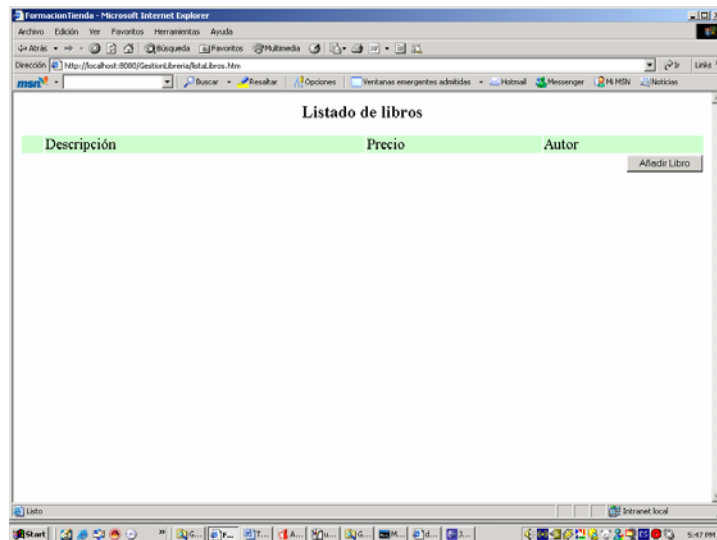


Una vez finalizada la configuración del despliegue de la aplicación en nuestro servidor **Tomcat**, sólo nos queda arrancar el servidor. Para realizar esta operación, seguiremos las siguientes indicaciones:

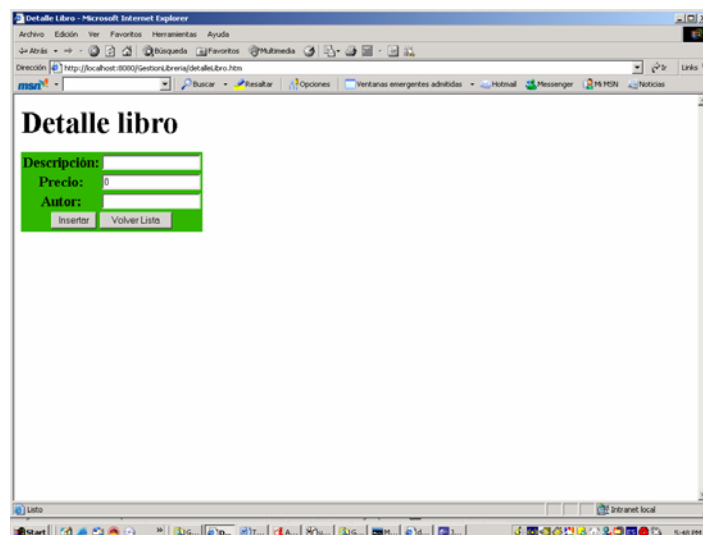
- Pulsamos sobre cualquiera de los iconos , como indicamos en el primer tema del manual. Con ello, surgirá una serie de trazas en la vista consola, como se puede observar en la siguiente captura.



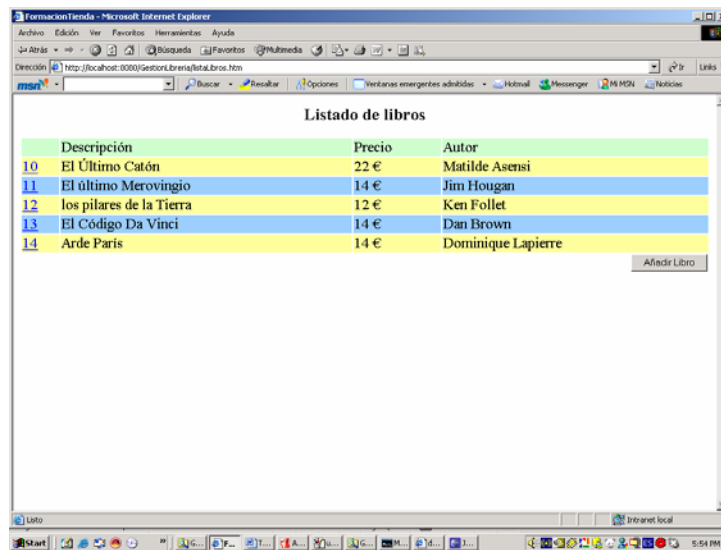
- A continuación, arrancamos un navegador, por ejemplo, Internet Explorer, y escribimos la URL de la aplicación: <http://localhost:8080/GestionLibreria> para que aparezca la pantalla de listado de libros, sin ningún libro.



