UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA INFORMÀTICA

"Material didáctico para la enseñanza de SPRING"

PROYECTO FINAL DE CARRERA

CÓDIGO: DISCA-121

Autor: Juan Carlos Ceresola Millet

Director: Dr. Lenin Guillermo Lemus Zúñiga
Profesor Titular de Universidad
Universidad Politécnica de Valencia

12 de Julio de 2012

Índice

Índice de figuras

Índice de Listados

1 Introducción

En la actualidad el lenguaje Java está de moda, cualquier empresa de software requiere un programador conocedor de este lenguaje. Además el desarrollo de Java se realiza mediante entornos integrados de desarrollo como Eclipse y con la utilización de un framework especializado y no intrusivo para un desarrollo más claro y rápido del codigo.

Este proyecto proporciona los conocimientos y ejemplos necesarios para empezar a manejar el framework Spring con el lenguaje Java.

1.1 Contexto

Actualmente las aplicaciones web, y más concretamente las aplicaciones web con el lenguaje Java están experimentando un gran auge. Para desarrollar este tipo de aplicaciones existen numerosos framework y aplicaciones para facilitar la creación de estas, ahorrando a los programadores muchas líneas de código y ayudando en la configuración de la aplicación en tareas tales como: dependencias, seguridad, tratamiento de errores, etc.…

Dentro de todos los framework disponibles hemos elegido Spring por ser un framework liviano y no intrusivo; ya que los objetos que programamos no tienen dependencias en clases específicas de Spring. Sus características principales son la inyección de dependencias y la programación orientada a aspectos. Spring está estructurado en módulos lo que hace mucho más flexible su configuración.

1.2 Motivación del proyecto

En estos momentos el framework Spring es uno de los más demandados por las empresas del sector para el desarrollo de aplicaciones tanto web como locales, dado que en la carrera hemos estudiado el lenguaje Java, es una opción interesante.

1.3 Metodología de trabajo

1. Búsqueda bibliográfica de documentación y características del framework Spring, del lenguaje de programación Java.
2. Búsqueda de la plataforma de desarrollo y del servidor de aplicaciones.
3. Análisis de las diferentes plataformas existentes y servidores.
4. Configuración del entorno de desarrollo y del servidor para su funcionamiento con el framework Spring.
5. Creación de una estructura para el portal web, roles, aspecto, menús de los distintos usuarios, seguridad, etc.…
6. Implementación del portal docente.
7. Pruebas de funcionamiento del portal docente.

2 Objetivo

El objetivo de este proyecto es proporcionar los conocimientos y ejemplos necesarios para que cualquier persona con conocimientos de programación y del lenguaje Java pueda iniciarse y entender cómo utilizar el framework Spring y acceder a las bases de datos con este. Además entenderá la estructura del framework y los diferentes módulos que lo integran.

2.1 Definiciones básicas.

Framework: es un esquema o patrón para el desarrollo e implementación de una aplicación.

Api: es el conjunto de funciones y procedimientos o métodos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

POJOS: son clases simples en java que no dependen de ningún framework en especial.

Bytecode: codigo intermedio más abstracto que el codigo maquina.

Dispatcher: es el encargado de enviar (despachar) las peticiones.

Factory: son las clases encargadas de crear los elementos más apropiados para el programa en ese momento.

Interceptors: clase que cumple con cierta interfaz, y su objetivo es "interponerse" en la ejecución de un método. Un interceptor es capaz de ejecutarse antes de un método dado, realizar acciones, y luego continuar con la ejecución normal (o abortarla).

Inversión de control: técnica de programación mediante la cual se deja a un programa o arquitectura externa lleva a cabo las acciones de control, especificando nosotros solamente el resultado que deseamos obtener.

2.2 Selección del entorno de desarrollo

Vamos a presentar varias de las distintas plataformas para el desarrollo con el lenguaje Java que existen, así como los distintos servidores para desplegar la aplicación java:

2.3 Entornos de desarrollo:

- Eclipse: es un entorno de desarrollo integrado de código abierto multiplataforma para el desarrollo de aplicaciones con varios lenguajes.
- Netbeans: es un entorno de desarrollo integrado, es libre y está hecho principalmente para el lenguaje Java.
- BlueJ: BlueJ es un entorno integrado de desarrollo para el lenguaje de programación Java, desarrollado principalmente con propósitos educacionales, pero también es adecuado para el desarrollo de software a pequeña escala.
- DrJava: es un entorno integrado de desarrollo para Java que está pensado para poder aprender este lenguaje, es ligero, de interfaz intuitiva, gratuito y tiene la capacidad de evaluar código de manera interactiva. Está programado en java, de esa manera también logra ser multiplataforma.
- JBuilder: es el entorno de desarrollo integrado para Java de Borland, existen 3.
- JCreator: entorno de desarrollo integrado de java para la plataforma Windows. Existen 2 versiones una gratuita y otra de pago.
- JEdit: editor de texto libre escrito en lenguaje java, dispone de múltiples plugins para diferentes aplicaciones.

- JDeveloper: es un software de desarrollo integrado de Oracle para lenguajes de programación entre ellos Java, es libre y desde el 2001 está basado en java por lo que funciona en todas las plataformas, siempre que tengan instalada la maquina virtual de Java.

2.4 Servidores

- Tomcat: funciona como un contenedor de servlets e implementa las especificaciones de servlets y JavaServerPages (JSP). Los usuarios de Tomcat tienen acceso a su código fuente, ante todo Tomcat no es un servidor de aplicaciones.

- "GlassFish": servidor de aplicaciones de software libre desarrollado por Sun que implementa las tecnologías definidas en la plataforma J2EE y permite ejecutar aplicaciones que siguen dicha especificación.

- Jonas: es un servidor de aplicaciones J2EE de código abierto implementado en Java, por lo que puede funcionar en cualquier plataforma que tenga instalada la maquina virtual de Java.

- JBoss: servidor de aplicaciones J2EE de código abierto implementado en Java puro, sus características más destacadas en que cumple los estándares y es confiable a nivel de empresa.

- Jetty: es un servidor HTTP basado en Java también es un contenedor de servlets escritos en Java, es software libre y es utilizado por otros proyectos como JBoss y Geronimo.

- WebSphere Aplication Server (WAS): es un servidor de aplicaciones de IBM basado en estándares abiertos como J2EE, XML y servicios web.

- BEA WebLogic: es un servidor de aplicaciones y un servidor HTTP de Oracle.

Al final nos hemos decantado por el entorno de desarrollo Eclipse por que existe una versión de Spring adaptada especialmente para el desarrollo de aplicaciones con el framework Spring, además de tener gran material de soporte y gestor de plugins.

En cuanto al servidor hemos elegido ""GlassFish"" por su sencillez de instalación en Eclipse y por su soporte para JPA.

Capitulo 1

1 Introducción.

El objetivo de este capítulo es conocer que es lenguaje Java y porque esta tan de moda y es el preferido para la mayoría de los programadores dada su facilidad para pasar de una plataforma a otra, como funciona el código Java y como se puede hacer funcionar mediante contenedores web y EJB.

2 ¿Qué es J2EE?

Para empezar a comprender que es Java2EE primero tenemos que comprender que es Java. Java es lenguaje de programación de alto nivel y orientado a objetos e independiente del hardware en la que se ejecuta. Esto último se consigue gracias a la maquina virtual de Java, para que una aplicación Java funcione tenemos que tener instalado la JMV (Java virtual machine) y el JRE (Java Runtime Enviorment) necesario para crear aplicaciones Java. Al compilar el código Java no se genera código maquina como se hace normalmente sino un código intermedio, esto es así ya que el código maquina depende del ordenador en que se ejecute, por eso se genera el código intermedio y después la maquina virtual de Java es la encargada de transformar este código intermedio en el código maquina correspondiente para cada ordenador en el que se ejecute para lograr la total independencia del hardware.

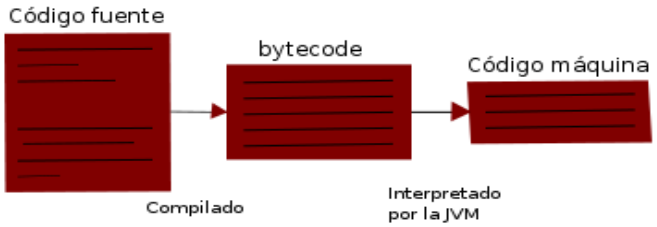

Figura 1. Transformación código fuente

Las diferentes versiones de Java existentes son:

1. Java EE: aplicaciones distribuidas multicapa sobre web.
2. Java SE: aplicaciones de escritorio y applets.
3. Java ME: aplicaciones para dispositivos móviles (versión simplificada de SE y Apis especificas).
4. Java FX: aplicaciones web que tienen las características y capacidades de aplicaciones de escritorio.

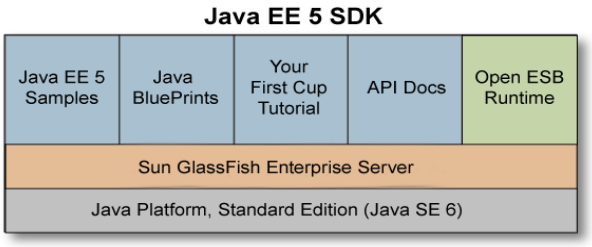

Figura 2. Estructura Java2EE

Java EE (*Enterprise edition*) es una especificación Java para el desarrollo de aplicaciones empresariales, Java EE especifica una serie Api´s tales como JDBC, RMI, e-mail, JMS, servicios web, XML, … que todos los programas deben cumplir, es decir un estándar que todos los fabricantes deben cumplir, esto hace que las aplicaciones sean portables e independientes de la plataforma. Java EE se apoya sobre la versión Java SE (Standard edition) que provee la plataforma de ejecución (Java virtual machine y Apis básicas) y de compilación. Java EE además ofrece una serie de especificaciones únicas para Java EE y sus componentes como Enterprise JavaBeans, servlets, portlets, JavaServer Pages, .. . Esto permite a los desarrolladores crear aplicaciones portables y escalables a la vez que integrables con tecnologías anteriores además esto permite que el servidor de aplicaciones pueda manejar transacciones, seguridad, concurrencia y la gestión de los componentes desplegados.

La arquitectura JEE implica un modelo de aplicaciones distribuidas en diversas capas o niveles (tier). La capa cliente admite diversas tipos de clientes (HTML, Applet, aplicaciones Java, etc.). La capa intermedia (middle tier) contiene subcapas (el contenedor web y el contenedor EJB). La tercera capa dentro de esta visión sintética es la de de aplicaciones backend como ERP, EIS, bases de datos, etc. El contenedor, no es más que un entorno de ejecución estandarizado que ofrece unos servicios por medio de componentes. Los componentes externos al contenedor tienen una forma estándar de acceder a los servicios de dicho contenedor, con independencia del fabricante

Por lo general las aplicaciones formadas por capas: presentación, negocio o lógica y persistencia, hacen que a esta división se le pueda asignar un experto en cada una de las capas y que se pueden mejorar la utilización de recursos. Pero esta división también tiene sus inconvenientes y dificultades tales como que la integración de niveles de forma eficiente es bastante costosa y complicada, requiere de un gran número de transacciones que se han de supervisar y de métodos de seguridad para su correcto funcionamiento sin poner en peligro los datos.

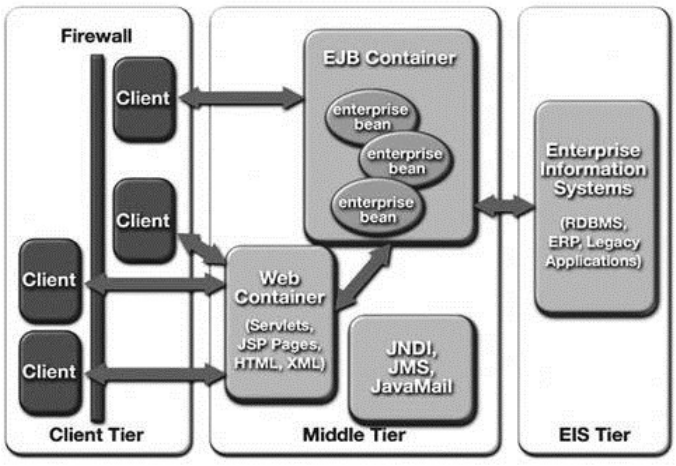

Figura 3. Contenedor Web.

Los contenedores de Java EE brindan dos tipos de servicios EJB y web.

3 Contenedores Web.

Los contenedores web son unos entornos que encapsulan los protocolos HTTP y TCP/IP es decir nos liberan de la tarea de creación de sockets y serverSockets así como la de realizar la conexión con el navegador, pueden contener tanto Servlets como páginas web dinámicas.

Los Servlets son básicamente código Java que puede tener código HTML incrustado aunque es desaconsejable esta práctica por diferentes razones:

- El HTML es más fácil de manejar en un entorno Web y no en un entorno Java.
- Programar web no es lo mismo que programar Java, por lo que se aconseja dejar cada tarea a desarrolladores expertos en cada campo.

Los Servlets pueden acceder a una gran variedad de Api's de Java y tiene la ventaja de que una vez invocados se quedan en memoria por lo que las siguientes invocaciones solo es necesario instanciarlos mejorando significativamente el tiempo de respuesta.

Los JSP (Java Server Pages) son equivalentes a las ASP de Microsoft, son unas páginas web con un grupo de etiquetas ampliado y pueden contener código Java incrustado, aunque esto debemos evitarlo.

Los JavaBeans siguen el concepto de encapsulación por lo que se accede a sus propiedades privadas a través de sus métodos públicos. Son llamados por los Servlets y los JSP para:

- Contener información en memoria
- Ejecutar lógica de negocio
- Conexión a bases de datos.

4 Contenedores EJB.

Los contenedores EJB son un entorno en el que se ejecutan los EJB (Enterprisse JavaBeans) no confundir con los JavaBeans pues los EJB pueden ser activados por si solos y los JavaBeans deben estar integrados en otros componentes. Son un programa que se ejecuta en el servidor y contiene todas las clases y objetos necesarios para el funcionamiento de los enterprisse beans.

Las ventajas que ofrece el contenedor EJB son:

- Manejo de transacciones.
- Seguridad y comprobación de los métodos de acceso.
- Concurrencia, llamada a un mismo recurso desde múltiples clientes.
- Servicios de red.
- Gestión de recursos: gestión de múltiples recursos.
- Persistencia: sincronización entre los datos de la aplicación y los de las tablas de los base de datos.
- Gestión de mensajes: Escalabilidad, posibilidad de constituir clúster de servidores de aplicaciones con múltiples equipos para poder dar respuesta a aumentos repentinos de carga de la aplicación con sólo añadir hosts adicionales.
- Adaptación en tiempo de despliegue: posibilidad de modificación de las características en el momento de despliegue del bean.

Capitulo 2

1 Introducción

En este capítulo vamos a ver uno de los servidores más populares para ejecutar java en entornos web, así como descargarlo, instalarlo, configurarlo y arrancarlo.

2 “GlassFish”

“GlassFish” es el servidor de aplicaciones desarrollado por Sun para implementar las tecnologías definidas por las Apis de la versión Java EE tanto de servicios web como de Enterprise beans. Existen dos versiones una gratuita de código libre llamada “GlassFish” solamente y otra comercial llamada Sun “GlassFish” Enterprise Server. “GlassFish” al contrario que su competidor Tomcat no es solamente un contenedor web sino que es un conjunto de contenedores Java EE, uno de los cuales es un contenedor web.

Vamos a ver una serie de ventajas que tiene “GlassFish” sobre Tomcat:

- **Ruta de migración más sencilla**. Con “GlassFish” v2 hay una forma clara y sencilla de aprovechar tecnologías tales como Enterprise Java Beans (EJBs), Java Persistence API (JPA) y Java Message Service (JMS), entre otras. Con Tomcat, estas tecnologías deben agregarse poco a poco, una a una.

- **Superioridad en la administración y la supervisión**. “GlassFish” permite la administración centralizada a través de una consola de administración y de una interfaz de línea de comandos (CLI). “GlassFish” ofrece supervisión del flujo de llamada, que permite a un desarrollador de aplicaciones o un administrador de servidores determinar a qué dedica la aplicación la mayor parte de su tiempo. Con Tomcat, el software nuevo se configura de forma poco sistemática.

- **Compatibilidad con lenguajes de script**. “GlassFish” admite, Ruby/JRuby, Python/Jython, Groovy, PHP, JavaScript/Phobos y Scala, entre otros lenguajes.

En cuanto a la diferencias entre los contenedores web:

- La capacidad de retener sesiones entre distintos despliegues de aplicaciones supone un importante ahorro de tiempo para los desarrolladores que crean aplicaciones Web Java.

- “GlassFish” incluye varias optimizaciones de rendimiento, como “flattened valve invocation”, una modificación de la arquitectura de válvula que racionaliza la forma de llamar a cada válvula, reduciendo

así la profundidad de la pila y mejorando el rendimiento. “GlassFish” v3 Prelude también admite válvulas de estilo Tomcat.

- “GlassFish” facilita la reconfiguración dinámica de servidores virtuales y Listeners HTTP sin necesidad de reiniciar el servidor. Con Tomcat, cuando se modifica una fuente de recursos, suele ser necesario reiniciar el servidor de aplicaciones.

3 Instalación de J2EE y “GlassFish”.

Lo primero que haremos será bajarnos la última versión de Java EE que se encuentra en la página oficial de Oracle existen varios paquetes para descargar pero nosotros vamos a elegir el paquete de Java EE que aparte de incluir el SDK de Java EE incluye el servidor “GlassFish”.

http://www.oracle.com/technetwork/java/javaee/downloads/index.html

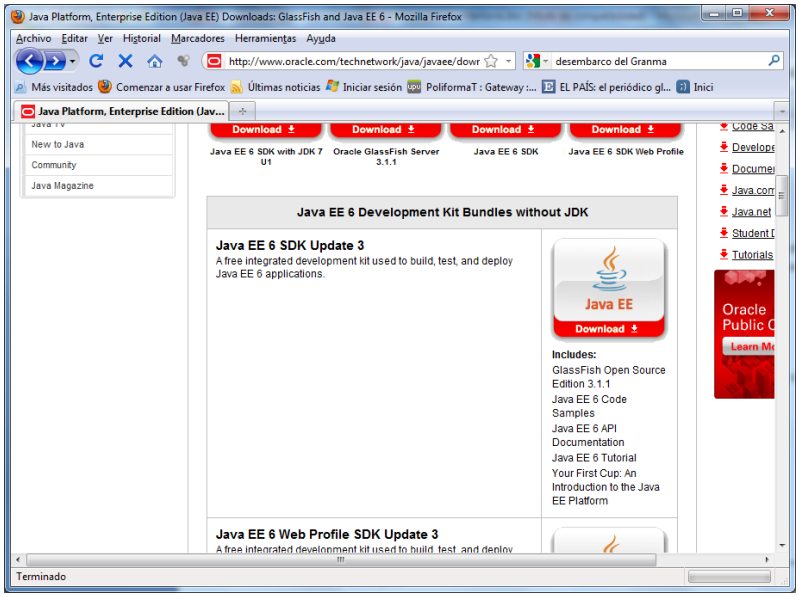

Figura 4. Página de descarga de Java EE.

Aceptamos el acuerdo de licencia y escogemos la versión que se corresponda con nuestro sistema operativo, en este caso Windows.

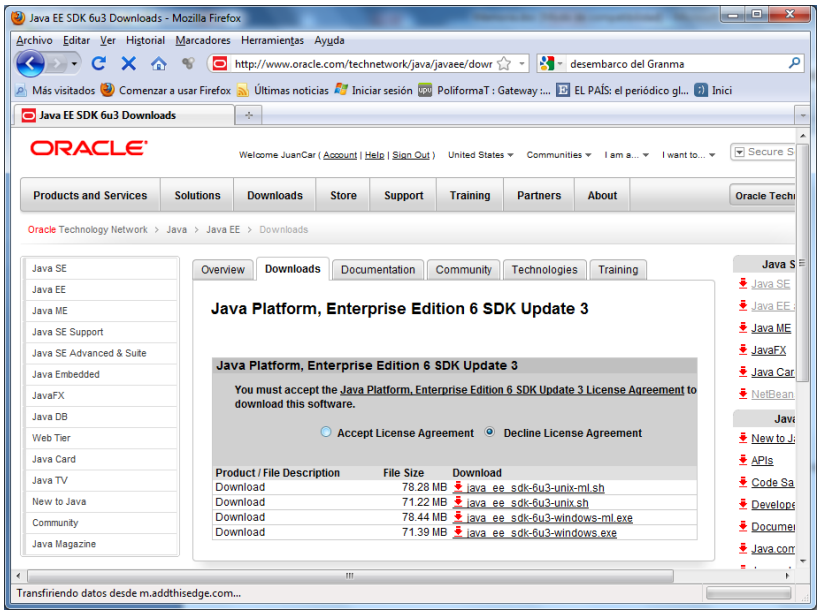

Figura 5. Selección del paquete a descargar.

Una vez descargado nos aparecerá la siguiente pantalla dándonos una explicación de lo que se va a instalar.

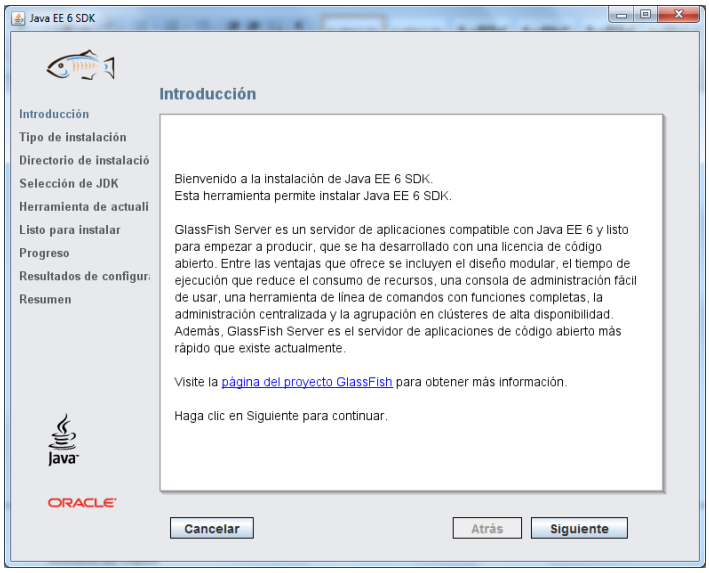

Figura 6. Página de bienvenida.

Después de presionar siguiente nos aparecerá una pantalla con la opción de escoger instalación. Típico o personalizada. Puedes escoger la típica y olvidarte de configurar directorios y puertos para la aplicación.

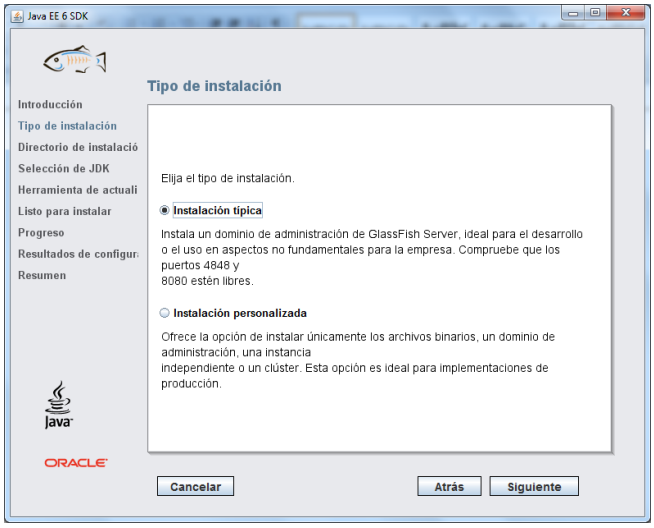

Figura 7. Selección del tipo de instalación

Seguidamente escogemos el directorio donde se va instalar el "GlassFish".

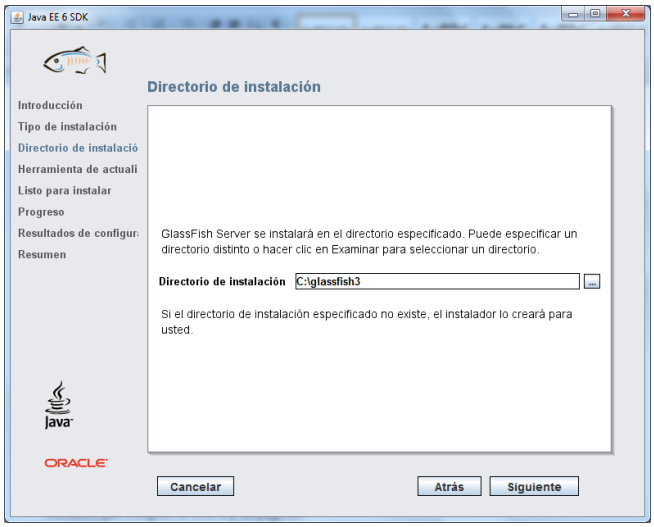

Figura 8. Selección del directorio para la instalación.

Nos aparecerá una pantalla para configurar la actualización automática del servidor de aplicaciones "GlassFish", dejaremos las opciones en blanco, hay que decir que existen varias formas de actualizar el "GlassFish" siendo esta la más sencilla.

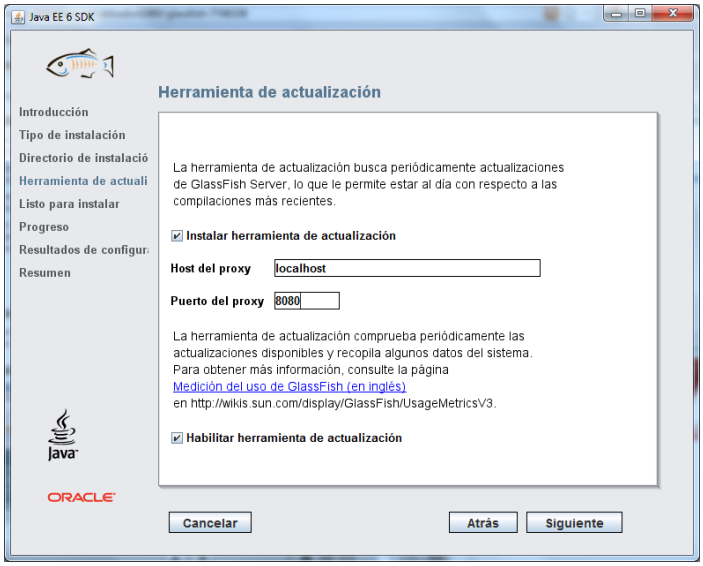

Figura 9. Parámetros de la instalación.

Seguidamente nos mostrara los elementos que se van a instalar.

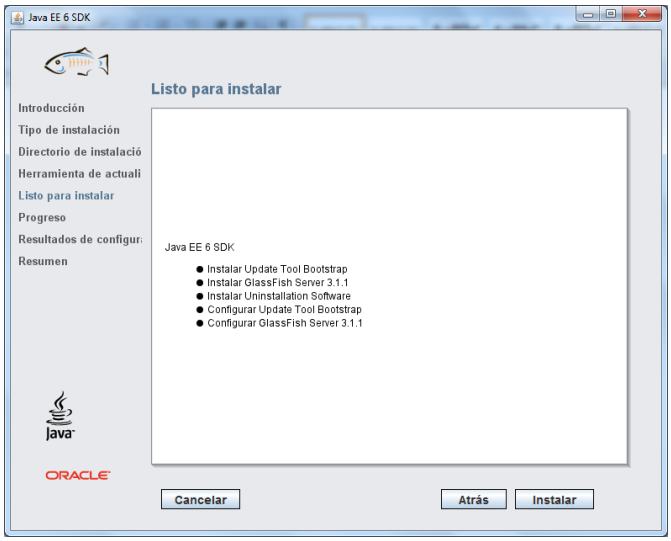

Figura 10. Lista de productos a instalar.

Después de presionar instalar la aplicación empezara a instalarse.

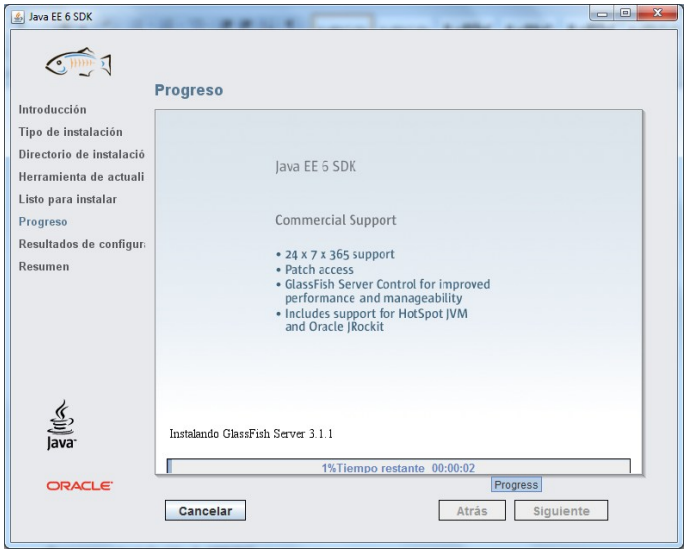

Figura 11. Progreso de la instalación.

Una vez instalada nos mostrara un resumen con todo lo que se ha instalado y si ha habido algún error durante la instalación.

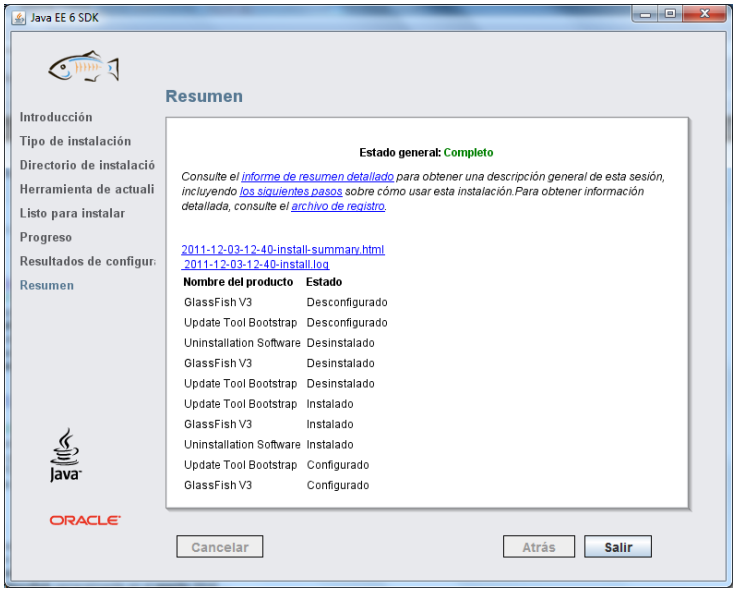

Figura 12. Resumen de la instalación.

Capitulo 3

1 Introducción

En este capítulo vamos a explicar que es el framework Spring, los módulos que lo forman así como algunas de las funciones de estos módulos, también veremos cómo descargar e instalar SpringToolSuite que es una versión de Eclipse especialmente creada para el desarrollo de aplicaciones con el framework Spring.

2 ¿Qué es SPRING?

Spring es un framework de código abierto de desarrollo de aplicaciones para la plataforma Java. La primera versión fue escrita por Rod Johnson. También hay una versión para la plataforma .NET, Spring.net.Inicialmente el framework se lanzó bajo la licencia Apache 2.0 en junio de 2003.

Spring Framework no obliga a usar un modelo de programación en particular, no obstante se ha popularizado en la comunidad de programadores en Java, siendo alternativa del modelo de Enterprise JavaBean. El diseño del framework ofrece mucha libertad a los desarrolladores en Java y soluciones muy bien documentadas y fáciles de usar.

Mientras que las características fundamentales de este framework pueden emplearse en cualquier aplicación hecha en Java, existen muchas extensiones y mejoras para construir aplicaciones basadas en web por encima de la plataforma empresarial de Java EE.

Spring está basado en los siguientes principios:

- El buen diseño es más importante que la tecnología subyacente.
- Los JavaBeans ligados de una manera más libre entre interfaces es un buen modelo.
- El código debe ser fácil de probar.

3 Módulos que integran SPRING.

Los módulos que integran Spring pueden trabajar independientemente unos de otros. Spring está formado más o menos por unos 20 módulos aquí vamos a explicar los más importantes.

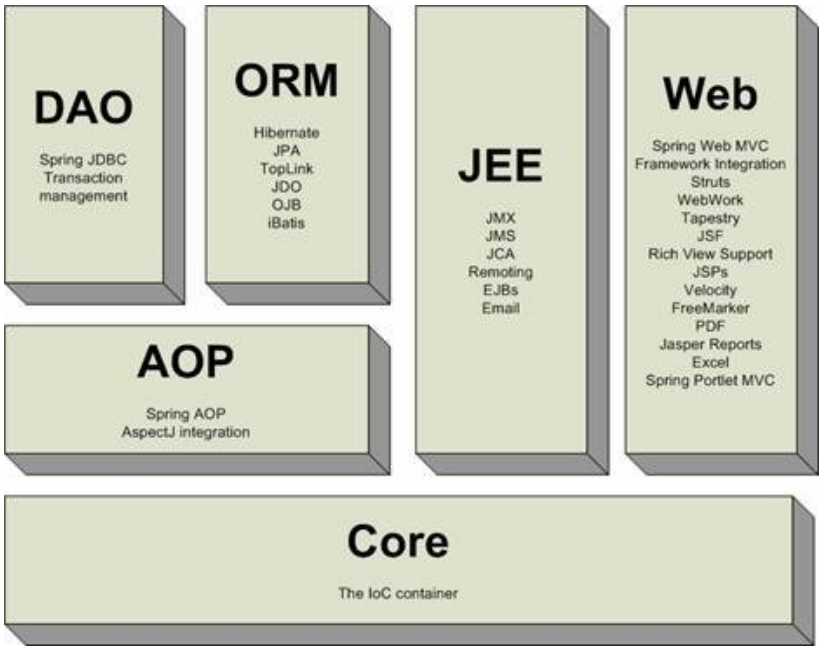

Figura 13. Módulos de Spring

3.1 Núcleo: la parte fundamental del framework es el módulo Core o "Núcleo", este provee toda la funcionalidad de Inyección de Dependencias permitiéndole administrar la funcionalidad del contenedor de beans. El concepto básico de este módulo es la fábrica de beans, que implementa el patrón de diseño Factory (fábrica) eliminando la necesidad de crear singletons programáticamente permitiéndole desligar la configuración y especificación de las dependencias de la lógica de programación.

La fábrica de beans (BeanFactory) utiliza el patrón de inversión de control (Inversion of Control) y configura los objetos a través de la inyección de dependencia (Dependency Inyection). El núcleo de Spring es el paquete "org.springframework.beans" que está diseñado para trabajar con JavaBeans.

La fábrica de beans es uno de los principales componentes del núcleo de Spring puede crear muchos tipos de beans que pueden ser llamados por su nombre y se encarga de manejar las relaciones entre ellos. Las fábricas de beans implementan la interface "org.springframework.beans.factory.BeanFactory" con instancias a las que se accede a través de esta interfaz. Además también soporta objetos de dos modos diferentes.

- Singleton: existe solamente una instancia compartida de un objeto con un nombre único, puede ser devuelto o llamado tantas veces como se requiera, es el método más común.
- Prototype o non-sigleton: cada vez que se realiza una devolución o llamada se crea un objeto independiente.

La implementación de la fábrica de beans más usada es "org.springframework.beans.factory.xml.XmlBeanFactory" que carga la definición de cada bean que se encuentra guardada en un archivo XML que consta de un id, una clase, si es singletos o prototype, propiedades con sus atributos name, value y ref, además de argumentos del constructor, método de inicialización y método de destrucción.

```
<beans>
<bean id="exampleBean" class="eg.ExampleBean" singleton="true" />
<property name="driveClassName" value="com.mysql.jdbc.Driver" />
</beans>
```

Listado 1. Declaración del vean que gestiona el manejador de BBDD de MySQL

La base del documento XML es el elemento <beans> y dentro puede contener más elementos bean para cada una de las funciones que se requieran.

Para cargar el XML se le manda un Inputstream al constructor pero esto no significa que se instancie directamente, una vez que la definición está cargada, únicamente se creara la instancia cuando se necesite el Bean.

Inversion de control: la fábrica de beans utiliza la inversión de control o IoC que es una de las funcionalidades más importantes de Spring, se encarga de separar el código de la aplicación que se está desarrollando, los aspectos de configuración y las relaciones de dependencia del framework. Configura los objetos a través de la inyección de dependencia. En lugar de que el código de la aplicación llame a una clase de la librería, un framework que utiliza la inyección de control llama al código.

Inyección de dependencia: Es una forma de IoC, que está basada en constructores de Java, en vez de usar interfaces específicas del framework. Con este método en lugar de que el código utilice la API del framework para resolver las dependencias las clases de la aplicación exponen sus dependencias para que el framework pueda llamarlas en tiempo de ejecución.

Spring soporta varios tipos de inyección pero estos dos son los más utilizados:

- Setter Inyection: este tipo de inyección es aplicada por métodos JavaBeans setters.
- Constructor Inyection: esta inyección es a través de los argumentos del constructor.

Spring Context: es un archivo de configuración que provee de información contextual al framework general. También provee servicios Enterprise como JDNI, EJB, e-mail, validación y funcionalidad de agenda.

Aplication Context: es una subinterfaz de la fábrica de beans, todo lo que puede realizar la fábrica de beans también lo puede realizar Aplication Context, agrega información de la aplicación que puede ser utilizada por todos los componentes.

Además brinda las siguientes funcionalidades extra:

- Localización y reconocimiento automático de las definiciones de los beans.
- Cargar múltiples contextos.
- Contextos de herencia.
- Búsqueda de mensajes para encontrar su origen.
- Acceso a recursos.
- Propagación de eventos, para permitir que los objetos de la aplicación puedan publicar y opcionalmente registrarse para ser notificados de los eventos.
- Agrega soporte para internacionalización.

3.2 DAO: El patrón DAO (Data Acces Object) es uno de los módulos más importantes y utilizados en J2EE.

DAO y JDBC: hay dos maneras de llevar a cabo la conexión, manejo y acceso a las bases de datos: utilizar alguna herramienta ORM o utilizar JDBC (Java Database Connectivity) que ofrece Spring. Si se trata de una aplicación

simple donde la conexión la realizara una sola clase la mejor opción sería JDBC, si por el contrario se desea más complejidad, dar mayor soporte y mejor robustez es recomendable utilizar una herramienta ORM.

El uso de JDBC puede llevar en muchos casos a repetir el mismo código en distintos lugares al crear la conexión, buscar y manejar los datos y cerrar la conexión. El uso de las tecnologías antes mencionadas mantiene el código simple y evita que sea tan repetitivo además de evitar los errores comunes al cerrar una conexión contra una base de datos. Las clases base que Spring propone para la utilización de DAO son abstractos y brindan un fácil acceso a los recursos comunes de las bases de datos. Existen diferentes implementaciones para cada una de las tecnologías de acceso a datos que soporta Spring.

Para JDBC esta la clase JdbcDaoSupport que provee los métodos para acceder al DataSource y a JdbcTemplate, este se encarga de atrapar todas las excepciones de SQL y las convierte en una subclase de DataAccesException. También los JdbcTemplate se encargan de controlar la conexión a la base de datos. Y se puede evitar configurar el DataSource en DAO, únicamente se tiene que establecer a través de la inyección de dependencia.

3.3 ORM: Spring no propone un ORM (Object-Relational Mapping) propio para los usuarios que no quieran simplemente usar JDBC, en su lugar ofrece un modulo que soporta los framework más populares, entre ellos:

- Hibernate. Es una herramienta de mapeo O/R muy popular. Utiliza su propio lenguaje de consultas llamado HQL.
- iBATIS SQL Maps: una solución sencilla y fuerte para hacer externas las declaraciones de SQL en archivos XML.
- Apache OJB: plataforma de mapeo O/R con múltiples Api's para clientes.
- Otros como JDO y Oracle TopLink.

Algunas de las ventajas con las que cuenta Spring al combinarse con alguna herramienta ORM son:

- Manejo de sesión: Spring hace de manera más sencilla, eficiente y segura que cualquier herramienta ORM, la forma en que se manejan las sesiones.
- Manejo de recursos: se puede manejar la localización y configuración de las fábricas de sesiones de Hibernate o las fuentes de datos de JDBC, por ejemplo haciendo que estos valores sean más fáciles de modificar.
- Manejo de transacciones integrado: se puede utilizar una plantilla de Spring para las diferentes transacciones ORM.
- Envolver excepciones: se pueden envolver todas las excepciones para evitar las declaraciones y los catch en cada segmento de código necesario.
- Evita limitarse a un solo producto: si se quiera migrar o actualizar a una versión de un ORM distinto o del mismo, Spring trata de no crear una dependencia entre la herramienta ORM, Spring y el código de la aplicación, para que cuando se quiera cambiar a otro ORM resulte muy sencillo.
- Facilidad de prueba: Spring trata de crear pequeños pedazos que se puedan aislar y probar por separado, ya sean sesiones o una fuente de datos.

3.4 AOP: Aspect-oriented programming (Programación orientada a aspectos), esta técnica permite a los programadores modularizar la programación. El núcleo de construcción es el aspecto que encapsula el comportamiento que afectan a diferentes clases, en módulos que pueden ser utilizados.

Alguna de sus utilidades son:

- Persistencia.
- Manejo de transacciones.
- Seguridad.
- Autentificación.
- Depuración.

Spring AOP es portable entre servidores de aplicación funciona tanto en servidores web como en contenedores EJB y soporta las siguientes funcionalidades:

- Intercepción: se puede insertar comportamiento personalizado antes o después de llamar a un método en cualquier clase o interfaz.
- Introducción: especificando que es un advice (acción tomada en un punto del programa durante su ejecución) debe hacer que un objeto implemente interfaces adicionales.
- Pointcuts estáticos y dinámicos: para especificar los puntos del programa donde debe hacer una intercepción.

Spring implementa AOP utilizando proxies dinámicos, además se integran transparentemente con las fábricas de beans que existen.

3.5 WEB: El modulo de Spring Web se encuentra en la parte superior del modulo Context. En este modulo se encuentra el MVC (Modelo Vista Controlador) por lo que este modulo se explicara en el apartado de MVC.

4 Descarga e Instalación de SPRING

Para instalar SpringToolSuite primero debemos descargárnoslo de la página oficial de Spring http://www.springsource.org/downloads/sts y escogemos la versión que nos interese para nuestro sistema operativo, en este caso Windows 64.

Una vez descargado el archivo lo ejecutamos.

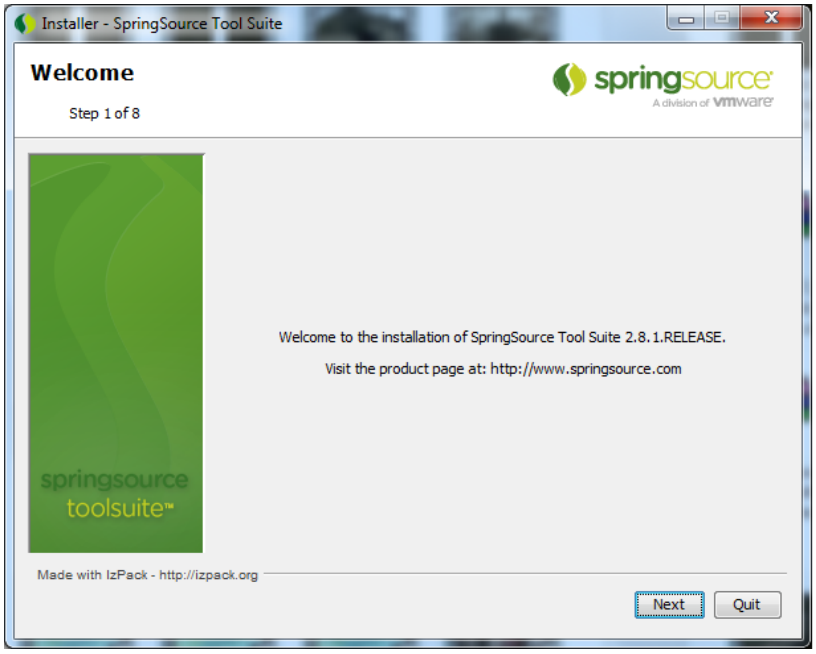

Figura 14. Página de inicio de la instalación.

Presionamos el botón Next.

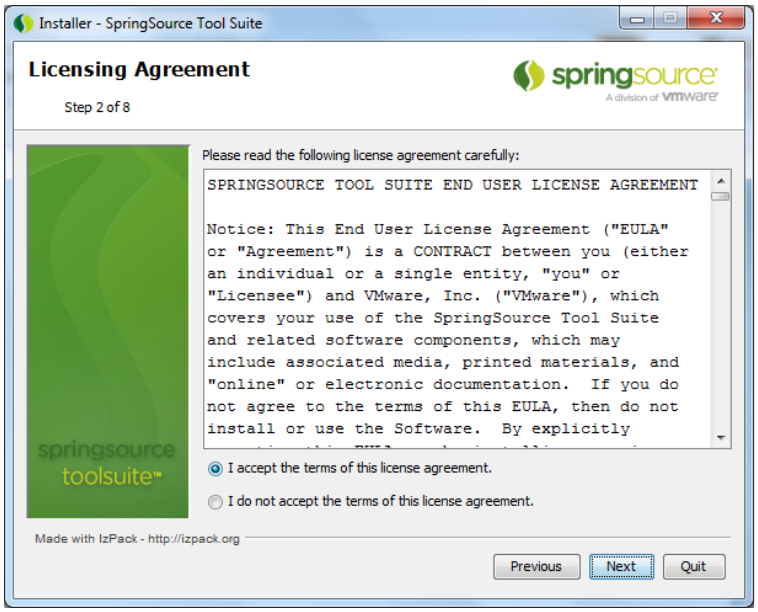

Figura 15. Aceptación de la licencia.

Aceptamos la licencia y pulsamos el botón Next.

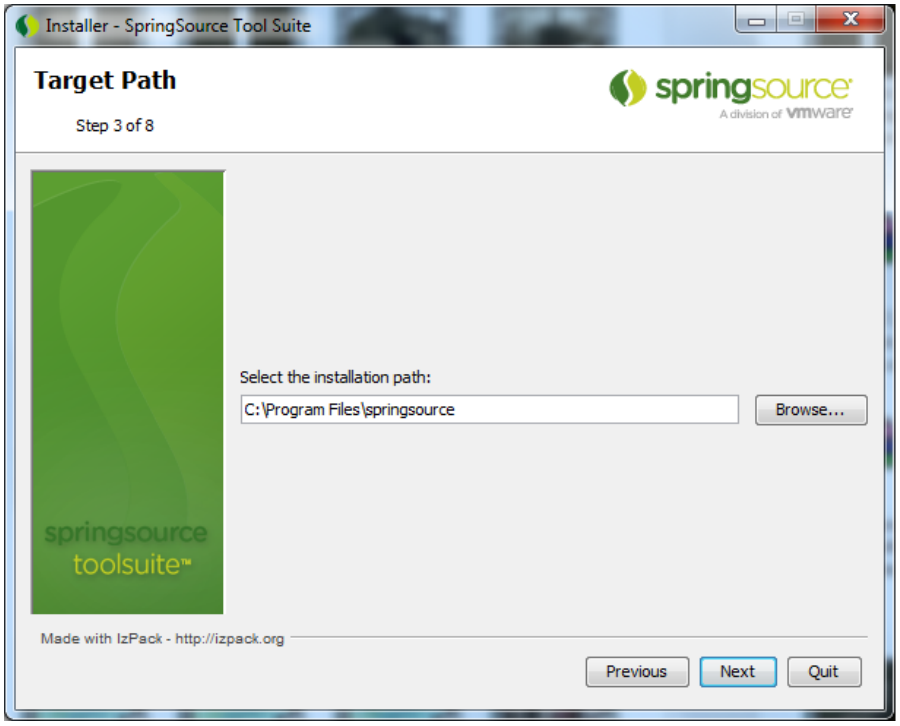

Figura 16. Selección del directorio de la instalación.

Escogemos la ubicación donde se instalara, es importante recordarla por que puede ser necesaria para instalar algunos plugins.

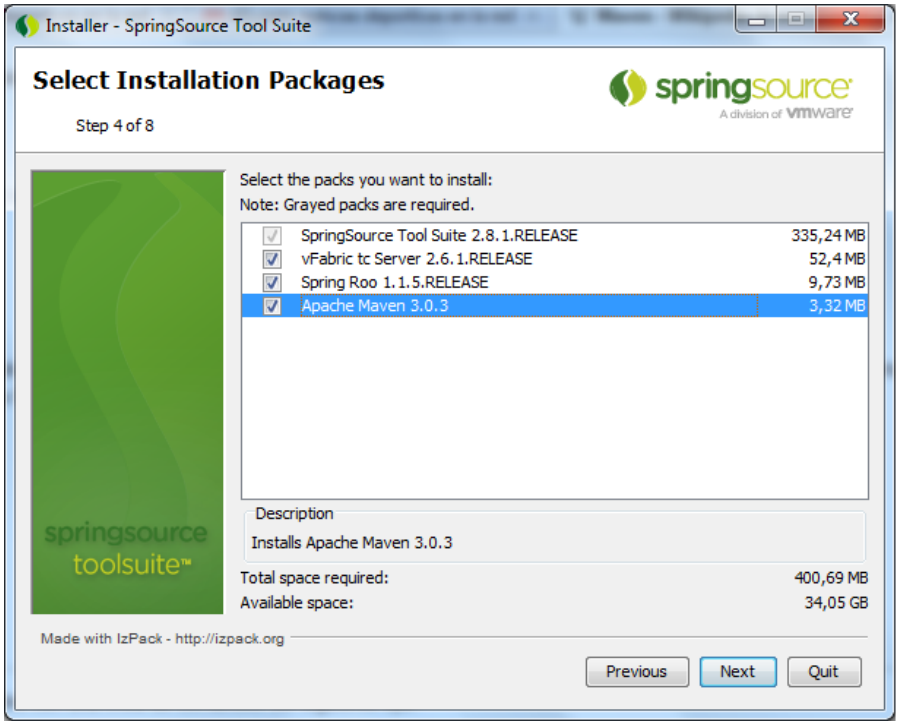

Figura 17. Selección de los componentes a instalar.

Ahora elegimos las características que deseamos instalar, por defecto se instalan todas.