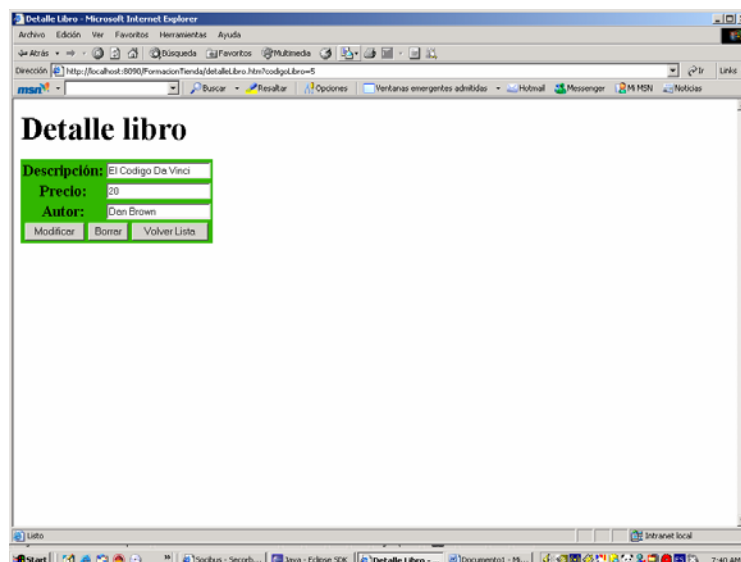
- Pulsamos el botón **Añadir Libro** para visualizar la pantalla de detalle desde donde introduciremos el libro que deseemos dar de alta y hacemos clic en **Insertar**.



- Seguidamente, repetimos el mismo proceso para dar de alta un conjunto de libros, hasta que aparezca el listado, como podemos apreciar en la siguiente pantalla.



- Asimismo, pulsando sobre cualquiera de los enlaces de la primera columna, nos remitirá al detalle del libro seleccionado, desde donde podremos modificar los valores asociados al libro que hemos seleccionado o eliminar dicho libro. En nuestro ejemplo, modificamos el precio asociado al libro y pulsamos **Modificar** para obtener el listado de libros con el valor modificado del libro.



5 Ejemplo práctico



- Es importante realizar un análisis previo de la funcionalidad que deseamos hacer, antes de empezar a desarrollar.
- Posteriormente, debemos realizar un diseño de la aplicación. Para este propósito, aconsejamos la utilización de **UML**.
- Finalizado el diseño, podemos construir nuestra aplicación, ya que en este momento tendremos una clara visión de la estructura de carpetas en el **workspace**. Esto es algo decisivo para la aplicación.
- Para aplicaciones que utilicen **Spring**, necesitamos configurar las etiquetas de **Spring** y el archivo descriptor de éstas.
- Asimismo, se han de configurar los ficheros necesarios para el desarrollo y posterior despliegue de una aplicación basada en **Spring**. Estos ficheros son **web.xml** y **application-context**, aunque debemos tener en cuenta que:
 - Una aplicación basada en **Spring** puede tener varios ficheros de configuración de **Spring**.
 - Si la aplicación utiliza **SpringMVC**, se aconseja renombrar este archivo de configuración siguiendo el patrón: **"Nombre Aplicación"-servlet.xml**.
- En el fichero **application-context.xml** se configuran todos los beans de la aplicación.
- Las **JSP**'s utilizan las etiquetas propias de **Spring**, aunque se apoyan de una manera muy significativa en las etiquetas de **jstl**.

C

Classpath. Variable de entorno que indica a Java dónde buscar las clases y librerías propias (el API de Java) y las clases de usuario.

D

DataSource. Interfaz que se encuentra en el paquete `javax.sql.DataSource` de **J2SDK**, la cual se emplea como factoría de objetos **Connection**.