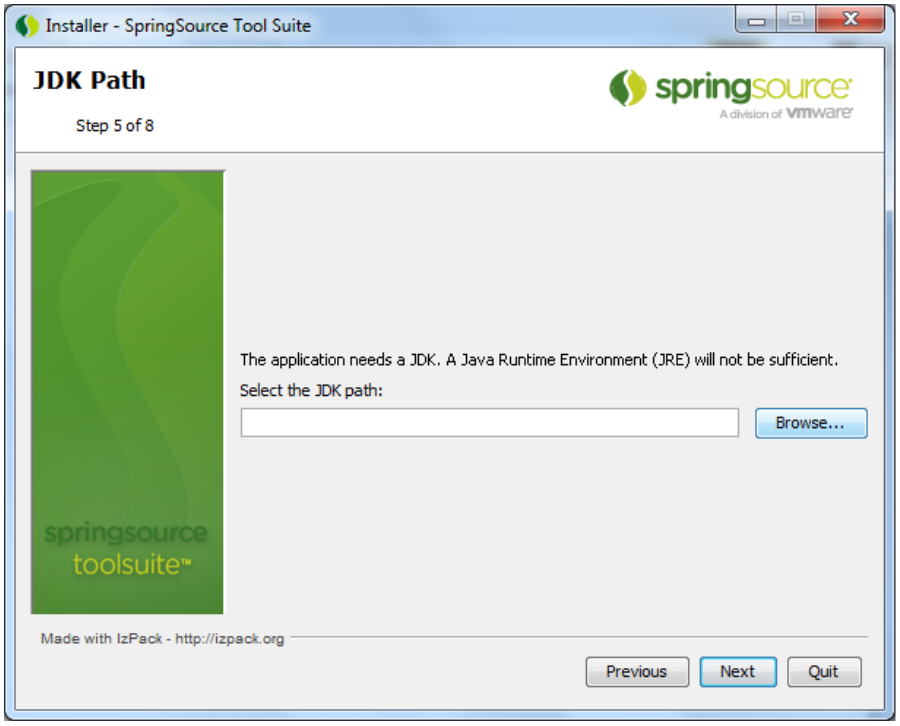

Figura 18. Selección del directorio JDK.

Ahora debemos decirle al programa donde se encuentra la JDK que hemos instalado previamente.

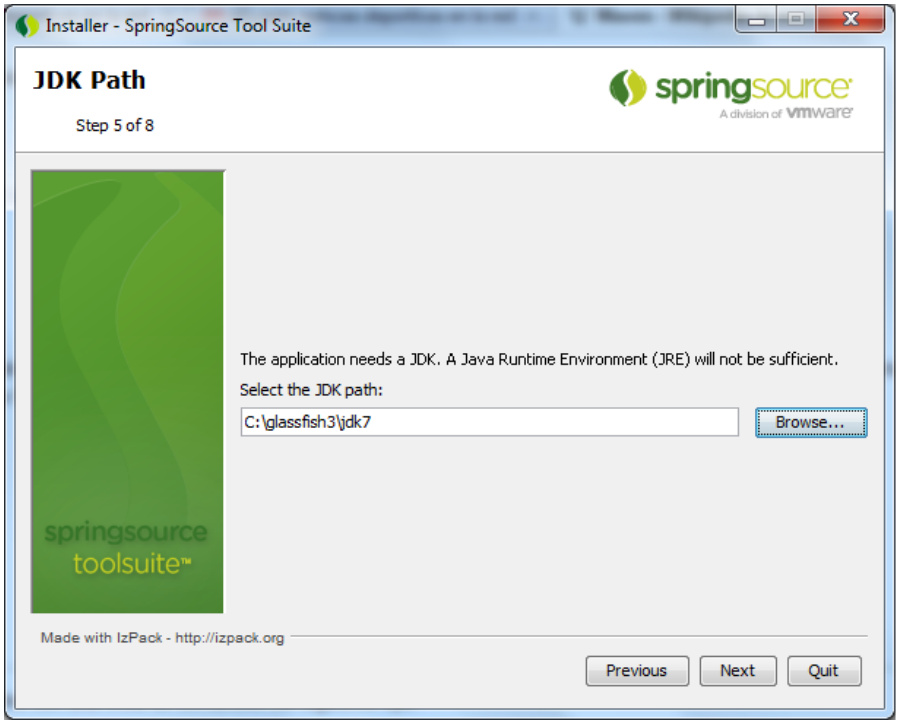

Figura 19. Directorio de JDK seleccionado.

Seleccionamos la carpeta y Next.

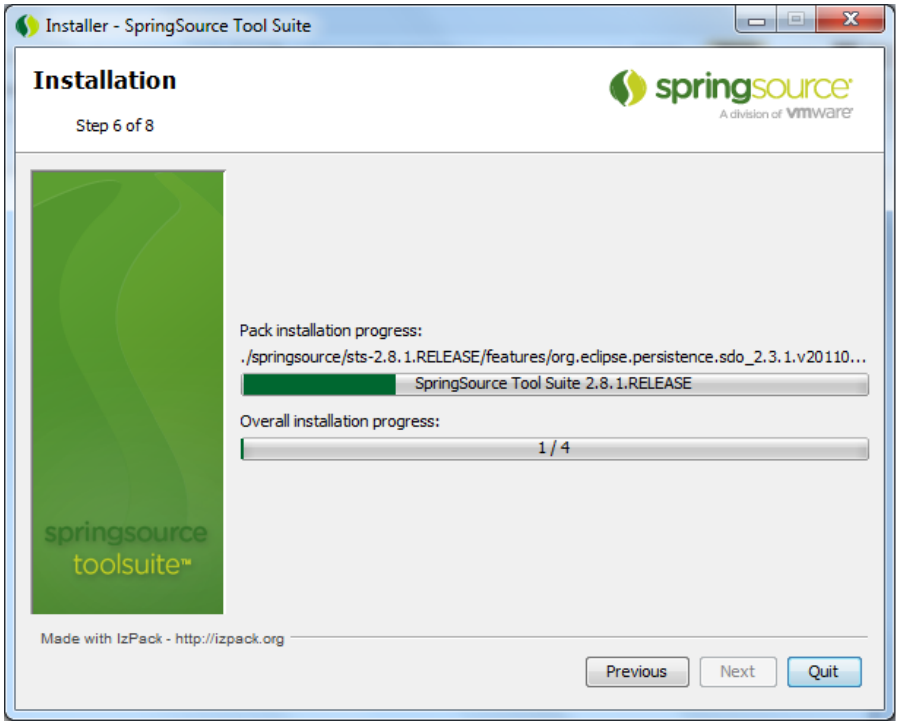

Figura 20. Progreso de la instalación.

Spring ToolSuite empezara la instalación de los componentes que hayamos seleccionado.

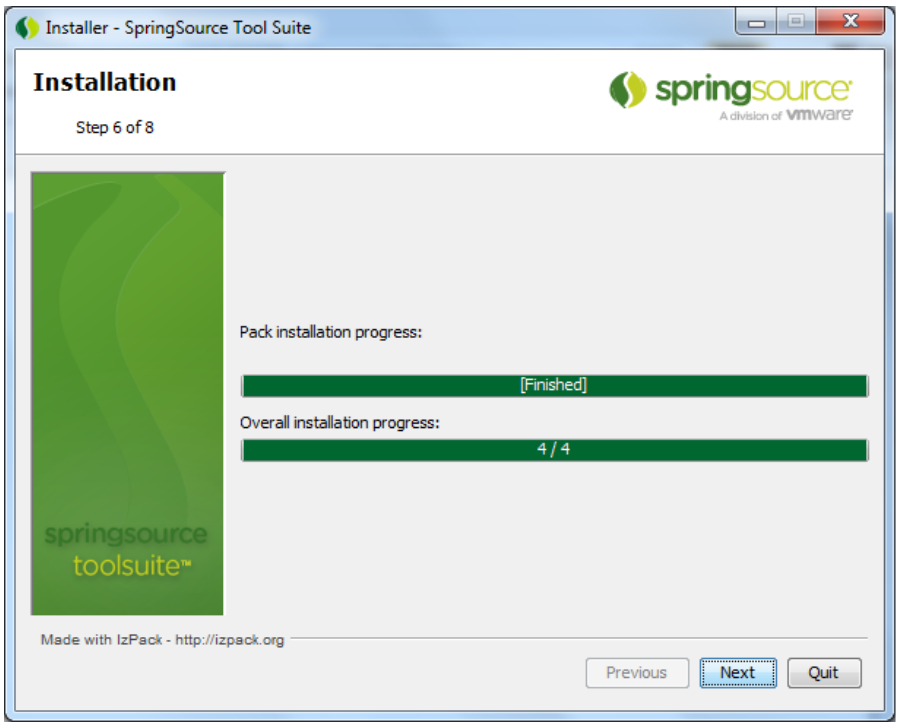

Figura 21. Progreso de la instalación finalizado.

Cuando finalice correctamente presionamos el botón de Next.

Figura 22. Selección de los permisos para los usuarios.

Ahora nos preguntara si deseamos crear un acceso directo y donde, además de tener que especificar si el programa será utilizado por todos los usuarios o solamente por el actual, por defecto nos crea un acceso en el menú de inicio y permisos para todos los usuarios, así que lo dejaremos así.

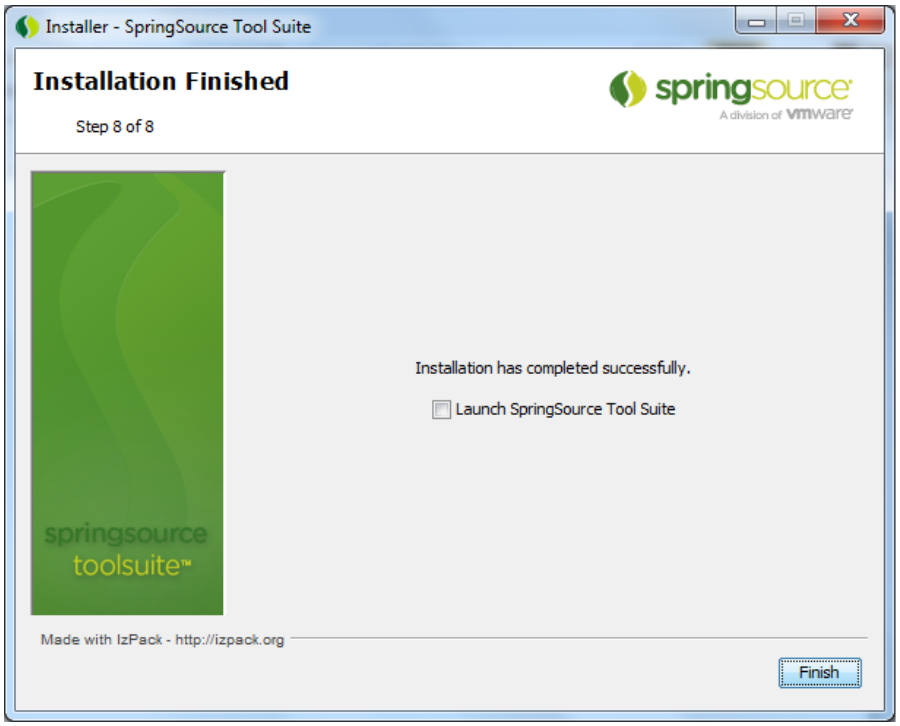

Figura 23. Instalación finalizada.

Por último finalizaremos la instalación y nos pedirá si deseamos arrancar el programa.

5 Instalación de “GlassFish” en Eclipse

Ahora vamos a mostrar como instalar “GlassFish” integrado en Eclipse, lo que nos facilitara mucho el trabajo.

Primero vamos a la pestaña window y seleccionamos preferencias.

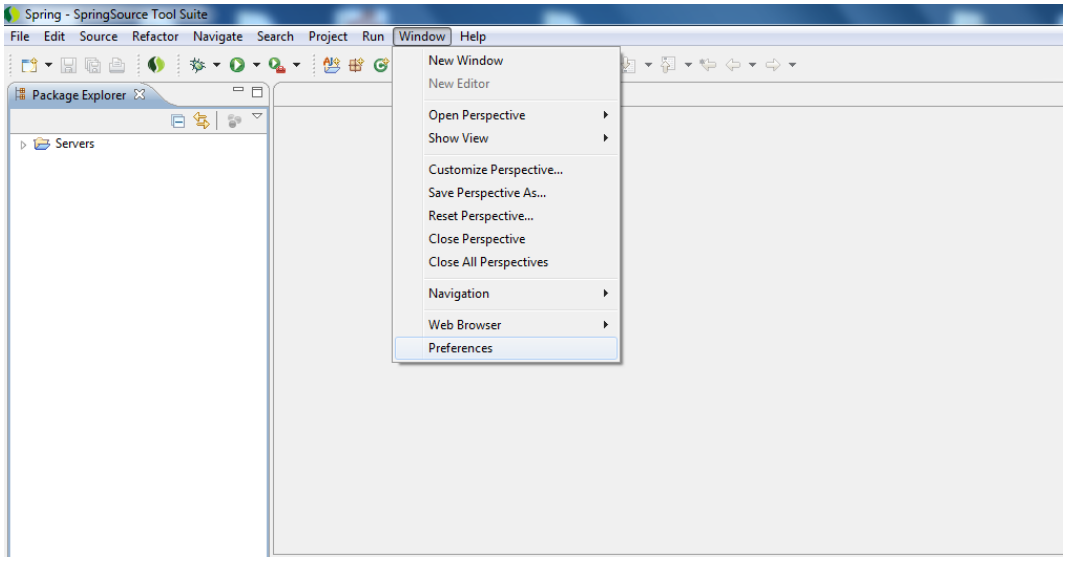

Figura 24. Selección de la pestaña preferencias.

A continuación Server, runtimeEnviorments y Add.

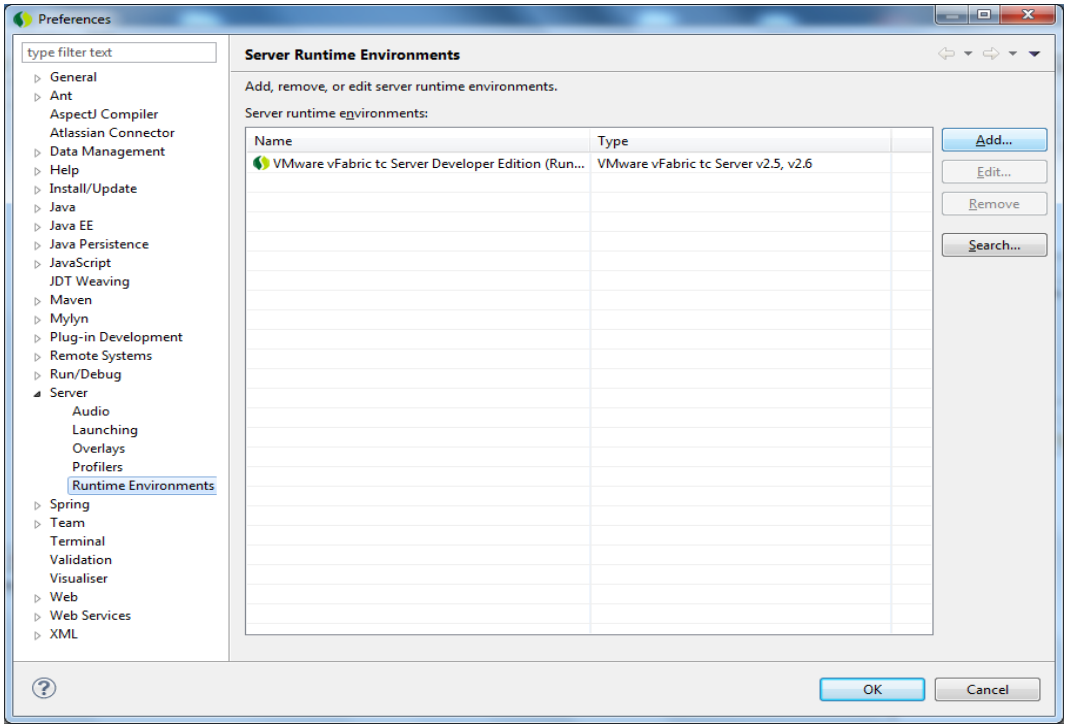

Figura 25. Selección del entorno.

Seguidamente seleccionamos Download additional server adapters ya que “GlassFish” no está cargado predeterminadamente.

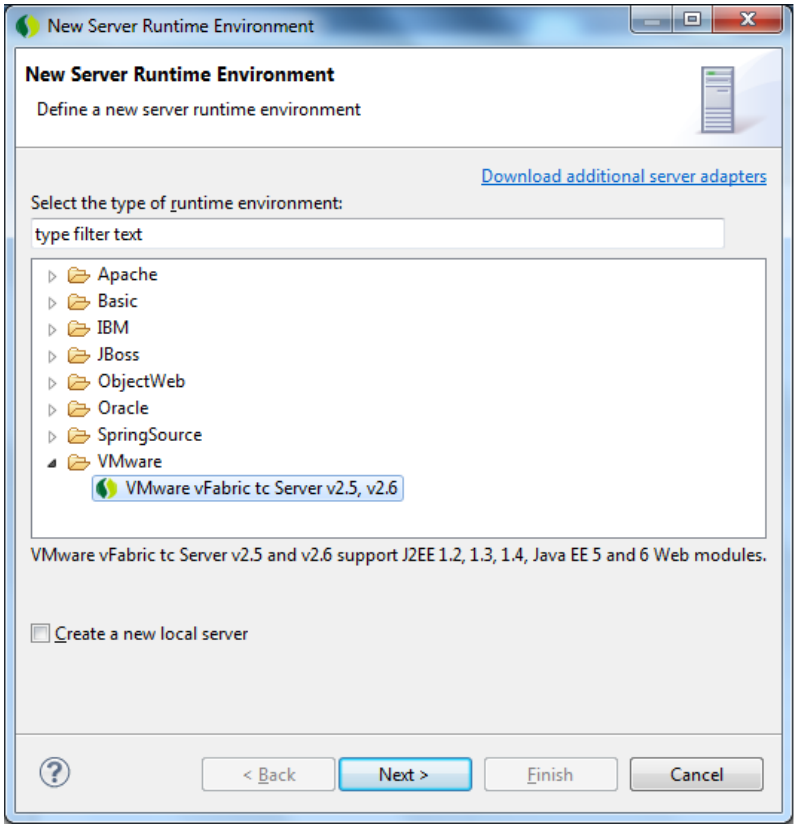

Figura 26. Selección de descarga de servidores.

Ahora seleccionamos “GlassFish” y Next.

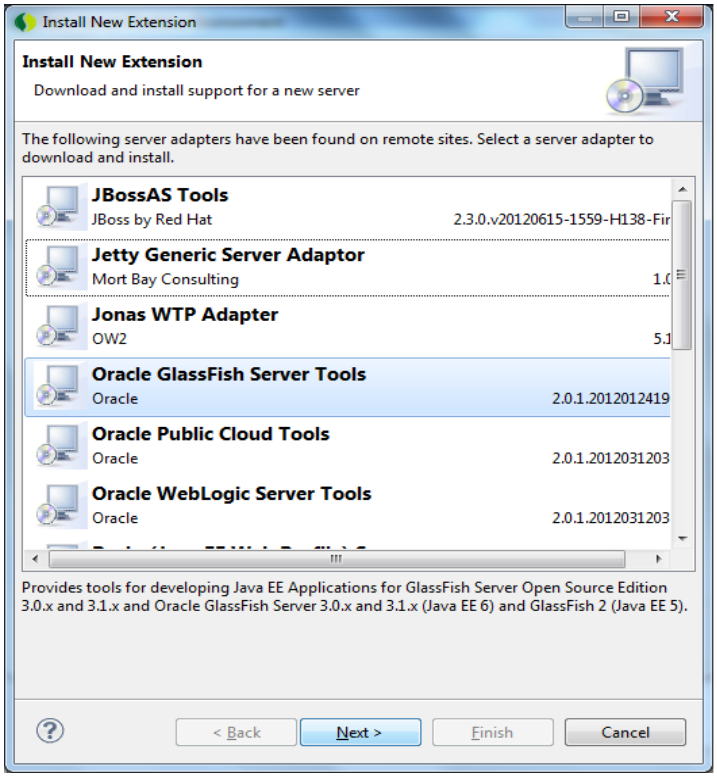

Figura 27. Selección del driver del servidor.

Aceptamos la licencia.

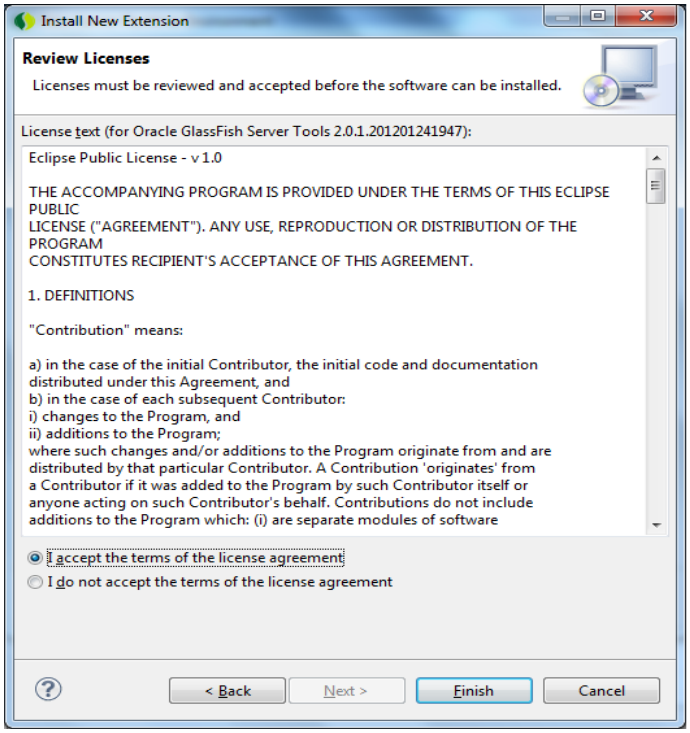

Figura 28. Aceptación de la licencia.

Y le damos a OK para que empiece la descarga.

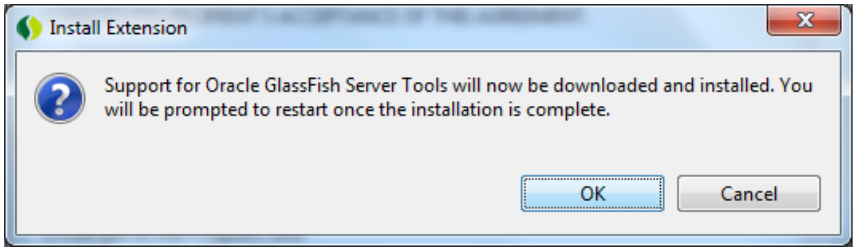

Figura 29. Aceptación de la descarga.

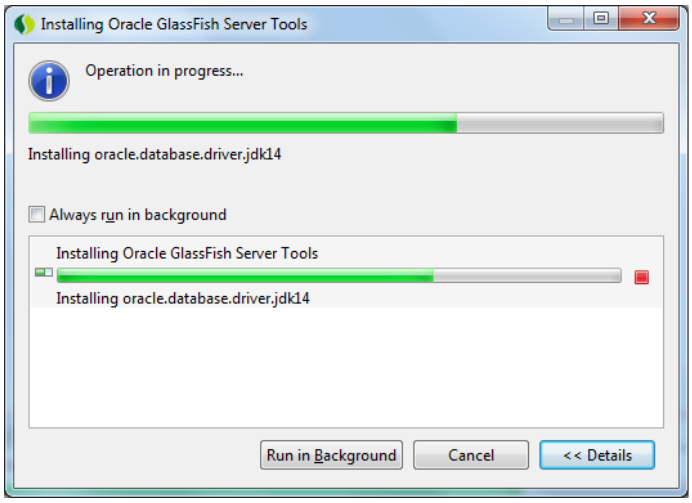

Figura 30. Descarga del servidor.

Ahora tenemos que reiniciar Eclipse.

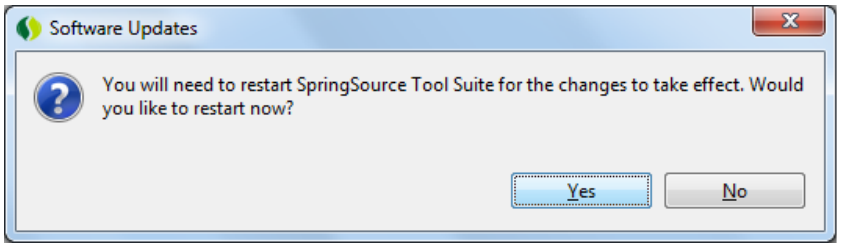

Figura 31. Reinicio de Eclipse.

Después de reiniciar vamos a la pestaña servers y añadimos un servidor.

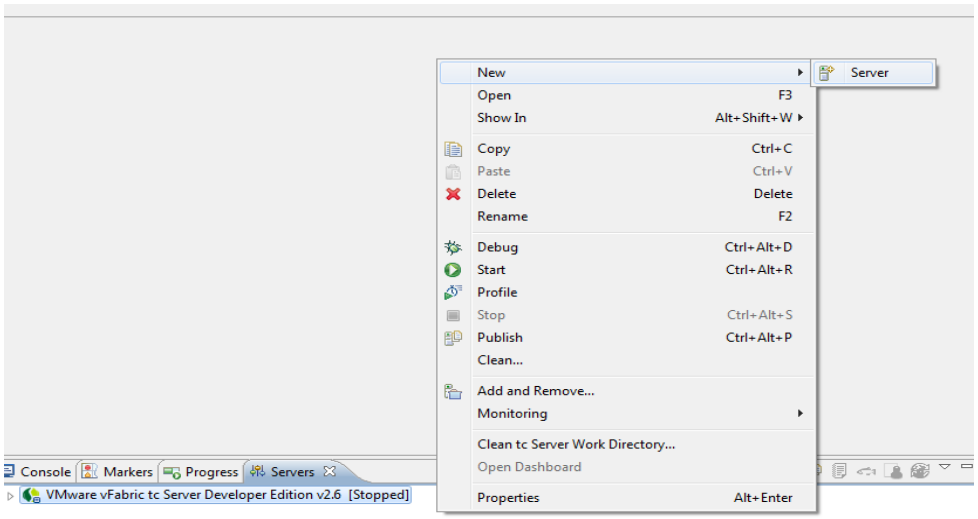

Figura 32. Nuevo servidor.

Ahora ya tenemos "GlassFish" disponible, lo seleccionamos y Next.

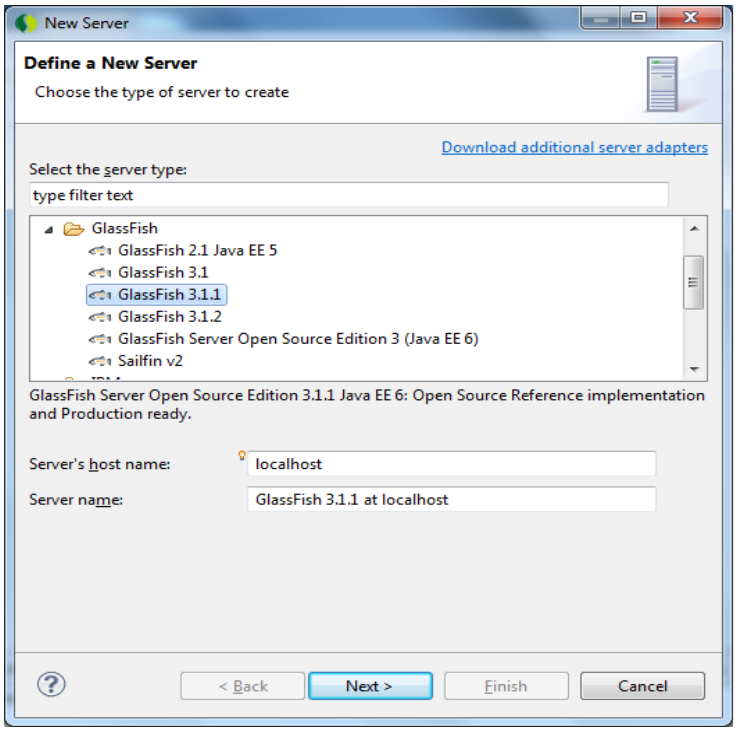

Figura 33. Selección del servidor a instalar.

Ahora seleccionamos como entorno el JDK que tengamos instalado, elegimos el directorio e instalamos.

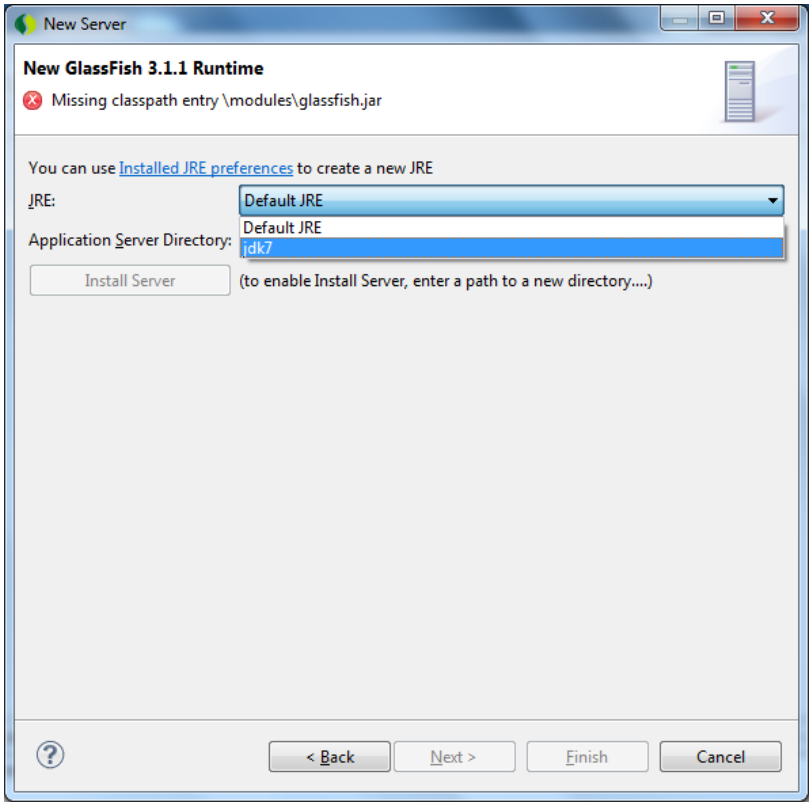

Figura 34. Selección del JRE para el servidor.

Y empezara la descarga.

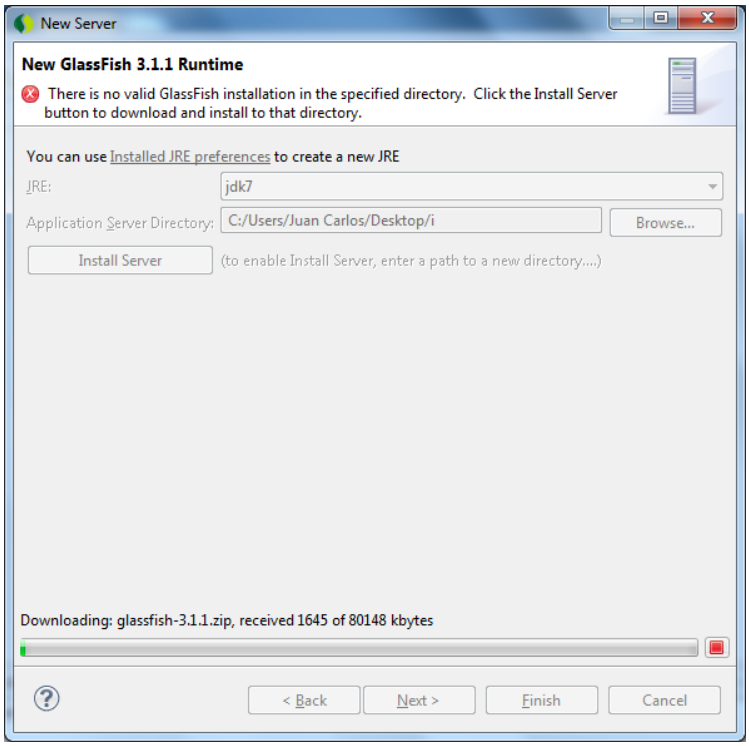

Figura 35. Descarga del servidor "GlassFish".

Ahora le damos a Next.

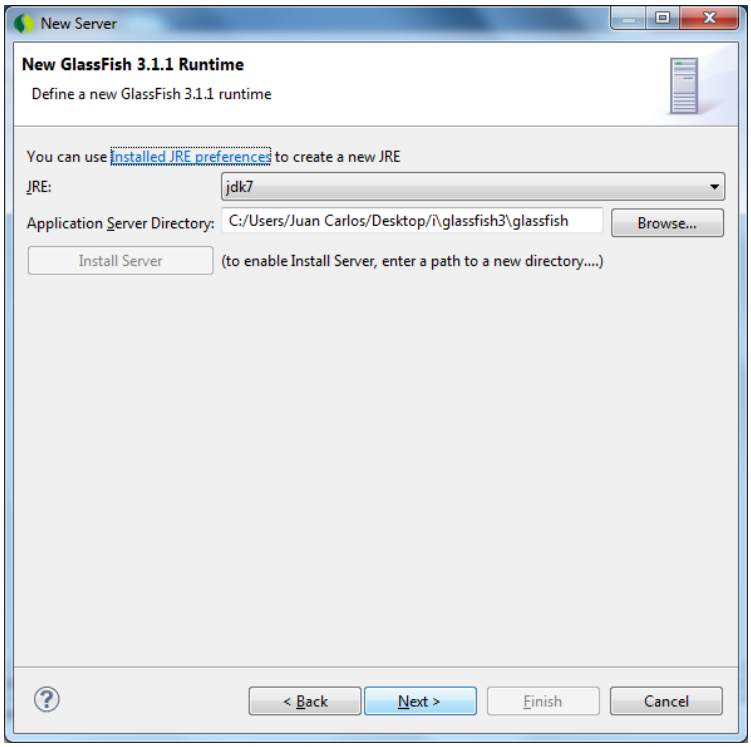

Figura 36. Instalación del servidor.

Elegimos una contraseña y finalizamos.

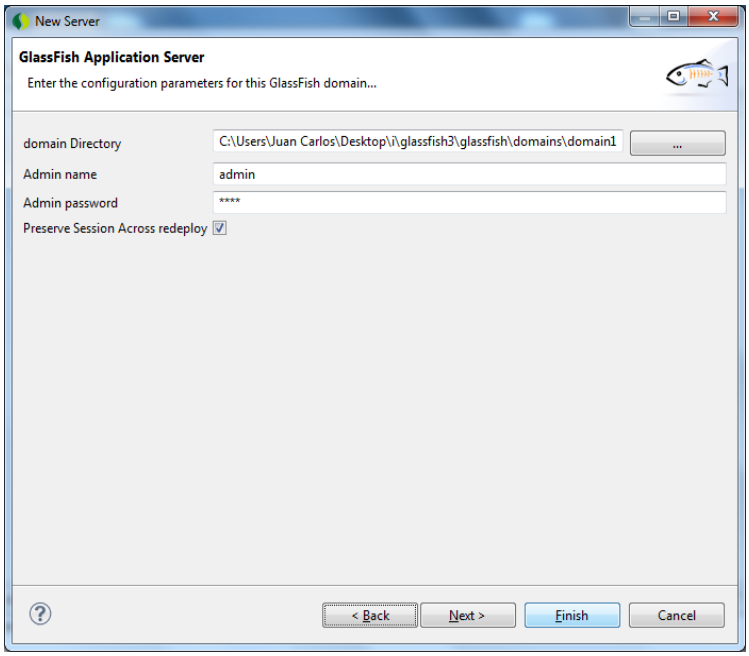

Figura 37. Introducimos usuario y contraseña

Y ya tenemos disponible “GlassFish” en Eclipse.

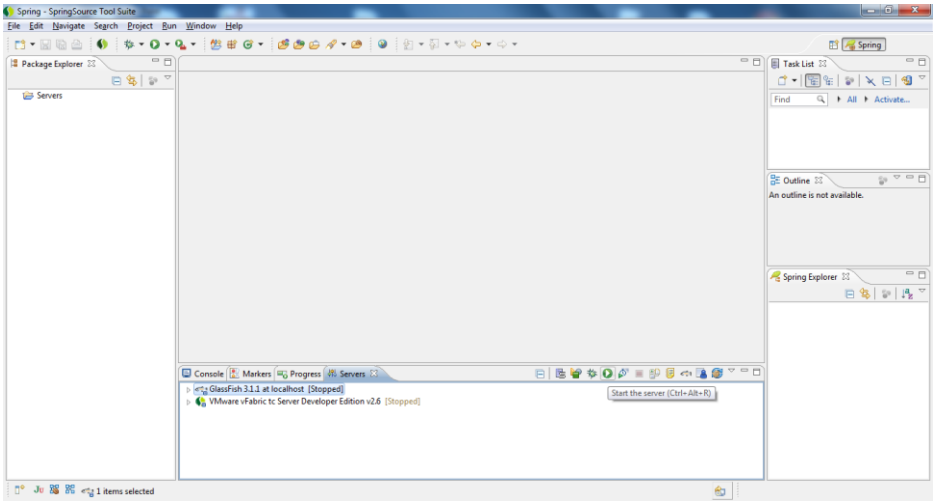

Figura 38. Servidor "Glassfish" disponible.

Capitulo 4

1 Introducción

En este capítulo vamos a explicar la autentificación Acegi que está incluida en Spring como un modulo adicional llamado Spring security, vamos a explicar las distintas maneras de autentificarse que hay con este modulo. También vamos a ver la cadena de filtros por los que pasa una validación para ser acepta por Spring security.

2 Autenticación

La autentificación en Spring se hace mediante Acegi que es un proyecto que se creó en 2003 por Ben Alex bajo licencia Apache en Marzo de 2004. Posteriormente fue incorporado a Spring.

Con el fin de mejorar la seguridad a los recursos de la aplicación Acegi utiliza el objeto autentificación (Authentication) que es donde almacena el nombre de usuario y la contraseña así como los roles perteneciente a dicho usuario, para poder determinar qué tipo de recursos serán accesibles para el usuario. En Acegi disponemos de 3 tipos de autentificación mediante un archivo que deberá ser configurado, mediante las tablas que nos proporciona Acegi o con base de datos propia.

Este objeto autentificación es creado por y validado por el gestor de autentificación (AuthenticationManager) y el acceso a los recursos es controlado por el gestor de decisiones de acceso (AccesDecisionManager).

Toda la configuración de Acegi se realiza mediante la asociación de JavaBeans a través de archivos XML.

Acegi usa cuatro filtros para manejar la seguridad:

1. El filtro del proceso de autentificación (AuthentificationProcessingFilter) maneja la petición y comprueba la autentificación, utiliza el comprobador de petición de autentificación (Authentication Request Check) y el gestor de autentificación.

2. HttpSessionContextIntegrationFilter mantiene el objeto autentificación entre varias peticiones y se lo pasa al gestor de autentificación y al gestor de decisiones de acceso si es necesario.

3. El filtro de excepciones (ExceptionTranslationFilter) comprueba si existe el objeto autentificación y maneja las excepciones de seguridad y ejecuta la acción apropiada. Este filtro depende del filtro interceptor de seguridad (FilterSecurityInterceptor) para hacer su trabajo.

4. El filtro interceptor de seguridad (FilterSecurityInterceptor) controla el acceso a determinados recursos y el chequeo de autorización. Conoce que recursos son seguros y quien puede acceder a ellos. El filtro interceptor de seguridad usa el gestor de autentificación y el gestor de decisiones de acceso para hacer su trabajo.

Haciendo uso de la inyección de dependencia de Spring estas clases delegan el trabajo más pesado a otras clases.

3 La cadena de filtros.

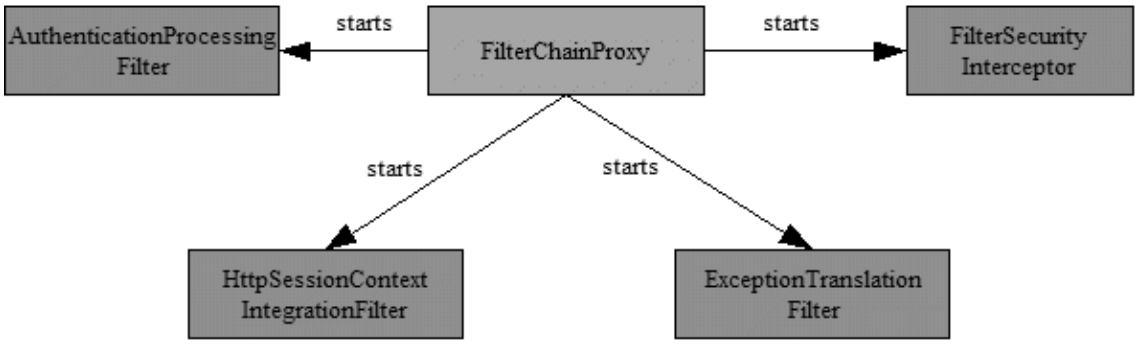

Figura 39. Cadena de filtros

Los cuatro filtros son inicializados por la cadena de filtros del proxy (FilterChainProxy) este proxy está configurado en el archivo de configuración XML y todos los demás filtros que se vayan a utilizar serán agregados a su lista de configuración. La siguiente configuración detalla los filtros que serán inicializados.

```
<bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
 <property name="filterInvocationDefinitionSource">
                <value>
                CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
                PATTERN_TYPE_APACHE_ANT
                /**=HttpSessionContextIntegrationFilter, formAuthenticationProcessingFilter,
                exceptionTranslationFilter, filterSecurityInterceptor
                </value>
 </property>
</bean>
```

Listado 2. Configuración de la cadena de filtros del proxy.

Filtro del proceso de autentificación (AuthenticationProcessingFilter)

Es el primer filtro por donde pasara la petición HTTP. Este filtro se encarga de manejar la petición de autentificación, como validar el usuario y contraseña. Su archivo de configuración XML es el siguiente:

```
<bean id="formAuthenticationProcessingFilter"
                        class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilter">
                <property name="filterProcessesUrl">
                        <value>/j_acegi_security_check</value>
                </property>
                <property name="authenticationFailureUrl">
                        <value>/loginFailed.html</value>
                </property>
                <property name="defaultTargetUrl">
                        <value>/</value>
                </property>
                <property name="authenticationManager">
                        <ref bean="authenticationManager" />
```

```
        </property>
</bean>
```

Listado 3. Configuración del filtro del proceso de autentificación.

El bean del filtro es del tipo "org.acegisecurity.ui.webapp.AuthenticationProcessingFilter". Este bean se utiliza en los formularios de login. Tiene tres propiedades de configuración: la URL ("filterProcessUrl"), la página a donde redirigir si falla el login ("authenticationFailureUrl"), y la llamada al gestor de autentificación. El filtro del proceso de autentificación se especializa en el manejo de las peticiones de autenticación. Puede devolver un diálogo de login o redirigirnos a la página de login, pero no es él mismo quien se encarga de verificar si es correcta la combinación de nombre de usuario y contraseña, y por lo tanto no se configura en este filtro.

HttpSessionContextIntegrationFilter

Su única tarea es la de propagar el objeto creado en la autentificación a través de todas las solicitudes. Lo hace envolviendo el objeto en un objeto hilo local y transmite este envoltorio a través de los demás filtros de la cadena.

```
<bean id="httpSessionContextIntegrationFilter"
  class="org.acegisecurity.context.HttpSessionContextIntegrationFilter">
</bean>
```

Listado 4. Configuración del filtro HttpSessionContextIntegrationFilter.

Filtro traductor de excepciones (ExceptionTranslationFilter).

Forma junto al filtro interceptor de seguridad los principales filtros de Acegi. Este filtro toma cualquier error de autenticación o autorización (bajo la forma de un excepción de tipo excepción de seguridad Acegi) y luego puede ejecutar una de las siguientes dos acciones.

Si la excepción fue generada por la ausencia del objeto de autentificación nos re direccionara al punto de entrada de la autentificación configurado para dicha excepción. En cambio si la excepción fue generada por el filtro interceptor de la seguridad, el filtro traductor de excepciones lanzara un error SC_FORBIDDEN (HTTP 403) al navegador web con el mensaje de acceso no autorizado.

```
<bean id="exceptionTranslationFilter"
                class="org.acegisecurity.ui.ExceptionTranslationFilter">
        <property name="authenticationEntryPoint">
                <ref bean="formLoginAuthenticationEntryPoint" />
        </property>
</bean>
```

Listado 5. Configuración del filtro traductor de excepciones.

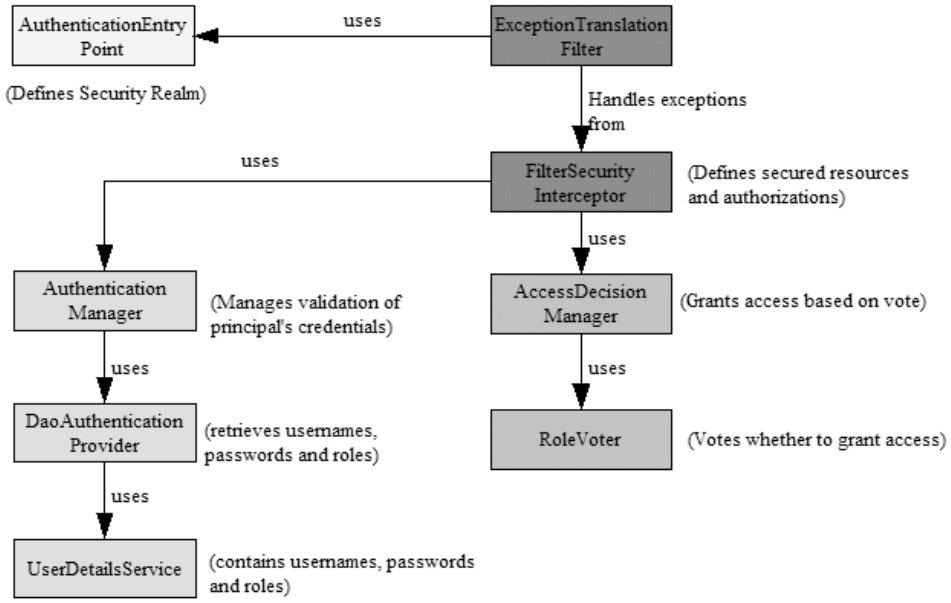

Figura 40. Ruta de peticiones

El filtro interceptor de seguridad (FilterSecurityInterceptor).

Es el encargado de administrar los recursos que queremos que sean seguros.

```
<bean id="filterSecurityInterceptor"
                class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
        <property name="authenticationManager">
                <ref bean="authenticationManager" />
        </property>
        <property name="accessDecisionManager">
                <ref bean="accessDecisionManager" />
        </property>
        <property name="objectDefinitionSource">
                <value>
                        CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
                        PATTERN_TYPE_APACHE_ANT
                        /secure/admin/*=ROLE_ADMIN
                        /secure/app/*=ROLE_USER
                </value>
        </property>
</bean>
```

Listado 6. Configuración del filtro interceptor de seguridad.

En Acegi los recursos que son seguros se llaman "Object definitions" por lo tanto la etiqueta ObjectDefinitionSource deberá tener patrones URL y directivas que sean seguros junto a roles que tendrán acceso a ello, es decir deberá tener directorios y funciones seguros junto a los tipos de usuario que podrán tener acceso a estos.

En esta configuración XML vemos dos directivas:

- CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON le dice a Acegi que convierta las peticiones URL a minúscula, para que el mecanismo de seguridad no tenga ningún

inconveniente ni pueda ser superado simplemente introduciendo una URL ligeramente diferente en el navegador.

- PATTERN_TYPE_APACHE_ANT hace que sea más fácil definir los patrones de URL que estarán.

Gestor de autentificación (AuthenticationManager).

```
<bean id="authenticationManager"
        class="org.acegisecurity.providers.ProviderManager">
    <property name="providers">
        <list>
            <ref bean="daoAuthenticationProvider" />
        </list>
    </property>
</bean>
<bean id="daoAuthenticationProvider"
        class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
    <property name="authenticationDao">
        <ref bean="authenticationDao" />
    </property>
</bean>
<bean id="authenticationDao"
        class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
    <property name="userMap">
        <value>
            jklaassen=4moreyears,ROLE_ADMIN
            bouerj=ineedsleep,ROLE_USER
        </value>
    </property>
</bean>
```

Listado 7. Configuración del gestor de autentificación.

El gestor de autentificación es del tipo proveedor de gestores por lo que utiliza el proveedor de autentificación, este es el encargado de verificar la combinación de usuario y contraseña introducida y devolver los roles asociados a dicho usuario, para ello utiliza AuthenticationDao que es como un registro donde se encuentran los nombres de usuarios y contraseñas con sus correspondientes roles (admin, user,…) en este ejemplo estamos utilizando la validación mediante archivo por lo que se han definido dos usuarios jklaassen con contraseña 4moreyears y rol de administrador y bouerj con contraseña ineedsleep con rol de usuario.

Gestor de decisión de acceso (AccessDecisionManager).

El anterior filtro comprobaba la combinación de usuario y contraseña pero no daba autorización, esto corresponde al gestor de decisión de acceso, este comprueba la información del usuario y decide si puede acceder al recurso demandado o no, para esto utiliza el votante (Voter):

```
<bean id="accessDecisionManager"
        class="org.acegisecurity.vote.UnanimousBased">
        <property name="decisionVoters">
            <list>
                <ref bean="roleVoter" />
            </list>
        </property>
</bean>
<bean id="roleVoter" class="org.acegisecurity.vote.RoleVoter">
        <property name="rolePrefix">
            <value>ROLE_</value>
```

```
        </property>
</bean>
```

Listado 8. Configuración del gestor de decisión de acceso.

Se deberán especificar para cada votante los diferentes roles que serán manejados por el especificando el prefijo del rol. Es posible vincular varios votantes con un gestor de decisión de acceso y que varios proveedores de gestores pueden vincularse a un único gestor de autentificación por lo que es posible configurar Acegi para consultar varios usuarios y contraseñas que estén registrados (es decir que puede ser una combinación de LDAP, base de datos y registros de dominio NT) con varios nombres de roles configurados y utilizados por varios votantes.

Punto de entrada de la autentificación (AuthenticationEntryPoint).

Para terminar la configuración solo queda definir el punto de entrada de la autentificación que será el punto de entrada, es decir la pagina donde nos logeamos, si el filtro interceptor de archivos de seguridad (FileSecurityInterceptor) como hemos mencionado antes no detecta un objeto de autentificación, la aplicación de filtros de seguridad cederá el control al punto de entrada de la autentificación.

```
        <bean id="formLoginAuthenticationEntryPoint"
        class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilterEntryPoint">
        <property name="loginFormUrl">
                <value>/login.jsp</value>
        </property>
        <property name="forceHttps">
                <value>false</value>
                </property>
</bean>
```

Listado 9. Configuración del filtro del punto de entrada de la autentificación.

Este filtro es muy sencillo de configurar solo se necesita la URL de la pagina y si se desea que el usuario y la contraseña sean encriptados antes de ser enviados se debe poner la propiedad forceHttps a true.

Autentificación por base de datos mediante JDBC

En el ejemplo anterior se utilizaba autentificación en memoria DAO (InMemoryDaoImpl) como AuthenticationDao es decir un archivo en memoria para almacenar los nombres de usuarios y contraseñas pero en la realidad es mucho más frecuente utilizar bases de datos para guardar los datos. Ahora vamos a ver la configuración utilizando la clase JdbcDaoImpl de Acegi:

```
        <bean id="userDetailsService"
                class="org.acegisecurity.userdetails.jdbc.JdbcDaoImpl">
        <property name="dataSource">
                <ref bean="dataSource"/>
        </property>
</bean>
```

Listado 10. Configuración del dataSource.