Dependencia de Inyección. Modificación del patrón de **Diseño IoC**. Se basa en el **principio de Hollywood** – “**No me llames, ya te llamo yo**”, es decir, las clases de una aplicación no necesitan buscar e instanciar las clases de las que dependen. El control se invierte y es un componente contenedor el que asume la responsabilidad de la definición de dichas dependencias y la gestión de las mismas en tiempo de ejecución.

Diagrama de UML. Diagrama que define una funcionalidad.

DTD. Es una definición de elementos que pueden aparecer en un fichero con extensión **xml**.

E

Etiqueta. Clase java que se usa en las páginas `jsp`. Crea elementos de formulario y realiza otras funcionalidades en tiempo de ejecución.

F

Framework. Arquitectura que se usa como base de aplicaciones.

H

HTTP. Protocolo de Internet (Hyper Text Transfer Protocol) que permite el envío y la recepción de datos de un servidor.

HTTPS. Protocolo de Internet (Hyper Text Transfer Protocol Secure) que permite el envío y la recepción de datos de un servidor de una forma segura.

I

Instancia. Es un objeto de una clase, es decir, el espacio de memoria que se va a usar para una implementación.

IoC (Inversión de Control). Es un patrón de diseño que permite la escritura de un código mucho más limpio y sencillo, facilitando el desacoplamiento entre los componentes de la arquitectura y la definición declarativa.

J

JNDI (Java Naming and Directory Interface). **API** que proporciona **sun**, para búsqueda de servicios por su nombre.

L

Librería. En java llamamos librerías a los archivos **.jar (punto jar)**. Es un conjunto de clases java compiladas (**.class**) y comprimidas de forma que son inteligibles por el compilador.

Lógica de negocio. Parte de la aplicación que maneja la lógica de negocio.

O

ORM. Mapeo objeto-relacional (object-relational mapping). Una ORM se utiliza para mapear las tablas de la base de datos de nuestra aplicación.

P

Patrón de diseño. Son soluciones estándar a un problema software que siguen un mismo patrón. Se puede reutilizar tantas veces como se desee.

Plug-in. Pequeño programa que extiende las capacidades de otros.

POJO. (Plain Old Java Object). Es una clase normal de java, compuesta por atributos y los métodos **set** y **get** para manejar los valores de los atributos.

S

Scriptlet. Evaluaciones de expresiones en un página **JSP's**. Anteriormente, se indicaban entre las etiquetas **<% expresion%>** y con Struts se presentan así **#{expresion}**.

Software libre. Son programas que pueden ser usados, distribuidos y mejorados como el usuario desee.

Spring IDE. Es un plug-ing de eclipse que proporciona soporte Spring para las aplicaciones.

Subclase. Clase hija, aquella que extiende de otra clase.

U

UML (Unify Modeling Language). Es un lenguaje que se usa para analizar una aplicación antes de desarrollarla.

V

Validación. Comprobación que se realiza de que los datos introducidos por el usuario son del tipo que se desea o cumplen las reglas necesarias en la aplicación.

W

Workspace. Área base de trabajo de un desarrollador, donde se tiene tanto las clases java, como las compiladas y las configuraciones precisas para que toda la aplicación pueda funcionar correctamente.

referencias web_

Wikipedia, la enciclopedia libre en Internet	http://es.wikipedia.org
AppFuse	www.appfuse.org

bibliografía_

- Spring java/j2ee Application Framework Reference Documentation Version 1.2.8.*
Craig Walls, Ryan Breidenbach. *Spring in Action*. Manning Publications Co. Greenwich. 2005.
- Matt Raible. *Spring live*. SourceBeat, LLC, Highlands Ranch, Colorado. 2005.
- Rob Harrop y Jan Machacek. *Pro Spring*. Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany. 2005.
- Justin Gehtland, Bruce A. Tate. *Spring: A Developer's Notebook*. Ed O'Reilly. 1005 Gravenstein Highway North, Sebastopol, CA 95472. 2005.
- Rob Johnson et al. *Professional Java Development with Spring Framework*. Ed John Wiley & Sons. 2005.