La configuración del dataSource es la estándar de Spring:

```
        <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/springweb_auth_db</value>
```

```
        </property>
        <property name="username">
                <value>j2ee</value>
        </property>
        <property name="password">
                <value>password</value>
        </property>
</bean>
```

Listado 11. Configuración completa del dataSource.

Ahora llegado a este punto tenemos dos posibilidades, la primera, más complicada, es la de utilizar nuestras propias tablas personalizados por lo que deberemos crear nosotros las consultas adecuadas y mapear las columnas para el AuthenticationDao, la segunda es utilizar el esquema de tablas de Acegi:

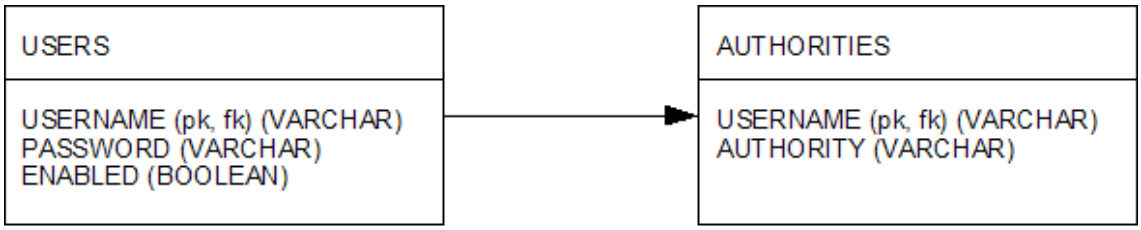

Figura 41. Tablas de la BD

La columna ENABLED puede ser de tipo carácter o cadena de caracteres conteniendo los valores "true" y "false". La columna AUTHORITY contendrá los nombres de roles que hayamos definido en los archivos de configuración.

Proveedor completo de autenticación (Custom AuthenticationProvider).

En el ejemplo anterior se acceda a la base de datos directamente, sino queremos esto podemos configurar un DAO personalizado:

Primero declaramos el bean (en este caso es la clase CustomAuthenticationDAO):

```
<beanid="customAuthenticationDao"
                class="test.mypackage.mydao.security.CustomAuthenticationDAO" />
```

Listado 12. Declaración del CustomAuthenticationDAO.

Luego el código de configuración XML para el gestor de autentificación y sus dependencias:

```
<bean id="authenticationManager"
                class="org.acegisecurity.providers.ProviderManager">
        <property name="providers">
                <list>
                        <ref bean="daoAuthenticationProvider" />
                </list>
        </property>
</bean>
<bean id="daoAuthenticationProvider"
                class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
        <property name="authenticationDao">
                <ref bean="customAuthenticationDao" />
        </property>
</bean>
```

Listado 13. Declaración del gestor de autentificación y sus dependencias.

Y finalmente el código de nuestra clase Java es el siguiente:

```
public class CustomAuthenticationDAO implements UserDetailsService {

        public UserDetails loadUserByUsername(String userName)
                        throws UsernameNotFoundException, DataAccessException {


        //Este método carga un usuario desde la base de datos a partir del nombre de
        //usuario
        final User user = AccountFacade.loadUserByUsername(userName);
        if (user == null)
                throw new UsernameNotFoundException("No existe el nombre de usuario");
        return new UserDetails() {

         private static final long serialVersionUID = -2868501916277412090L;

                public GrantedAuthority[] getAuthorities() {
                        return new GrantedAuthority[] { new GrantedAuthority() {

                                public String getAuthority() {
                                        if (user.isSystemAdmin())
                                                return "ROLE_ADMIN";
                                        else if (user.isAccountAdmin())
                                                return "ROLE_ACCOUNTADMIN";
                                        else
                                                return "ROLE_USER";
                                }
                        } };
                }

                public String getPassword() {
                        return user.getPassword();
                }

                public String getUsername() {
                        return user.getUsername();
                }

                public boolean isAccountNonExpired() {
                        return true;
                }

                public boolean isAccountNonLocked() {
                        return true;
                }

                public boolean isCredentialsNonExpired() {
                        return true;
                }

                public boolean isEnabled() {
                        return user.isActivated();
                }
        };
}
}
```

Listado 14. Clase Java para la autenticación.

Como se observa en el código anterior nuestra clase personalizada implementa la interfaz UserDetailsService para lo cual implementamos el método loadUserByUsername que retorna algo de tipo UserDetails (este método será

invocado por Acegi, el cual pasará el nombre de usuario como parámetro). Al retornar algo de este tipo lo que debemos hacer es devolver en los métodos heredados los datos correspondientes que tenemos en nuestro objeto User.

Logout

Agregar la funcionalidad de logout es muy simple. Sólo hay que configurar un filtro. Éste es de tipo org.acegisecurity.ui.logout.LogoutFilter.

```
<bean id="logoutFilter"
                        class="org.acegisecurity.ui.logout.LogoutFilter">
 <constructor-arg value="/index.jsp" />
 <constructor-arg>
  <list>
   <bean class="org.acegisecurity.ui.logout.SecurityContextLogoutHandler" />
            </list>
 </constructor-arg>
  <property name="filterProcessesUrl">
   <value>/logout.jsp</value>
            </property>
</bean>
```

Listado 15. Configuración del logout.

No debemos olvidar agregar el filtro LogoutFilter a la cadena de filtros:

```
<bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
 <property name="filterInvocationDefinitionSource">
            <value>
            CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
            PATTERN_TYPE_APACHE_ANT
            /**=httpSessionContextIntegrationFilter, formAuthenticationProcessingFilter,
            exceptionTranslationFilter, filterSecurityInterceptor, logoutFilter
            </value>
 </property>
</bean>
```

Listado 16. Agregar el filtro logout a la cadena de filtros.

Filtro recuérdame.

EL filtro principal que hay que configurar es RememberMeProcessingFilter, agregar la referencia del filtro en el punto de acceso y el filtro a la cadena de filtros:

```
<bean id="rememberMeProcessingFilter"
            class="org.acegisecurity.ui.rememberme.RememberMeProcessingFilter">
            <property name="rememberMeServices">
                        <ref local="rememberMeServices" />
            </property>
            <property name="authenticationManager">
                        ref="authenticationManager" />
            </property>
</bean>
<bean id="rememberMeServices"
            class="org.acegisecurity.ui.rememberme.TokenBasedRememberMeServices">
            <property name="userDetailsService">
                        <ref local="customAuthenticationDao" />
            </property>
            <property name="key" value="myKey" />
            <property name="parameter" value="rememberMe" />
</bean>
```

```
<bean id="rememberMeAuthenticationProvider"
class="org.acegisecurity.providers.rememberme.RememberMeAuthenticationProvider">
        <property name="key">
                <value>myKey</value>
        </property>
</bean>
```

Listado 17. Filtro recuérdame.

La referencia en el punto de acceso:

```
<bean id="formLoginAuthenticationEntryPoint"
        class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilterEntryPoint">
        <property name="loginFormUrl">
                <value>/login.jsp</value>
        </property>

        <property name="rememberMeServices">
                <ref bean="rememberMeServices" />
        </property>
</bean>
```

Listado 18. Agregar referencia al punto de acceso.

Agregar el filtro a la cadena de filtros:

```
        <bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
         <property name="filterInvocationDefinitionSource">
        <value>
        CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
        PATTERN_TYPE_APACHE_ANT
        /**=httpSessionContextIntegrationFilter,formAuthenticationProcessingFilter,
        exceptionTranslationFilter,filterSecurityInterceptor,logoutFilter,
        rememberMeProcessingFilter
        </value>
        </property>
      </bean>
```

Listado 19. Agregar referencia del filtro recuérdame.

ChannelProcessingFilter (SSL)

Por último vamos a ver como encriptar los datos utilizando SSL. Para ellos debemos configurar dos beans: ChannelProcessingFilter y ChannelDecisionManager. En el primero (el filtro) debemos especificar qué recursos requerirán SSL. El otro bean es donde definimos las referencias a las clases que implementan la funcionalidad (SecureChannelProcessor y ChannelProcessingFilter):

```
<bean id="channelProcessingFilter"
        class="org.acegisecurity.securechannel.ChannelProcessingFilter" >
        <property name="channelDecisionManager" ref="channelDecisionManager"/>
        <property name="filterInvocationDefinitionSource">
                <value>
                PATTERN_TYPE_APACHE_ANT
                /login.jsp=REQUIRES_SECURE_CHANNEL
                /**=REQUIRES_INSECURE_CHANNEL
   </value>
 </property>
<bean id="channelDecisionManager"
        class="org.acegisecurity.securechannel.ChannelDecisionManagerImpl">
        <property name="channelProcessors">
                <list>
    <bean class="org.acegisecurity.securechannel.SecureChannelProcessor"/>
    <bean class="org.acegisecurity.securechannel.InsecureChannelProcessor"/>
```

```
    </list>
  </property>
</bean>
```

Listado 20. Configuración del filtro SSL.

Agregamos ChannelProcessingFilter a la cadena de filtros:

```
<bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
 <property name="filterInvocationDefinitionSource">
                <value>
                CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
                PATTERN_TYPE_APACHE_ANT
                /**=httpSessionContextIntegrationFilter,formAuthenticationProcessingFilter,
                exceptionTranslationFilter,filterSecurityInterceptor,logoutFilter,
                rememberMeProcessingFilter,channelProcessingFilter
                </value>
 </property>
</bean>
```

Listado 21. Agregar referencia del filtro SSL.

Y recordamos que en el filtro AuthenticationEntryPoint estaba la propiedad forceHttps, que ahora debemos poner a true .

```
<bean id="formLoginAuthenticationEntryPoint"
                class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilterEntryPoint">
                <property name="loginFormUrl">
                                <value>/login.jsp</value>
                </property>
                <property name="forceHttps">
                                <value>true</value>
                </property>
</bean>
```

Listado 22. Configuración de la propiedad forceHttps.

Por último solo nos quedaría instalar los correspondientes certificados en nuestro equipo.

Resumen

La autentificación Acegi en Spring permite configurar la seguridad independientemente de los demás sistemas de seguridad que ya tenga el servidor, además nos ahorra el costoso trabajo de comprobar mediante código el acceso a los recursos de la aplicación ya que sus filtros realizan este trabajo por nosotros una vez esta todo configurado correctamente, por ultimo podríamos decir que Acegi nos permite una total portabilidad sin depender de ninguna plataforma como especifica Java EE lo que lo hace recomendable para construir este tipo de aplicaciones.

Capitulo 5

1 Introducción

En este capítulo trataremos de explicar que es un modelo de capas, su función y los diferentes tipos de modelos que existen. Nos centraremos especialmente en el que vamos a utilizar en nuestras aplicaciones el modelo de 3 capas.

2 Modelo de capas

El modelo de capas es una técnica software para separar las diferentes partes de la aplicación, con el objetivo de mejorar su rendimiento, mantenimiento y sus funciones. Esta separación de las diferentes partes hace que los cambios en cualquiera de las capas no afecten o afecten poco a las otras capas en que está dividida la aplicación.

Nos vamos a centrar en dos modelos el n-capas y el más importante para nuestra aplicación web el de 3 capas:

2.1 N-capas(n-tier)

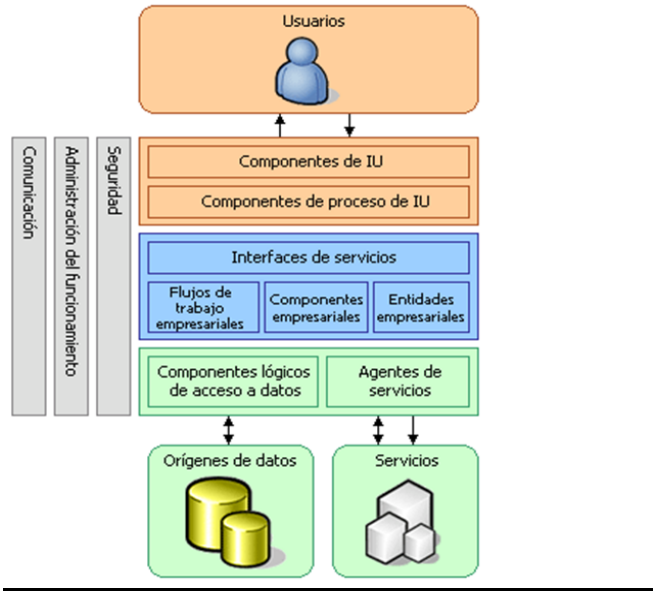

Figura 42. Modelo de capas

Las Ventajas de este modelo son:

- Desarrollo paralelo(en cada capa)
- Aplicaciones mas robustas debido al encapsulamiento
- Mantenimiento y soporte más sencillo
- Mayor flexibilidad(se pueden añadir nuevos módulos sin tener que retocar mucho)
- Alta escalabilidad.

Las desventajas:

- Mayor carga en red, debido a un mayor tráfico de red.

- Es más difícil realizar las pruebas por que tienden a intervenir más dispositivos.

En una aplicación distribuida en n-capas, los diferentes procesos están distribuidos en diferentes capas no sólo lógicas, sino también físicas. Los procesos se ejecutan en diferentes equipos, que pueden incluso residir en plataformas o sistemas operativos completamente distintos. Cada equipo posee una configuración distinta y está optimizado para realizar el papel que le ha sido asignado dentro de la estructura de la aplicación, de modo que tanto los recursos como la eficiencia global del sistema se optimicen. Esta arquitectura es más propia de los grandes sistemas.

2.2 Modelo 3 capas.

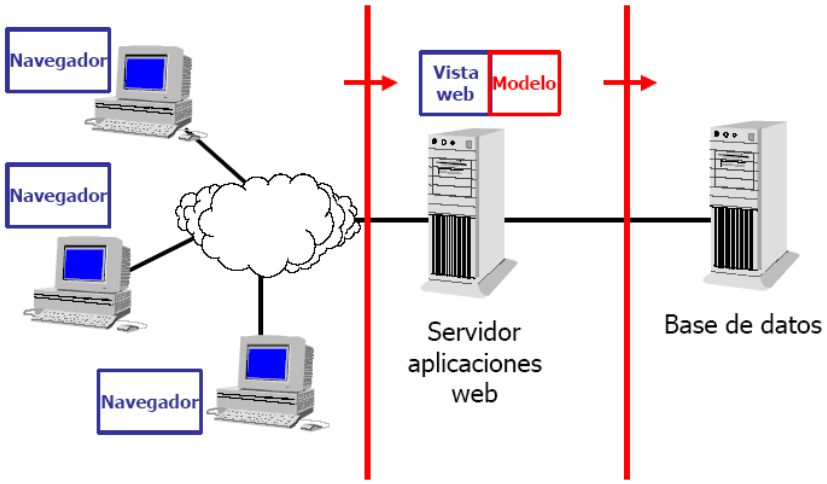

Figura 43. Modelo 3 capas

Capa de presentación: es la capa que ve el usuario, presenta el sistema, captura la información y la presenta la información al usuario en un mismo proceso. Esta capa se comunica únicamente con la capa de negocio**.**

Capa de negocio o lógica: se reciben las peticiones del usuario y se envían las respuestas tras el proceso. En esta capa se establecen todas las reglas que deben cumplirse. Se comunica con la capa de presentación para recibir las solicitudes y presentar los resultados, también se comunica con la capa de persistencia o de datos para solicitar al gestor de la base de datos para recuperar, modificar o insertar datos en la base de datos.

Capa de persistencia o de datos: es donde residen los datos y la encargada de acceder a ellos. Está formada por uno o más gestores de datos que realizan todo el almacenamiento de datos, reciben solicitudes de almacenamiento o recuperación desde la capa de negocio.

Capitulo 6

1 Introducción

En este capítulo vamos a ver el modelo vista controlador aplicado al framework Spring y las ventajas que nos ofrece Spring y sus módulos para aplicarlo a nuestros programas. También vamos a ver los distintos filtros y módulos de que dispone Spring para hacernos más fácil este tipo de implementación.

2 MVC en SPRING

El MVC es un patrón de diseño de software que separa los datos de una aplicación, la interfaz de usuario y la lógica de control de forma que cualquier modificación de algún componente del modelo tenga un mínimo impacto en los otros.

Sus componentes son:

- Modelo: representa los datos que el usuario está esperando ver.
- Vista: es el responsable de presentar los datos al usuario de una forma que este pueda entenderlos, la vista no debe trabajar con las peticiones esto debe dejárselo al controlador.
- Controlador: es el responsable del comportamiento y procesamiento de las peticiones hechas por el usuario (como una petición a la base de datos).

Spring brinda un MVC para web bastante flexible y configurable, pero esto no le quita sencillez ya que se pueden realizar aplicaciones sencillas sin tener que configurar muchas opciones.

El MVC Web de Spring tiene algunas características que lo hacen único:

- Clara división entre controladores, modelos web y vistas.
- Está basado en interfaces y es bastante flexible.
- Provee “interceptors al igual que controladores.
- No obliga a usar JSP como única tecnología para la parte de la vista.
- Los controladores son configurados como los demás objetos, a través de “IoC”.

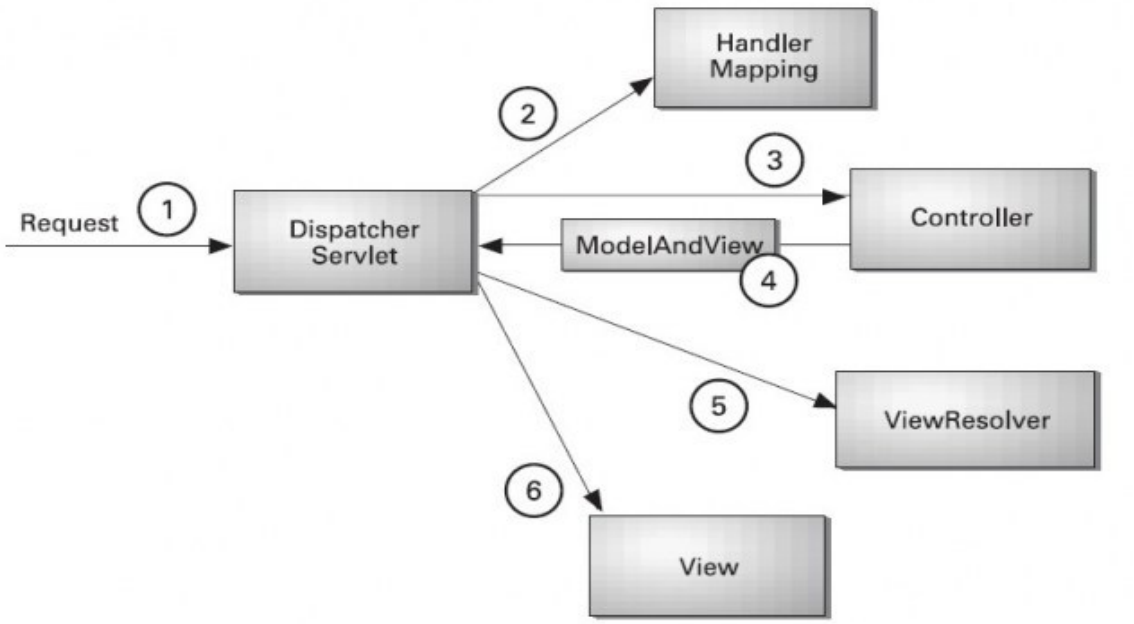

Figura 44. Petición de MVC

El navegador envía una petición y el distribuidor de servlets se encarga de recoger esta petición (paso 1) para pasárselo al controlador de mapeos (paso 2) que comprueba que dicha petición este mapeada y le devuelve el controlador asociado a dicha petición al distribuidor de servlets. Una vez que sabemos que controlador necesitamos el distribuidor de servlets le pasara el control a dicho controlador (paso 3) para que este se encargue de realizar toda la lógica de negocio de nuestra aplicación, este controlador devolverá un objeto modelo y vista (paso 4), el modelo son la información que deseamos mostrar y la vista donde deseamos mostrar dicha información.

Una vez el distribuidor de servlets tiene el objeto modelo y vista tendrá que asociar el nombre de la vista retornado por el controlador con una vista concreta es decir una página jsp, jsf,.. (Paso 5). Una vez resulto esto nuestro distribuidor de servlets tendrá que pasar a la vista el modelo, es decir los datos a presentar, y mostrar la vista (paso 6).

2.1 Distribuidor de servlets (Dispatcher Servlet).

Para configurar como Servlet central se tiene que hacer en el archivo de configuración web.xml (Deployment Descriptor):

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

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

Listado 23. Configuración del distribuidor de servlets.

El distribuidor de servlets buscará como está indicado por el <servlet- name> el contexto de aplicación (application Context) correspondiente con el nombre que se haya puesto dentro <servlet- name> acompañado de la terminación –servlet.xml, en este caso buscará el archivo ejemplo-servlet.xml. En donde se pondrán las definiciones y los diferentes beans, con sus correspondientes propiedades y atributos, para el caso del Web MVC, se pondrán los diferentes revolvedores de vistas a utilizar, los controladores y sus propiedades, el controlador de mapeo, así como los diferentes beans que sean necesarios.

2.2 Controlador de mapeo (HandlerMapping)

Existen diversas maneras de que distribuidor de servlet sepa que controlador debe manejar cada petición y a que bean del contexto de la aplicación se debe asignar. Esta función la realiza controlador de mapeo, existen dos tipos diferentes:

- BeanNameUrlHandlerMapping: mapea el URL hacia el controlador en base al nombre del bean el controlador.

```
<bean id="beanNameUrlMapping" class="org.springframework.web.
servlet.handler.BeanNameUrlHandlerMapping"/>
```

Listado 24. Configuración del BeanNameUrlHandlerMapping.

A continuación se escribe cada uno de los diferentes URL que se vayan a utilizar en la aplicación, poniéndolo en el atributo name del bean y en el atributo class, la clase del Controlador que vaya a procesar dicha petición, y finalmente se ponen las diferentes propiedades que utilizará dicho controlador.

```
<bean name="/ejemplo.htm" class="web.ExampleController">
<property name="Servicio">
<ref bean="Servicio"/>
</property>
</bean>
```

Listado 25. Configuración de las URL del BeanNameUrlHandlerMapping.

Así cuando el usuario entre a una página con el URL /ejemplo.htm la petición que haga será dirigido hacia el controlador ExampleController.

- SimpleUrlHandlerMapping: mapea el URL hacia el controlador basando en una colección de propiedades declarada en el contexto de la aplicación.

```
<bean id="simpleUrlMapping" class= "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
<property name="mappings">
<props>
<prop key="/ejemplo.htm">ExampleController</prop>
```

```
<prop key="/login.htm">LoginController</prop>

</props>

</property>

</bean>
```

Listado 26. Configuración del SimpleUrlHandlerMapping.

En este controlador de mapeo se declara una lista de propiedades en las cuales se pondrá cada una de las URL como una propiedad, con la URL como atributo clave y como valor el nombre del bean del controlador que será responsable de procesar la petición. Por ejemplo la URL /login.htm será responsabilidad del controlador con el nombre del bean LoginController.

2.3 Resolvedor de vistas (View Resolvers).

Una vista es un bean que transforma los resultados para que sean visibles por el usuario, el bean tiene que implementar la interfaz "org.springframework.web.servlet.ViewResolver", la vista es la encargada de resolver el nombre lógico que devuelve un controlador en un archivo físico que el navegador podrá utilizar para presentar los resultados.

Spring cuenta con 4 resolvedores de vistas:

- InternalResourceViewResolver: resuelve los nombres lógicos en un archivo tipo vista que es convertido utilizando una plantilla JSP, JSTL o Velocity.
- BeanNameViewResolver: resuelve los nombres lógicos de las vistas en beans de tipo vista en el contexto de la aplicación del distribuidor de servlets.
- ResourceBundleViewResolver: resuelve los nombres lógicos de las vistas en objetos tipo vista contenidos en el ResourceBundler o en un archivo con extensión .properties.
- XMLViewResolver: resuelve los nombres lógicos de las vistas que se encuentran en un archivo XML separado.

El más utilizado de estos 4 beans es el InternalResourceViewResolver y se declara en el contexto de la aplicación web (web AapplicationContext), su configuración es la siguiente.

```
<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>
```

Listado 27. Configuración del InternalResourceViewResolver.

En la etiqueta id se debe especificar el tipo de resolvedor de vista, de los 4 mencionados anteriormente, se quiere utilizar. Después de esto vienen 3 propiedades:

- ViewClass: especifica el tipo de plantilla que se desea utilizar para desplegar la vista, en este caso JSTL.

- Prefix: es el prefijo al nombre lógico de la vista, comúnmente es la ruta donde se encuentran los JSP.

- Suffix: son los sufijos o extensiones que tendrán nuestras vistas en este caso .jsp

Gracias a esta configuración si se quisiera cambiar el directorio de las vistas solamente cambiaríamos la propiedad Prefix.

2.4 Controlador (Controller).

Existen controladores específicos para cada caso y los más comunes que son para uso general. Para crear un controlador basta con implementar la interfaz de este y sobrescribir los métodos que sean necesarios. Existen varios tipos de controladores en Spring, por lo que se brinda la posibilidad de escoger el controlador a implementar.

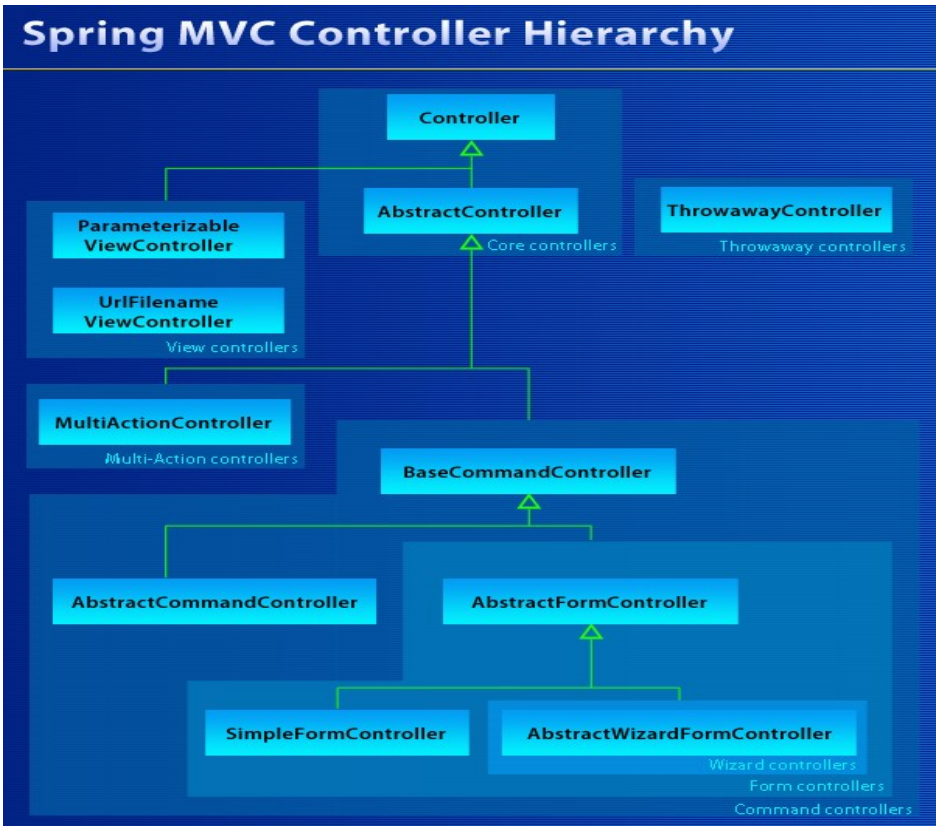

Figura 45. Controladores

La manera en que trabaja cada uno de dichos controladores es similar, cada uno tiene su método específico para procesar las peticiones que haga el usuario, pero todos regresan un objeto tipo "*ModelAndView*" y está compuesto de 2 o más atributos. El principal es el nombre de la vista a la que se va a devolver el modelo para que sea desplegado, y los demás atributos pueden ser parámetros que se le agregan por medio del método. AddObject ("nombre_parámetro", valor_parámetro). Además el Modelo puede estar formado por un solo objeto o por un mapa de Objetos, los cuales se identificarán en la vista con el mismo nombre que se haya mandado dentro del Modelo que se le dio al objeto "*ModelAndView*" que haya devuelto para ser procesado.

```
public class MoviesController extends AbstractController {

//Método que controla el Request

public ModelAndView handleRequestInternal( HttpServletRequest request, HttpServletResponse response)
throws Exception {
```

```
//Se recupera una lista de todas las peliculas
List movies = MovieService.getAllMovies();
//Se manda a la vista movieList el objeto movies, con el nombre lógico //movies
return new ModelAndView("movieList", "movies", movies);
}
}
```

Listado 28. Configuración del controlador.

Capitulo 7

1 Introducción

En este capítulo vamos a ver y explicar que es JPA así como que es un ORM y la manera de utilizar ambos conjuntamente para acceder más fácilmente a las bases de datos.

2 JPA

Java Persistence API, más conocida por su sigla JPA, es la API de persistencia desarrollada para la plataforma Java EE e incluida en el estándar EJB3. Esta API busca unificar la manera en que funcionan las utilidades que proveen un mapeo objeto-relacional. El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos, como sí pasaba con EJB2, y permitir usar objetos regulares (conocidos como POJOs).

3 ORM

[1-*Wikypedia]*

El mapeo objeto-relacional (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilizando un motor de persistencia. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo). Hay paquetes comerciales y de uso libre disponibles que desarrollan el mapeo relacional de objetos, aunque algunos programadores prefieren crear sus propias herramientas ORM.

Es decir para aislar la aplicación del tipo de base de datos que vamos a utilizar vamos a crear una “capa” o programa que es la que se va a encargar de comunicarse con la base de datos y transformaros los datos para que podemos tratar con ellos haciendo que cualquier cambio en la base de datos o incluso un cambio de motor de base de datos no afecte a las otras capas de nuestra aplicación solamente tendremos que cambiar de ORM y listo, en este caso vamos a hablar de Hibernate que es el más popular para Java.

Figura 46. Modelo ORM

4 Hibernate

¿Qué es Hibernate? Hibernate es una librería ORM open-source, que facilita trabajar con bases de datos relacionales. Hace que el desarrollador trabaje con objetos (POJO's), lo que lleva a que el desarrollador se preocupe por el negocio de la aplicación y no del cómo se guardara, recuperará o eliminará la información de la base de datos.

Hibernate funciona asociando a cada tabla de la base de datos un "Plain Old Java Object" (POJO, a veces llamado Plain Ordinary Java Object). Un POJO es similar a una Java Bean, con propiedades accesibles mediante métodos setter y getter, como por ejemplo:

Veamos como configurar hibernate con Spring:

Spring nos provee la clase HibernateDaoSupport para brindarle a nuestros DAO soporte para Hibernate.

HibernateTemplate

En particular, uno de los métodos más útiles que provee es getHibernateTemplate(). Este método devuelve un template con varios métodos útiles, que simplifican el uso de Hibernate. Estos métodos suelen encapsular varias excepciones propias de acceso a datos de Hibernate (y SQL) dentro de una DataAccessException (que hereda de RuntimeException).

Creando un DAO con soporte de Spring

La forma más simple de usar Hibernate con Spring es crear clases que hereden de HibernateDaoSupport. Esta clase ofrece varias utilidades para manipular la sesión de Hibernate, y se encarga de manejar automáticamente las transacciones.

La interfaz del DAO

La interfaz de nuestro DAO es simple, y no necesita tener ninguna herencia ni restricción en particular:

```
public interface InvasorDao {
   guardarInvasor(Invasor invasor);
}
```

Listado 29. Interfaz DAO

La implementación del DAO

La implementación del DAO deberá extender HibernateDaoSupport. Esta clase nos proveerá de varios métodos útiles para manipular la sesión de Hibernate.

```
public class InvasorDaoImpl extends HibernateDaoSupport implements InvasorDao  {
   public void guardarInvasor(Invasor invasor) {
      getHibernateTemplate().save(invasor);
   }
}
```

Listado 30. Implementación DAO.

La declaración del DAO

En el XML de Spring deberemos declarar el DAO y la Factory de Hibernate. Al DAO se le asocia el factory de Hibernate del cual Spring sacará la session de Hibernate.

```
<bean id="dao.InvasorDao" class="com.dosideas.dao.impl.InvasorDaoImpl">
```

```
    <property name="sessionFactory" ref="defaultSessionFactory"/>
  </bean>

  <bean id="defaultSessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="defaultDataSource" />
    <property name="mappingResources">
      <list>
        <!-- Agregar los mappings necesarios -->
          <value>com/dosideas/domain/map/Invasor.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
      </props>
    </property>
  </bean>
```

Listado 31. Declaración del DAO

Uso de la sesión directamente

Es posible utilizar la sesión de Hibernate directamente, a través del método getSession () que provee HibernateDaoSupport. Cuando se pide la sesión de esta manera, es fundamental

```
Session session = null;
try {
  session = getSession();
  //hacer cosas...
}
finally {
  releaseSession(session);
}
```

Listado 32. Uso de la sesión del DAO.

Configuración con anotaciones

Hibernate puede configurarse con anotaciones en vez de archivos XML para definir los mapeos. Para esto usamos la clase de Spring AnnotationSessionFactoryBean.

```
<beans>

 <bean id="defaultSessionFactory"
  class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
  <property name="dataSource" ref="defaultDataSource" />
  <property name="annotatedClasses">
   <list>
    <!-- Agregar los mappings necesarios -->
    <value>com.dosideas.domain.MiClaseDeDominio</value>
   </list>

  </property>
  <property name="hibernateProperties">
   <props>
       <prop key="hibernate.show_sql">true</prop>
   </props>
  </property>
 </bean>

</beans>
```

Listado 33. Configuración con anotaciones.

También puede usarse un escaneo automático de paquetes, de esta manera de no tendríamos que agregar manualmente cada clase anotada. Para hacer esto, es necesario configurar el atributo "packagesToScan" de la clase "AnnotationSessionFactoryBean", indicando allí el paquete donde buscar. Por ejemplo:

```
<beans>

 <bean id="defaultSessionFactory"
  class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
  <property name="dataSource" ref="defaultDataSource" />
  <property name="packagesToScan" value="com.dosideas.domain"/>
 </bean>

</beans>
```

Listado 34. Indicación del paquete a buscar.

Y la clase de dominio anotada como corresponde:

```
package com.dosideas.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "TABLA_DE_ORIGEN")
public class MiClaseDeDominio {
   @Id
   @Column(name = "COLUMNA_PK")
   private String clase;

   @Column(name = "NOMBRE")
   private String nombre;

   //getters y setters...
}
```

Listado 35. Clase con anotaciones.

Capitulo 8 Conclusiones y Trabajo Futuro

1 Introducción

Finalmente en este ultimo capitulo vamos a proponer una serie de trabajos para continuar en el futuro, son funciones que no se han podido implementar en este primer proyecto o mejoras que se proponen y no se han implementado por diversas razones.

2 Conclusiones

Spring es un framework muy potente y completo, gracias a su fácil configuración nos permite crear diferentes aplicaciones adaptadas a nuestras necesidades. Además gracias a sus diferentes módulos nos permite introducir características adicionales como la seguridad, que nos permite configurar todas las opciones y permisos fácilmente además de poder gestionarlos fácilmente sin tener que crearnos objetos adicionales ya que spring security los crea por nosotros.

En cuanto a JPA e hibernate, nos hace más fácil el manejo de los datos al poder tratarlos como entidades, además al ser JPA un estándar y no estar ligado a ningún ORM nos permite poder cambiar de ORM sin tener que realizar grandes cambios en las aplicaciones.

La combinación de JPA, Spring e hibernate nos permite separar más fácilmente la aplicación en capas y realizar cambios en la aplicación que de otra manera requerirían volver a diseñar la aplicación desde el principio, como por ejemplo cambiar de motor de bases de datos.

3 Trabajo Futuro

Como ampliación y mejora para este portal se podrían incluir ciertas funcionalidades como:

- Añadir la funcionalidad de administración de usuarios como dar de alta un usuario, modificar sus privilegios, desactivar un usuario, etc.…
- Añadir la opción de crear páginas web mediante la inclusión de un editor web en el portal.
- Crear nuevos roles para diferentes recursos.
- Añadir funciones específicas para cada tipo de usuario o grupos de usuarios.
- Personalizar las tablas de autentificación para adaptarlas a nuevos requerimientos.
- Crear una versión para Android aprovechando la versión que nos brinda Spring.

Anexos

1 Introducción

En este anexo se proporcionan los manuales de las aplicaciones realizadas.

2 Como crear un proyecto MVC con SpringToolSuite

Lo primero que debemos hacer es crear un proyecto tipo Spring Template Project y dentro de este tipo de proyecto Spring MVC Project.

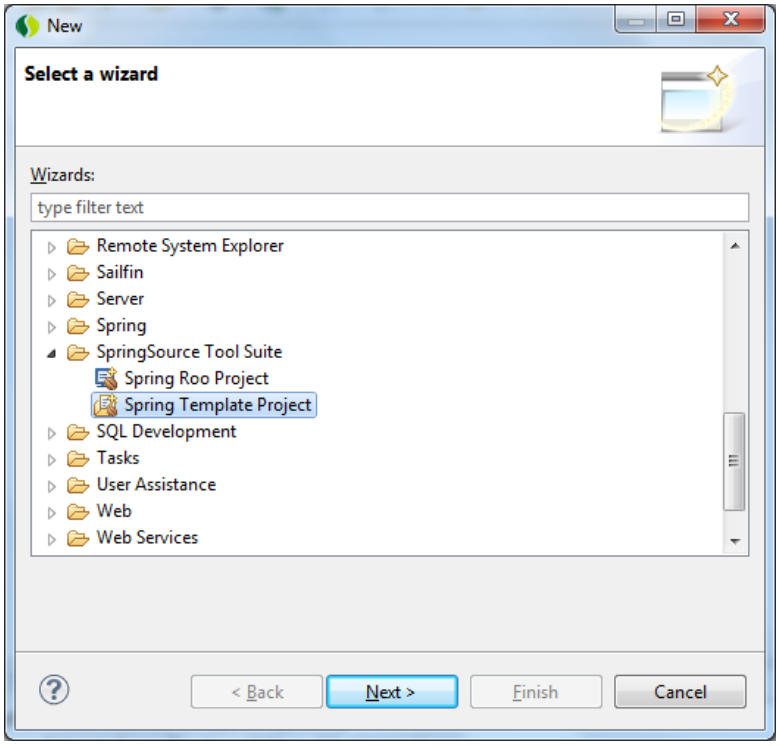

Figura 47. Selección del asistente para crear una aplicación Spring.

File>>New>>Spring Template Project>>MVCProject

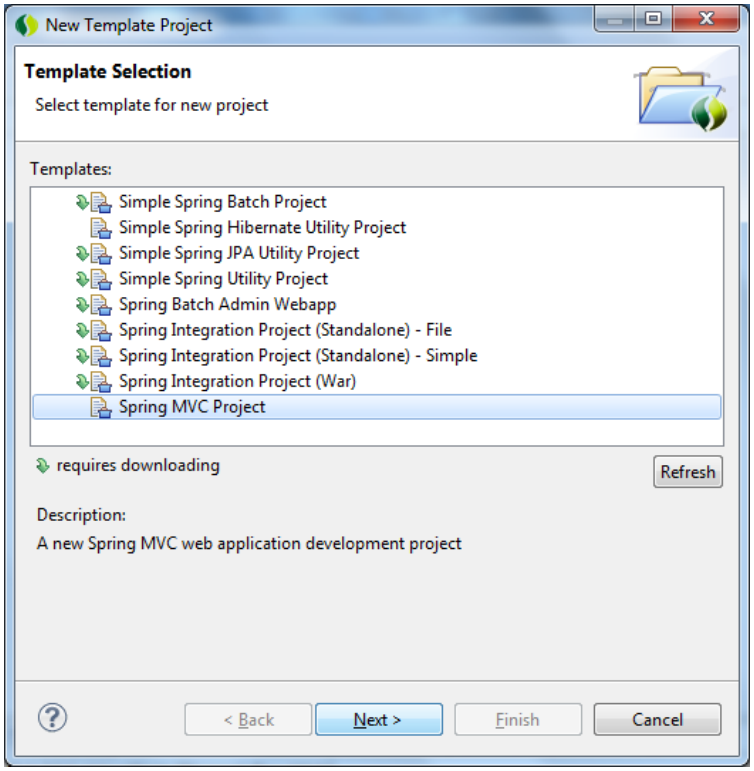

Figura 48. Selección del tipo de proyecto.

Al elegir cualquier tipo de proyecto de la lista si es la primera vez que lo utilizamos, nos pedirá permisos para descargar todas las dependencias necesarias automáticamente, ahora le damos a Next y nos pedirá un nombre de proyecto y un nombre de paquete que debe ir separado por 2 puntos.

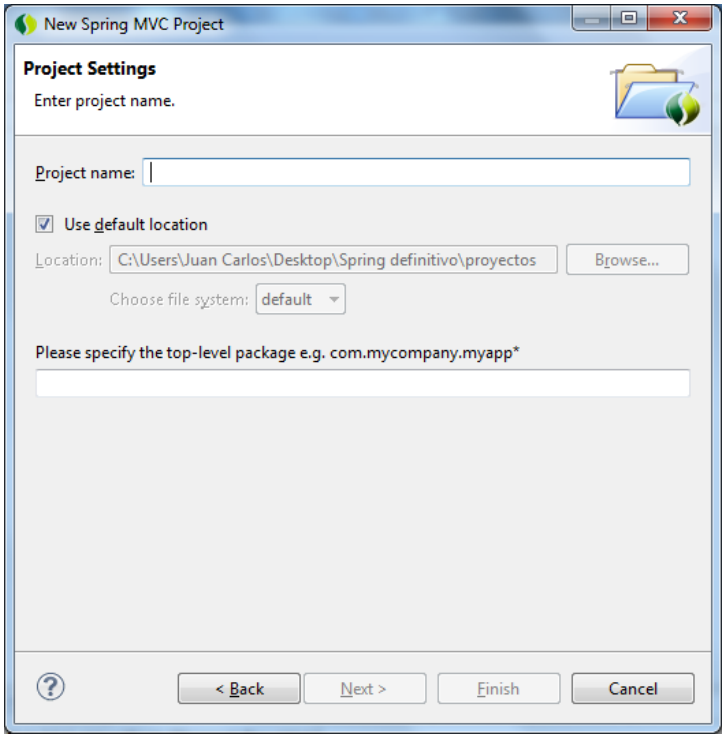

Figura 49. Nombre del proyecto.

Hemos nombrado al proyecto portalDocenteSpring y al paquete le llamaremos upv.pfc.pd, introducimos los datos y finalizamos para que nos cree el proyecto.

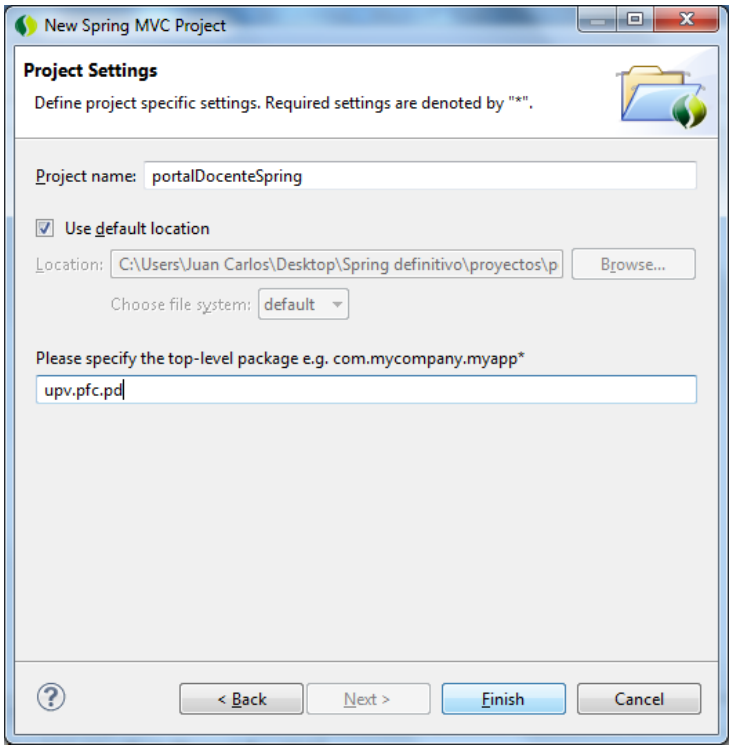

Figura 50. Nombre del paquete.

Una vez creado el proyecto vamos a ver la estructura del proyecto.

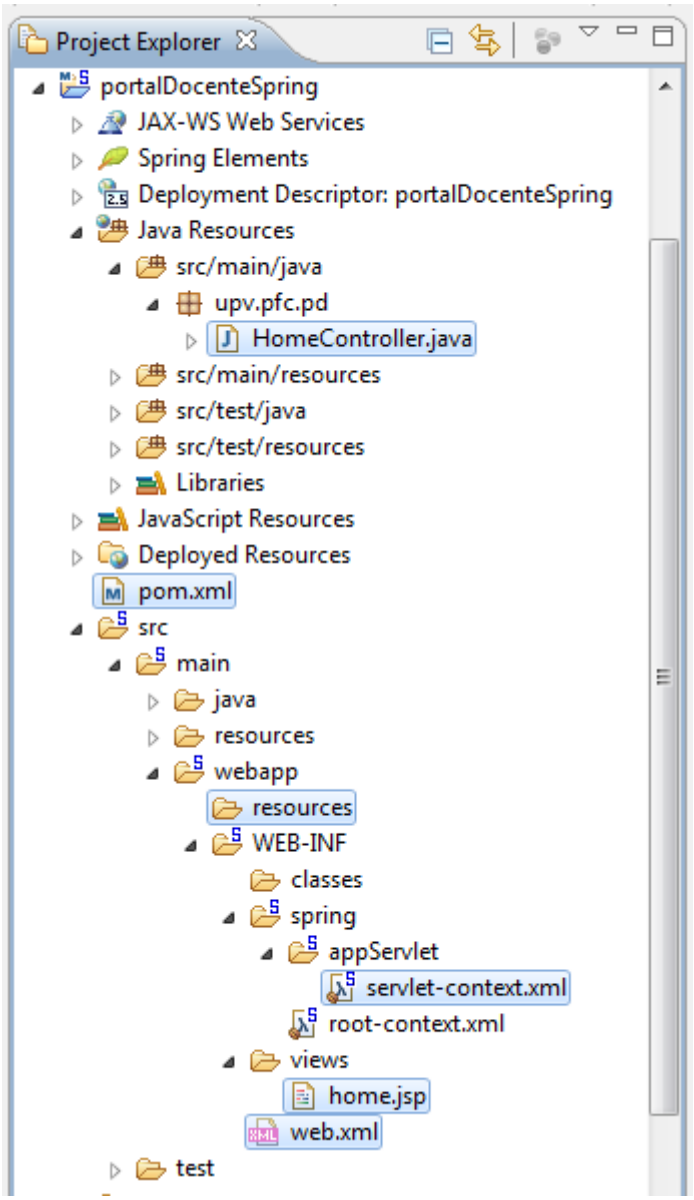

Figura 51. Estructura del proyecto.

El proyecto esta creado con Maven, con lo que nos ahorramos el trabajo de crear nosotros la estructura del proyecto, ahora vamos a explicar donde se ubican los archivos más importantes.

El archivo "pom.xml" es el archivo que utiliza Maven para la configuración del proyecto.

Figura 52. Archivo pom.xml

Como podemos ver en el archivo ahí una etiquetas llamadas dependencias, estas son muy importantes porque son la librerías que vamos a utilizar en el proyecto , cada vez que queramos incluir una librería nueva en el proyecto vamos a añadir una etiqueta de dependencia en el archivo "pom.xml" y Maven automáticamente se bajara la librería correspondiente y la incluirá en el proyecto, es importante hacerlo de esta manera porque si incluimos la librería en el proyecto de otra forma al compilar Maven no encontrara la dependencia y el proyecto no funcionara correctamente .

La etiqueta tiene esta forma:

```
<dependency>
        <groupId> identifica normalmente a un proyecto </groupId>
        <artifactId>  nombre concreto del jar  </artifactId>
        <version> número de versión que queremos  </version>
</dependency>
```

Listado 36. Etiqueta de Maven.

Ahora vamos a ver el archivo "web.xml" que es el que tiene nuestra configuración.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <!-- The definition of the Root Spring Container shared by all Servlets and Filters -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/root-context.xml</param-value>
    </context-param>

    <!-- Creates the Spring Container shared by all Servlets and Filters -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- Processes application requests -->
    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

Figura 53. Archivo web.xml

En este archivo vemos varias etiquetas, vamos a explicar para que sirve cada una:

```xml
<context-param>
	<param-name>contextConfigLocation</param-name>
	<param-value>/WEB-INF/spring/root-context.xml</param-value>
</context-param>
```

Listado 37. Etiqueta archivo de contexto.

Esta etiqueta nos indica donde están nuestros archivos de configuración para la aplicación.

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

Listado 38. Declaración del listener.

Aquí se declara un listener que es el que cargara el contexto de la aplicación.

```xml
	<servlet>
		<servlet-name>appServlet</servlet-name>
		<servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
		<init-param>
			<param-name>contextConfigLocation</param-name>
			<param-value>/WEB-INF/spring/appServlet/servlet-
context.xml</param-value>
		</init-param>
		<load-on-startup>1</load-on-startup>
```

```
</servlet>
```

Listado 39. Declaración del DispatcherServlet.

En esta etiqueta se declara el controlador y se le indica que será el primer o en ser cargado al inicializar la aplicación.

```
<servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>/</url-pattern>
</servlet-mapping>
```

Listado 40. Configuración de las peticiones que aceptaremos.

Por último en esta etiqueta se indica que clase de peticiones va a aceptar, en este caso le estamos diciendo que acepte cualquier petición que le entre.

Ahora vamos a explicar el archivo de configuración "Servlet-context.xml".

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <!-- DispatcherServlet Context: defines this servlet's request-processing infrastructure -->

    <!-- Enables the Spring MVC @Controller programming model -->
    <annotation-driven />

    <!-- Handles HTTP GET requests for /resources/** by efficiently serving up static resources in the ${webappRoot}/resources directory -->
    <resources mapping="/resources/**" location="/resources/" />

    <!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->
    <beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".jsp" />
    </beans:bean>

    <context:component-scan base-package="upv.pfc.pd" />


</beans:beans>
```

Figura 54. Archivo Servlet-context.xml.

Vamos a explicar en primer lugar la etiqueta resources:

```
<resources mapping="/resources/**" location="/resources/" />
```

Listado 41. Etiqueta resources.

Esta etiqueta es la que define donde estarán ubicados nuestros recursos, como hemos mencionado anteriormente los vamos a poner en la carpeta resources, al poner esta definición estamos diciendo que la aplicación tiene permiso para acceder a los archivos que están dentro. Con la opción mapping="/resources/**" le estamos diciendo que nos de acceso a todos los recursos dentro de esta carpeta y con location="/resources/" le estamos diciendo donde se encuentra la carpeta.

```
<beans: bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
            <beans: property name="prefix" value="/WEB-INF/views/" />
```

```
            <beans: property name="suffix" value=".jsp" />
</beans: bean>
```

Listado 42. Configuración de prefijo y sufijo.

Aquí le estamos indicando a la aplicación donde están todas las vistas que terminan en .jsp (archivos jsp) esto es importante porque nos ahorramos tener que poner la ruta completa en el controlador.

Ahora nos queda la última etiqueta:

```
<context:component-scan base-package="upv.pfc.pd" />
```

Listado 43. Configuración del escaneo de paquetes

Esta etiqueta le dice a la aplicación que busque en al paquete upv.pfc.pd los controladores para las páginas, esto nos ahorra tener que declararlos cada vez y poder definir los controladores más fácilmente mediante anotaciones.

Por último vamos ver el controlador, como hemos definido en el "Servlet.Context.xml" los controladores están ubicados en JavaResources>>src/main/java>>upv.pfc.pd y dentro de este paquete nos encontramos en "HomeController", vamos a ver lo básico de un controlador.

```
package upv.pfc.pd;

import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 * Handles requests for the application home page.
 */
@Controller
public class HomeController {

    private static final Logger logger = LoggerFactory.getLogger(HomeController.class);

    /**
     * Simply selects the home view to render by returning its name.
     */
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String home(Locale locale, Model model) {
        logger.info("Welcome home! the client locale is "+ locale.toString());

        Date date = new Date();
        DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.LONG, locale);

        String formattedDate = dateFormat.format(date);

        model.addAttribute("serverTime", formattedDate );

        return "home";
    }
```

Figura 55. HomeController.

Lo más importante es la anotación @Controller, esto indica que es un controlador y cuando la aplicación escanee el paquete lo encontrara. Después tenemos la anotación @RequestMapping (value = "/", method = RequestMethod.GET) esta anotación captura la petición que es igual al valor de value y se realiza por el método Get, al ser la petición raíz se está indicando que esta será la página de inicio de la aplicación.

Tenemos una función llamada home que nos devolverá una cadena, esta cadena será el nombre de la página que queremos servir pero sin la extensión ni la ruta ya que la hemos configurado anteriormente en el Servlet-Context, por lo que solamente devolvemos home y nos devolverá el archivo "home.jsp" que está dentro de la carpeta views.

Esta función en concreto se le pasa como argumento un modelo esto es necesario para pasar parámetros, se define una variable con la hora actual y se le añade al modelo con el nombre serverTime, este nombre es por el que accederos a ese parámetro desde la vista, es decir desde el archivo "home.jsp".

Y finalmente el archivo "home.jsp" que no es más que un archivo jsp normal.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
<head>
    <title>Home</title>
</head>
<body>
<h1>
    Hello world!
</h1>

<P>  The time on the server is ${serverTime}. </P>
</body>
</html>
```

Figura 56. Archivo home.jsp

Como vemos estamos accediendo a la variable que tiene la fecha por medio de ${serverTime}.

Ahora queda arrancar el servidor y añadir la aplicación.

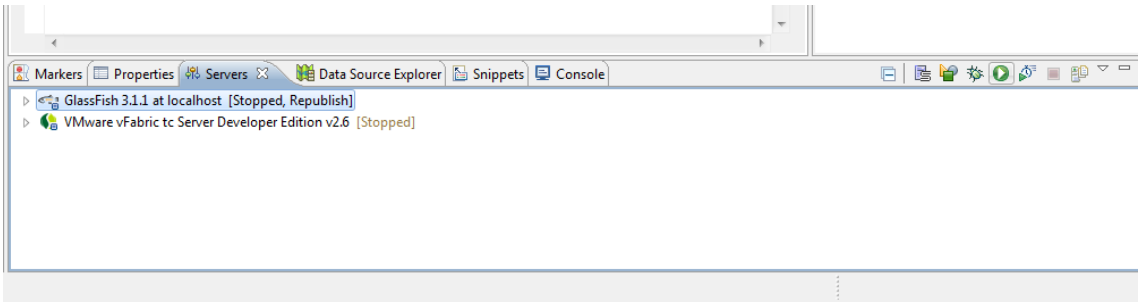

Figura 57. Arranque de la aplicación.

Añadimos la aplicación al servidor y presionamos Finish.

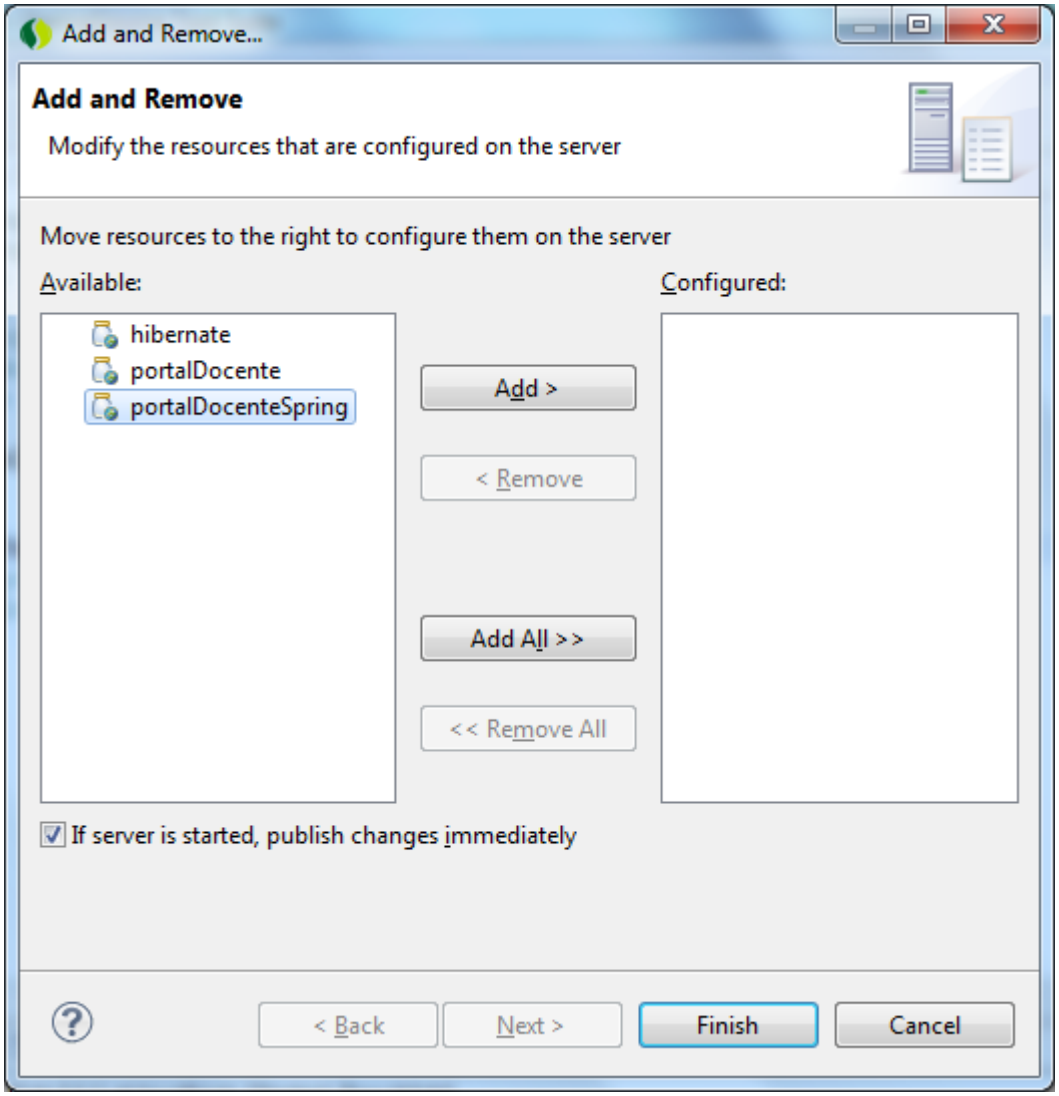

Figura 58. Añadiendo la aplicación al servidor.

Ahora ya podemos acceder a la aplicación por medio de la dirección
http://localhost:8080/portalDocenteSpring

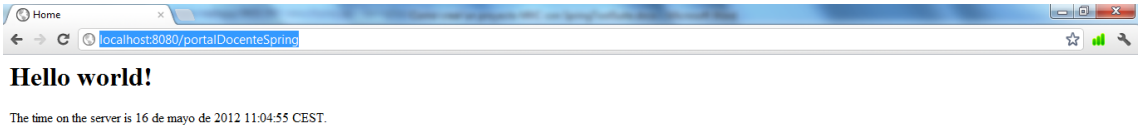

Figura 59. Aplicación en marcha.

3 Añadir autentificación en memoria con Spring security.

Vamos a continuar con el proyecto portalDocenteSpring ahora que ya tenemos creado el proyecto básico vamos a añadirle seguridad para controlar los acceso a sus recursos, para ello es necesario añadirle el modulo Spring security a nuestro proyecto. Como hemos explicado anteriormente la forma más fácil de hacer esto es mediante Maven, es decir incluyendo las librerías en el archivo "pom.xml" como dependencias para que se las descargue y las añada al proyecto. Las dependencias incluir en el archivo son:

```
<!-- Spring security -->
 <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-config</artifactId>
            <version>3.1.0.RELEASE</version>
 </dependency>
 <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-web</artifactId>
            <version>3.1.0.RELEASE</version>
</dependency>
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-taglibs</artifactId>
        <version>3.1.0.RELEASE</version>
</dependency>
```

Listado 44. Dependencias a incluir en el proyecto.

Añadimos las dependencias al proyecto.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${org.springframework-version}</version>
    <exclusions>
        <!-- Exclude Commons Logging in favor of SLF4j -->
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
         </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${org.springframework-version}</version>
</dependency>
        <!-- Spring security -->
 <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-config</artifactId>
        <version>3.1.0.RELEASE</version>
 </dependency>
 <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-web</artifactId>
        <version>3.1.0.RELEASE</version>
</dependency>
<dependency>
      <groupId>org.springframework.security</groupId>
      <artifactId>spring-security-taglibs</artifactId>
      <version>3.1.0.RELEASE</version>
</dependency>


<!-- AspectJ -->
```

Figura 60. Archivo pom.xml con las dependencias.

Y guardamos el proyecto, automáticamente Maven se bajar los archivos y los añadirá a nuestro proyecto.

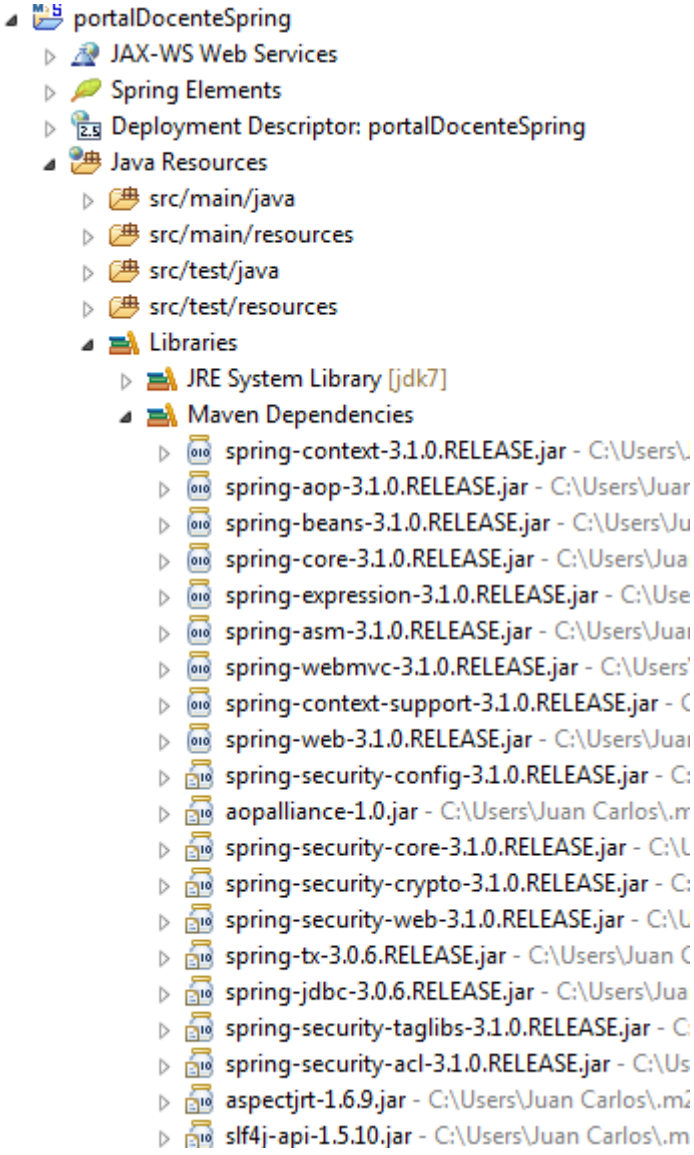

Figura 61. Dependencias ya incluidas.

Ahora ya podemos empezar a crear nuestra configuración para la aplicación. Lo primero que vamos a hacer es crear una página de inicio personalizada y un menú lateral para navegar por la aplicación. Esto no tiene ningún secreto ya que es HTML básico , ponemos nuestras páginas web dentro de la carpeta views, el menú lateral lo vamos a incluir con la función include en el jsp así que no es necesario que lleve las etiquetas <HTML><head> y <body> para que no estén duplicadas en el archivo. También vamos a crearnos en la carpeta resources tres carpetas Img, css y js que corresponderán a las imágenes, estilos y archivos javascript de nuestra página web, y lo hacemos en esa carpeta por que como hemos mencionado antes tenemos configurada la aplicación para que nos deje acceder solamente a esta carpeta.
Al igual que con menú lateral para no tener que picar cada vez el código del menú y de la cabecera , vamos a dividir los archivos en 3 partes cabecera, menú lateral y cuerpo y lo único que cambiara en todos los archivos será el archivo del cuerpo.

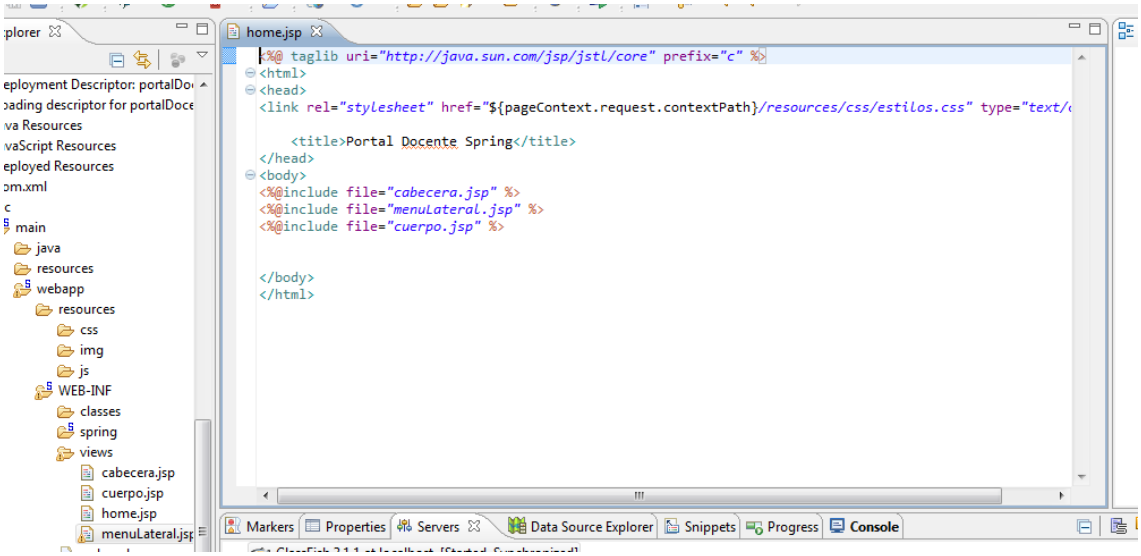

Figura 62. Archivo dividido.

La función ${pageContext.request.contextPath} que vemos en la imagen al incluir los archivos de estilos en necesaria para que nos devuelva la ruta para llegar hasta la carpeta resources, por lo que en toda la inclusión de recursos que se encuentren en dicha carpeta tendremos que utilizar esta función. Ahora publicamos la aplicación en el servidor.

Figura 63. Vista de la aplicación.

Bien ahora que ya hemos personalizado un poco nuestras páginas, podemos crear dentro de nuestra carpeta views otra carpeta llamada documentación donde podemos crear archivos jsp con información sobre temas como por ejemplo Java, MVC, etc..
Ahora vamos a dotar de seguridad el acceso a las páginas que hemos creado, para empezar tenemos que añadir dos filtros a nuestro archivo "web.xml" que son estos dos:

```
 <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-
class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
      </filter>
```

```
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Listado 45. Filtros a añadir.

Con lo que nuestro archivo "web.xml" quedara de la siguiente forma.

```
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app

   <!-- The definition of the Root Spring Container shared by all Servlets and Filters -->
   <context-param>
       <param-name>contextConfigLocation</param-name>
       <param-value>/WEB-INF/spring/root-context.xml</param-value>
   </context-param>

   <!-- Creates the Spring Container shared by all Servlets and Filters -->
   <listener>
       <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
   </listener>

   <!-- Processes application requests -->
   <servlet>
       <servlet-name>appServlet</servlet-name>
       <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
       <init-param>
           <param-name>contextConfigLocation</param-name>
           <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
       </init-param>
       <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
       <servlet-name>appServlet</servlet-name>
       <url-pattern>/</url-pattern>
   </servlet-mapping>
    <filter>
     <filter-name>springSecurityFilterChain</filter-name>
     <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
   </filter>
   <filter-mapping>
     <filter-name>springSecurityFilterChain</filter-name>
     <url-pattern>/*</url-pattern>
   </filter-mapping>
/web-app>
```

Figura 64. Archivo web.xml con seguridad.

Ahora debemos añadir todo lo referente a la configuración de los filtros, para esto vamos a crearnos un nuevo archivo llamado "seguridad.xml" en la carpeta spring, debemos incluir su ubicación en el archivo "web.xml".

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
    /WEB-INF/spring/root-context.xml
    /WEB-INF/spring/seguridad.xml
    </param-value>
</context-param>
```

Figura 65. Añadiendo el archivo seguridad.xml.

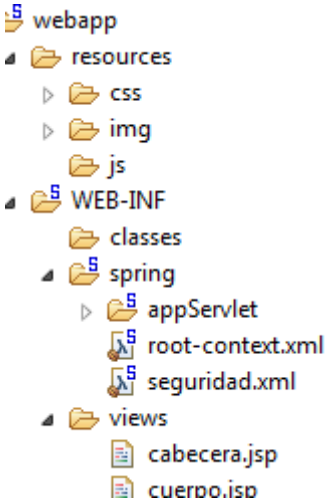

Figura 66. Vista del directorio con el archivo seguridad.xml.

Ahora puede que el servidor nos esté dando un fallo, esto es porque el archivo "seguridad.xml" aun no tiene definido los DTD y los esquemas propios de un archivo de configuración de Spring, debemos abrir el archivo e incluir en el este código.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
           http://www.springframework.org/schema/security
           http://www.springframework.org/schema/security/spring-security-
3.1.xsd">


           </beans:beans>
```

Listado 46. DTD de Spring.

Ahora que ya tenemos definido el schema del archivo vamos a incluir la configuración de seguridad para esto debemos incluir los siguientes tags.

```
<http auto-config="true" use-expressions="true">
  <intercept-url pattern="/**" access="hasRole('alumno')" />
  </http>
```

Listado 47. Tag del rol alumno.

Este tag especifica las URL que se van a filtrar vamos a ver las distintas propiedades:

- Auto-config="true" esto nos configura automáticamente la pagina de login y el logout.
- Use-expressions= esto hablita el uso de las funciones como hasRole () que devuelven true o false.
- Pattern= especifica el recurso al que se quiere mapear.
- Acces= especifica los roles que tendrán permisos

Ahora vamos a añadir algunos usuarios para entrar a nuestra aplicación.

```
      <authentication-manager>
        <authentication-provider>
            <user-service>
```

```
            <user name="alumn" password="alumno" authorities="alumno" />
        </user-service>
    </authentication-provider>
</authentication-manager>
```

Listado 48. Añadiendo un usuario a nuestra aplicación.

Ahora hemos creado un usuario alumn con el password alumno y con el rol de alumno para poder entrar en la aplicación, ahora publicamos la aplicación y nos denegara el acceso mostrándonos la pagina de login que tiene spring security por defecto.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
          http://www.springframework.org/schema/security
          http://www.springframework.org/schema/security/spring-security-3.1.xsd">

        <http auto-config="true" use-expressions="true">
  <intercept-url pattern="/**" access="hasRole('alumno')" />
  </http>

    <authentication-manager>
      <authentication-provider>
          <user-service>
            <user name="alumn" password="alumno" authorities="alumno" />
          </user-service>
      </authentication-provider>
  </authentication-manager>

</beans:beans>
```

Figura 67. Configuración de seguridad.xml.

Esta es una configuración muy básica y muy restrictiva como mas ejemplo vamos a añadir un rol más el de profesor y vamos a hacer que el acceso a la página de inicio sea anónimo, pero que el acceso a los demás recursos necesiten autentificación, al tener más de dos roles podemos hacer que dependiendo del rol de nuestro usuario aparezca un menú o otro.

El archivo "seguridad.xml" debe quedar así:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
          http://www.springframework.org/schema/security
          http://www.springframework.org/schema/security/spring-security-3.1.xsd">

  <!-- Configuración de seguridad para las URL's -->

  <http pattern="/" security="none"/>
  <http auto-config="true" use-expressions="true">
        <intercept-url pattern="/alumno/**" access="hasRole('alumno')"/>
        <intercept-url pattern="/profesor/**" access="hasRole('profesor')"/>
        <intercept-url pattern="/tutorial/**" access="isAuthenticated()"/>
        <intercept-url pattern="/documentacion/**" access="isAuthenticated()" />
        <intercept-url pattern="/**" access="permitAll" />
        <form-login login-page="/home" />
        <logout logout-success-url="/" delete-cookies="JSESSIONID"/>
  </http>

      <!-- Usuarios y contraseñas  -->
      <authentication-manager>
        <authentication-provider>
            <user-service>
               <user name="alumn" password="alumno" authorities="alumno" />
               <user name="profe" password="profesor" authorities="profesor" />
            </user-service>
        </authentication-provider>
    </authentication-manager>




</beans:beans>
```

Figura 68. Configuración completa de seguridad.xml.

Veras que hemos creado más secciones y además hemos definido el home como nuestra página de login y hemos configurado el logout para que podemos cerrar la sesión y nos envié a la página de inicio.

Como es de suponer hemos creado en la carpeta views las carpetas alumno, profesor, documentación y tutorial y en ellas pondremos las paginas correspondientes a cada una de ellas.

Ahora vamos a explicar cómo se mostraran los diferentes menús en el menú lateral para ello vamos a utilizar el tag <sec: authorize>.

Por ejemplo.

```xml
        <sec:authorize access="hasRole('alumno')">
    <table>
    <tr>
    <td><b>Menu alumno</b></td>
    </tr>
    <tr>
    <td>
     <ul class="enlaces_lateral">
    <li type="disc"><a href="/portalDocenteSpring/alumno/listar">Listar
archivos</a></li>
    <li type="disc"><a href="/portalDocenteSpring/tutorial/mvcTutorial">Subir
archivo</a></li>
```

```
</ul>
</td>
</tr>
</table>
</sec:authorize>
```

Listado 49. Configuración de permisos.

Esto comprueba si el usuario tiene permisos de alumno si es así devuelve true y si no devuelve false y no pinta el menú.
Aparte de todo esto también debemos hacer los controladores para que nos devuelvan las vistas, como siempre los controladores irán en la carpeta src/main /java y dentro del paquete upv.pfc.pd, ahí vamos a crear un controlador llamado "adminController.java" para todas estas páginas del tutorial.

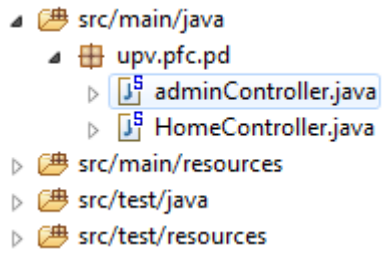

Figura 69. Directorios con controlador adminController.java.

Haremos lo mismo con todas las demás páginas.
Ahora ya podemos publicar la aplicación y veremos que dependiendo del rol con el que entremos tendremos un menú u otro.

Figura 70. Aplicación completa.

4 Autentificación contra base de datos.

Como la autentificación en memoria no recomendable si tenemos muchos usuarios ya que nos complica mucho la gestión, vamos a adaptar la aplicación para que se autentifique contra una base de datos MySql. Spring security dispone de una configuración en la que no hace falta declarar ninguna función especial ni consulta SQL ya que la propia configuración interna de spring security se encarga de hacerlo, para ello spring security propone un tipo de tablas base para trabajar con ellas con lo que la aplicación sabe su estructura y realiza todo el trabajo por nosotros, este es el esquema propuesto por spring security:

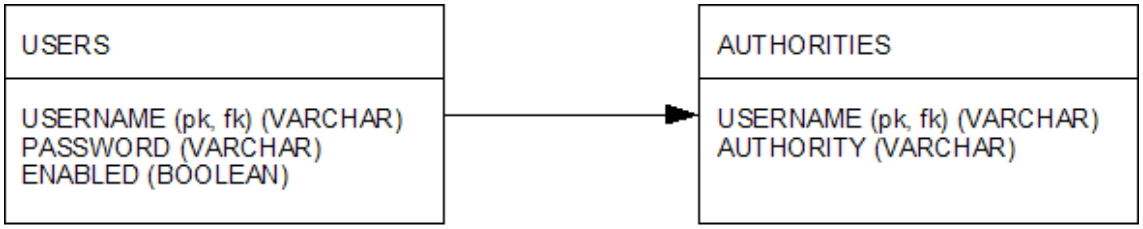

Figura 71. Tablas de Spring security.

```
CREATE DATABASE IF NOT EXISTS PD;
USE PD;
CREATE TABLE IF NOT EXISTS USERS (
  USERNAME varchar(45) NOT NULL,
  PASSWORD varchar(45) DEFAULT NULL,
  ENABLED enum('true','false') DEFAULT 'true',
  PRIMARY KEY (USERNAME)
) ENGINE=InnoDB;
CREATE TABLE IF NOT EXISTS AUTHORITIES (
  USERNAME varchar(45) NOT NULL,
  AUTHORITY varchar(45) DEFAULT NULL,
  PRIMARY KEY (USERNAME),
  FOREIGN KEY (USERNAME) REFERENCES USERS (USERNAME) ON DELETE CASCADE ON
UPDATE CASCADE
) ENGINE=InnoDB;
INSERT INTO USERS (USERNAME, PASSWORD, ENABLED) VALUES ('alumn',
'alumno', 'true');
INSERT INTO USERS (USERNAME, PASSWORD, ENABLED) VALUES ('prof',
'profesor', 'true');
INSERT INTO AUTHORITIES (USERNAME, AUTHORITY) VALUES ('alumn',
'alumno');
INSERT INTO AUTHORITIES (USERNAME, AUTHORITY) VALUES ('prof',
'profesor');
```

Listado 50. Creación de las tablas para Spring security.

Ejecutamos el script en nuestro MySql y ya tenemos creada nuestra base de datos para la autentificación.
Ahora debemos incluir en el proyecto el driver para poder conectarnos a la base de datos, vamos a hacer como hasta ahora le vamos a indicar a Maven la dependencia que tiene que incluir en el proyecto y la misma ara el resto del trabajo.

```
<!-- Mysql -->
            <dependency>
              <groupId>mysql</groupId>
              <artifactId>mysql-connector-java</artifactId>
              <version>5.1.8</version>
          </dependency>
```

Listado 51. Dependencia para MySQL.

Ahora lo incluimos en el archivo "pom.xml" y guardamos el proyecto y comprobamos que se ha incluido en el proyecto.

```
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${org.springframework-version}</version>
        <exclusions>
            <!-- Exclude Commons Logging in favor of SLF4j -->
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
             </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${org.springframework-version}</version>
    </dependency>
                    <!-- Mysql -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.8</version>
    </dependency>
            <!-- Spring security -->
     <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-config</artifactId>
            <version>3.1.0.RELEASE</version>
     </dependency>
     <dependency>
            <groupId>org.springframework.security</groupId>
```

Figura 72. Archivo pom.xml con dependencia para MySQL.

Comprobaos que está en el proyecto.

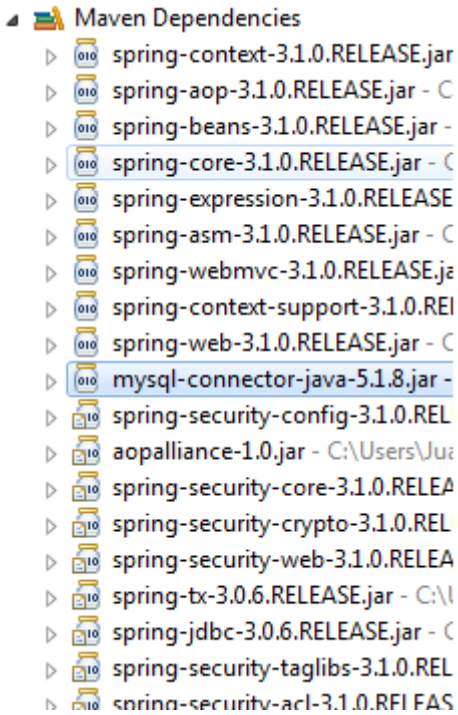

Figura 73. Dependencia de MySQL descargada.

Ahora debemos modificar nuestro archivo "seguridad.xml" para que la autentificación se haga por base de datos, no será necesario modificar el mapeo de la URL y solamente tendremos que eliminar la parte correspondiente a los usuarios y password que ahora se encuentran en la base de datos, después de esto deberemos declarar un dataSource para realizar la conexión a la base de datos.
Primero que todo debemos declarar nuestro AuthenticationManager definiendo el nombre de nuestro dataSource

```
<authentication-manager alias="authenticationManager">
 <authentication-provider>
  <jdbc-user-service data-source-ref="seguridadDataSource" />
  </authentication-provider>
  </authentication-manager>
```

Listado 52. Definición del authenticationManager.

Ahora debemos declarar nuestro bean para nuestro dataSource, indicando la clase que va utilizar.

```
<beans:bean
     id="userService" class="org.springframework.security.core.userdetails.
     jdbc.JdbcDaoImpl">
  <beans:property name="dataSource" ref="seguridadDataSource" />
  </beans:bean>
```

Listado 53. Declaración de clase a utilizar por el dataSource.

Ahora ya solo nos queda declarar nuestro dataSource al que hemos llamado seguridadDataSource con las opciones de configuración correcta, es decir la clase que utilizara, la URL, el driver, el usuario para conectarnos a la base de datos y el password.

```
  <beans:bean
  id="seguridadDataSource" class="org.springframework.jdbc.datasource.D
  riverManagerDataSource">
```

```
<beans:property name="driverClassName" value="com.mysql.jdbc.Driver"
  />
<beans:property name="url" value="jdbc:mysql://localhost:3306/pd" />
<beans: property name="username" value="root" />
<beans: property name="password" value="root" />
</beans:bean>
```

Listado 54. Declaración del dataSource.

Después de todo esto el archivo quedaría así.

```
<!-- Configuración de seguridad para las URL's -->

<http pattern="/" security="none"/>
<http auto-config="true" use-expressions="true">
      <intercept-url pattern="/alumno/**" access="hasRole('alumno')"/>
      <intercept-url pattern="/profesor/**" access="hasRole('profesor')"/>
      <intercept-url pattern="/tutorial/**" access="isAuthenticated()"/>
      <intercept-url pattern="/documentacion/**" access="isAuthenticated()" />
      <intercept-url pattern="/**" access="permitAll" />
      <form-login login-page="/home" />
      <logout logout-success-url="/" delete-cookies="JSESSIONID"/>
</http>
<authentication-manager alias="authenticationManager">
<authentication-provider>
<jdbc-user-service data-source-ref="seguridadDataSource" />
</authentication-provider>
</authentication-manager>
<!--
**** BEAN 'JDBCDAOIMPL' ****

-->
<beans:bean id="userService" class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
<beans:property name="dataSource" ref="seguridadDataSource" />
</beans:bean>
<!--
**** BEAN 'DATASOURCE' ****

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

Figura 74. Archivo seguridad.xml con dataSource.

Ahora publicamos la aplicación y comprobamos que funciona correctamente y ya tenemos nuestra autentificación contra base de datos.

Figura 75. Aplicación funcionando correctamente.

Entramos como alumno.

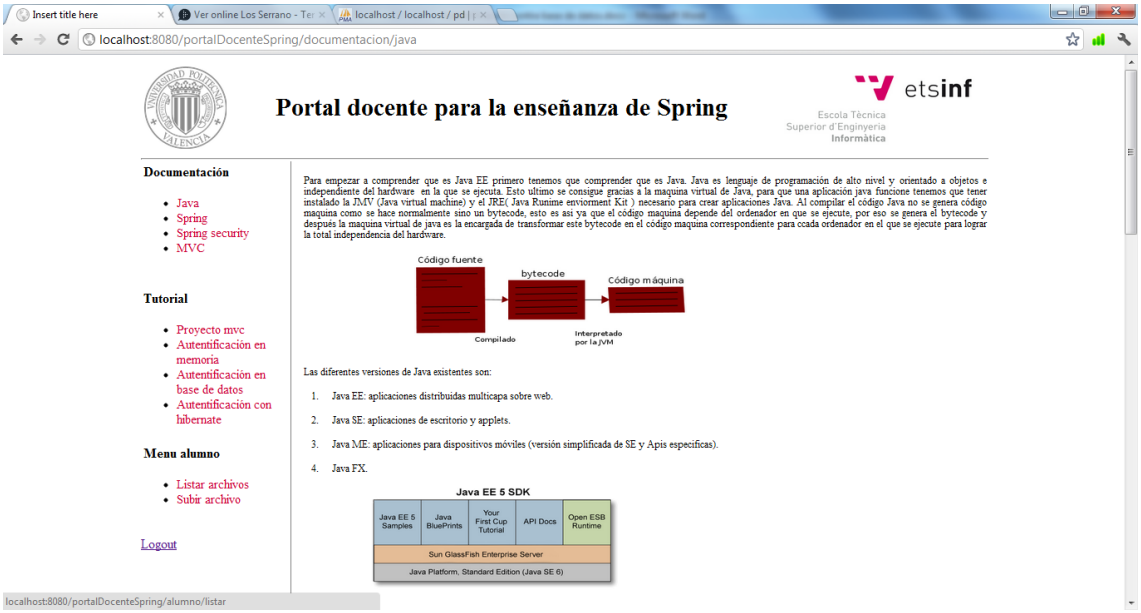

Figura 76. Dentro de la aplicación.

5 Autentificación con base de datos con hibernate.

Ahora que ya tenemos configurada nuestro portal para que trabaje con base de datos vamos aponer otra capa mas en nuestra aplicación configurándola apara usar hibernate anotaciones, esto nos libraría de trabajar con archivos de mapeo y aprovecharnos de las facilidades que tiene implementado el framework Spring para su utilización con ORM en este caso hibernate.

En primer lugar hibernate necesita muchas dependencias y algunas de estas solo trabajan con ciertas versiones de librerías para que funcione correctamente por este motivo vamos a cambiar casi todas las dependencias de las librerías que hay en Maven y las vamos a sustituir por estas versiones que son las que funcionan con hibernate 3.

```
<dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
```

```
            <version>4.8.1</version>
            <type>jar</type>
            <scope>compile</scope>
      </dependency>
      <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <version>3.0.5.RELEASE</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>3.0.5.RELEASE</version>
      <type>jar</type>
      <scope>compile</scope>
      <exclusions>
            <exclusion>
                  <artifactId>commons-logging</artifactId>
                  <groupId>commons-logging</groupId>
            </exclusion>
      </exclusions>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>3.0.5.RELEASE</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>jstl</groupId>
      <artifactId>jstl</artifactId>
      <version>1.1.2</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>taglibs</groupId>
      <artifactId>standard</artifactId>
      <version>1.1.2</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>3.0.5.RELEASE</version>
      <type>jar</type>
```

```
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>3.0.5.RELEASE</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>commons-digester</groupId>
    <artifactId>commons-digester</artifactId>
    <version>2.1</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>commons-collections</groupId>
    <artifactId>commons-collections</artifactId>
    <version>3.2.1</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>3.3.2.GA</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>persistence-api</artifactId>
    <version>1.0</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>c3p0</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.1.2</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>3.0.5.RELEASE</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.6.1</version>
    <type>jar</type>
    <scope>compile</scope>
```

```
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.6.1</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib-nodep</artifactId>
  <version>2.2</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-annotations</artifactId>
  <version>3.4.0.GA</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>jboss</groupId>
  <artifactId>javassist</artifactId>
  <version>3.7.ga</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.14</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
  <!-- Servlet -->
        <dependency>
              <groupId>javax.servlet</groupId>
              <artifactId>servlet-api</artifactId>
              <version>2.5</version>
              <scope>provided</scope>
        </dependency>
        <dependency>
              <groupId>javax.servlet.jsp</groupId>
              <artifactId>jsp-api</artifactId>
              <version>2.1</version>
              <scope>provided</scope>
        </dependency>
        <dependency>
              <groupId>javax.servlet</groupId>
              <artifactId>jstl</artifactId>
              <version>1.2</version>
        </dependency>
                    <!-- Spring security -->
         <dependency>
                    <groupId>org.springframework.security</groupId>
```

```
                <artifactId>spring-security-config</artifactId>
                <version>3.1.0.RELEASE</version>
        </dependency>
        <dependency>
                <groupId>org.springframework.security</groupId>
                <artifactId>spring-security-web</artifactId>
                <version>3.1.0.RELEASE</version>
    </dependency>
    <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-taglibs</artifactId>
            <version>3.1.0.RELEASE</version>
    </dependency>
```

Listado 55. Dependencias para hibernate.

Ahora guardamos el proyecto y Maven ara el trabajo por nosotros. Ahora que ya tenemos todas las librerías en nuestro proyecto vamos a empezar por cambiar todo nuestro archivo "seguridad.xml" por este contenido.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://www.springframework.org/schema/security"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
        http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
        http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
        http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <!-- ========================= DATASOURCE ========================= -->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/pd" />
        <property name="username" value="root" />
        <property name="password" value="root" />
    </bean>
    <!-- ========================= HIBERNATE ========================== --
>
    <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactory
Bean">
        <property name="dataSource" ref="dataSource" />
        <property name="annotatedClasses">
            <list>
                <value>upv.pfc.pd.User</value>
                <value> upv.pfc.pd.Authority</value>
            </list>
        </property>
        <property name="annotatedPackages">
```

```
        <list>
            <value> upv.pfc.pd </value>
        </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.dialect=org.hibernate.dialect.MySQLDialect
                hibernate.show_sql=true
                hibernate.hbm2ddl.auto=update
            </value>
        </property>
    </bean>
    <bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
    <tx:annotation-driven transaction-manager="transactionManager" />
    <bean id="hibernateTemplate"
class="org.springframework.orm.hibernate3.HibernateTemplate">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
    <!-- ======================== BEANS ======================== -->
    <bean id="userDao" class=" upv.pfc.pd.UserDaoImpl">
        <property name="hibernateTemplate" ref="hibernateTemplate" />
    </bean>
    <bean id="userService" class=" upv.pfc.pd.UserServiceImpl">
        <property name="userDao" ref="userDao" />
    </bean>
    <!-- ======================== URL ======================== -->
      <sec:http pattern="/" security="none"/>
    <sec:http  auto-config="true" use-expressions="true">
        <!-- <sec:intercept-url pattern="/**"
access="IS_AUTHENTICATED_ANONYMOUSLY" /> -->
            <sec:intercept-url pattern="/alumno/**"
access="hasRole('alumno')"/>
          <sec:intercept-url pattern="/profesor/**"
access="hasRole('profesor')"/>
            <sec:intercept-url pattern="/tutorial/**"
access="isAuthenticated()"/>
            <sec:intercept-url pattern="/documentacion/**"
access="isAuthenticated()" />
            <sec:intercept-url pattern="/**" access="permitAll" />
        <sec:form-login login-page="/home" />
        <sec:logout logout-success-url="/" delete-cookies="JSESSIONID"/>

    </sec:http>

    <sec:authentication-manager>
        <sec:authentication-provider user-service-
ref="userUserDetailsService"/>
    </sec:authentication-manager>

    <bean id="userUserDetailsService"
class=" upv.pfc.pd.UserUserDetailsService">
        <constructor-arg ref="userService"/>
    </bean>
</beans>
```

Listado 56. Archivo seguridad.xml para hibernate.

Ahora vamos a proceder a explicar cada uno de los apartados del archivo.
Lo primero de todo necesitamos más definiciones de esquemas que anteriormente, el dataSource sigue siendo el mismo ya que seguimos utilizando la misma base de datos y las mismas tablas lo que ahora el dataSource lo hemos llamado dataSource.
Ahora vamos a definir nuestra fabrica de sesiones que será el encargado de decirle al sistema donde se encuentran todos los ficheros necesarios para hibernate, el dialecto que vamos a utilizar, en este caso el de MySql 5, y también es el encargado de asociar los DAO para su correcto funcionamiento, aquí también le decimos que cargue los datos de conexión del dataSource.
Después en "annotatedClasses" se listan los archivos de las clases definidos mediante anotaciones, en "annotatedPackage" le indicaos el paquete donde se encuentran las clases y los archivos y en las propiedades de hibernate le indicamos el dialecto de la base de datos además le decimos que nos muestre por consola las consultas que realice y que modifique la base de datos solamente cuando haya cambios.
Después declaramos el transactionManager que es el encargado del manejo de las transacciones y también declaramos a continuación el HibernateTemplate que se encarga de guardar, borrar, listar y obtener los objetos de la base de datos. También se encarga de las excepciones ahorrándonos así mucho trabajo y a continuación solo nos quedara declarar los beans que vamos a utilizar el userDao y el userService.

Ahora que tenemos el archivo de "seguridad.xml" ya configurado vamos a proceder a implementar las clases y los DAO para hibernate, para saber con más precisión que significan las anotaciones podemos visitar la documentación de hibernate en http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html_single/, aquí vamos a dar por sentado unos conocimientos básicos de estas.
Todas las clases y los DAO irán en src/main/java dentro del paquete upv.pfc.pd, comencemos por definir nuestras clases que implementan las tablas de nuestras bases de datos, es decir User y Authority.

User.java

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import javax.persistence.*;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.GrantedAuthorityImpl;
import org.springframework.security.core.userdetails.UserDetails;
/**
 * The persistent class for the USERS database table.
 *
 */
@Entity
@Table(name="USERS")
public class User implements Serializable, UserDetails {
    private static final long serialVersionUID = 1L;
    //Original props
    @Id
    @Column(name="USERNAME")
    private String username;
    @Column(name="ENABLED")
    private String enabled;
    @Column(name="PASSWORD")
    private String password;
    //bi-directional one-to-one association to Authority
    @OneToOne
    @JoinColumn(name="USERNAME")
    private Authority authority;
    // Getters & Setters for original props
    public String getUsername() {
```

```
        return this.username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getEnabled() {
        return this.enabled;
    }
    public void setEnabled(String enabled) {
        this.enabled = enabled;
    }
    public String getPassword() {
        return this.password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    //Getters and setters for relation property
    public Authority getAuthority() {
        return this.authority;
    }
    public void setAuthority(Authority authority) {
        this.authority = authority;
    }
    //Spring Security props
    private transient Collection<GrantedAuthority> authorities;
    //UserDetails methods
    @Transient
    public Collection<GrantedAuthority> getAuthorities() {  return
authorities;}
    @Transient
    public boolean isAccountNonExpired() { return true;}
    @Transient
    public boolean isAccountNonLocked() { return true; }
    @Transient
    public boolean isCredentialsNonExpired() {return true; }
    @Transient
    public boolean isEnabled() {
        return getEnabled().equals("true");
    }
    @Transient
    public void setUserAuthorities(List<String> authorities) {
        List<GrantedAuthority> listOfAuthorities = new
ArrayList<GrantedAuthority>();
        for (String role : authorities) {
            listOfAuthorities.add(new GrantedAuthorityImpl(role));
        }
        this.authorities = (Collection<GrantedAuthority>) listOfAuthorities;
    }
    //Constructors
    public User() {
    }
}
```

Listado 57. Clase User

Authority.java

```java
import java.io.Serializable;
import javax.persistence.*;
/**
 * The persistent class for the AUTHORITIES database table.
 *
 */
@Entity
@Table(name="AUTHORITIES")
public class Authority implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name="USERNAME")
    private String username;
    @Column(name="AUTHORITY")
    private String authority;
    @OneToOne(mappedBy="authority")
    private User user;
    public Authority() {
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getAuthority() {
        return this.authority;
    }
    public void setAuthority(String authority) {
        this.authority = authority;
    }
    public User getUser() {
        return this.user;
    }
    public void setUser(User user) {
        this.user = user;
    }
}
```

Listado 58. Clase Authority

Ahora ya tenemos las clases implementadas vamos a implementar los DAO.

UserDao.java

```java
import java.util.List;
public interface UserDao {
    public User getUserByUserName(String userName);
    List<String> getAuthoritiesByUserName(String userName);
}
```

Listado 59. Clase UserDao.

UserService.java

```
import java.util.List;
public interface UserService {
    User getUserByUserName(String userName);
    List<String> getAuthoritiesByUserName(String userName);
}
```

Listado 60. Clase UserService.

UserDaoImpl.java

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

import org.springframework.orm.hibernate3.HibernateTemplate;
public class UserDaoImpl implements UserDao {
    HibernateTemplate hibernateTemplate;
    private String queryString = "from User where username = ?";
    public void setHibernateTemplate(HibernateTemplate arg0) {
        hibernateTemplate = arg0;
    }
    public HibernateTemplate getHibernateTemplate() {
        return hibernateTemplate;
    }

    public User getUserByUserName(String userName) {
        return (User) hibernateTemplate.find(queryString, userName).get(0);
    }

    public List<String> getAuthoritiesByUserName(String userName) {
        User u = (User) hibernateTemplate.find(queryString, userName).get(0);
        Authority a = u.getAuthority();
        String auth = a.getAuthority();
        List<String> l = new ArrayList<String>();
        l.add(auth);
        return l;
    }
}
```

Listado 61. Clase UserDaoImpl.

UserServiceImpl.java

```
import java.util.List;
public class UserServiceImpl implements UserService {
    UserDao userDao;

    public User getUserByUserName(String userName) {
        return userDao.getUserByUserName(userName);
    }

    public List<String> getAuthoritiesByUserName(String userName) {
        return userDao.getAuthoritiesByUserName(userName);
    }
    public UserDao getUserDao() {
        return userDao;
    }
```

```
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
}
```

Listado 62. Clase UserServiceImpl.

UserUserDetailsService.java

```
import java.util.List;
import org.springframework.dao.DataAccessException;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
public class UserUserDetailsService implements UserDetailsService {
    private UserService userService;
    public UserUserDetailsService(UserService userService) {
        this.userService = userService;
    }
    public UserDetails loadUserByUsername(String userName)
            throws UsernameNotFoundException, DataAccessException {
        User user;
        try {
            user = userService.getUserByUserName(userName);
        } catch (Exception e) {
            throw new UsernameNotFoundException(
                    "getUserByUserName returned null.");
        }
        List<String> authorities = userService
                .getAuthoritiesByUserName(userName);
        user.setUserAuthorities(authorities);
        return (UserDetails) user;
    }
}
```

Listado 63. Clase UserUserDetailsService.

Ahora que ya tenemos implementadas las clases podemos publicar la aplicación y ver su correcto funcionamiento.

Figura 77. Portal con hibernate.

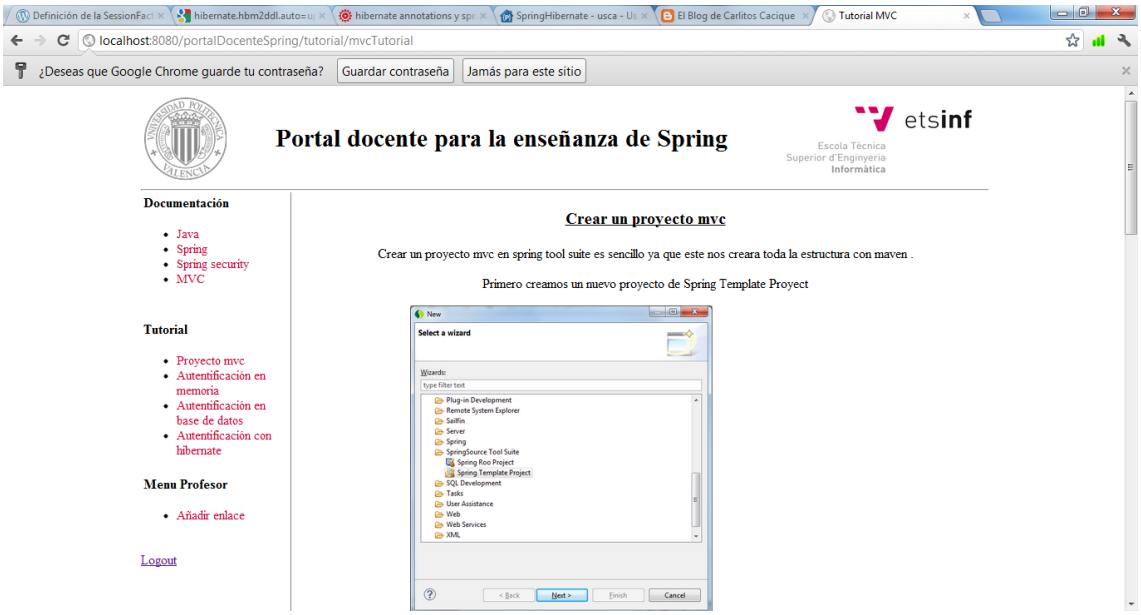

Figura 78. Portal con hibernate funcionando correctamente.

.

5 Tutorial Spring MVC, hibernate y JPA parte 1, Relaciones OneToOne

Vamos a empezar los tutoriales referentes a la utilización de Spring MVC, hibernate y JPA para acceder a base de datos, la razón para utilizar el ORM hibernate es las facilidades que nos ofrece para el manejo de los datos aligerándonos de tener que diseñar consultas SQL y facilitándonos el manejo de estos y desligándonos de tener que usar exclusivamente un gestor de base de datos, podemos cambiar fácilmente sin afectar al código de la aplicación, además la inclusión de JPA(Java Persistence Api) al ser un estándar hace que tampoco estemos ligados a un ORM exclusivamente, pudiendo cambiar este con tan solo cambiando un fichero.

En este primer tutorial nos centraremos en las relación 1:1 para lo que nos crearemos una base de datos llamada docencia que contendrá las tablas Alumnos y Direcciones (no es necesario crear las tablas la aplicación lo hará automáticamente si no existen, en cambio si es necesario crear la base de datos vacía) por lo que una alumno tendrá una dirección solamente y una dirección solo podrá pertenecer a un alumno.

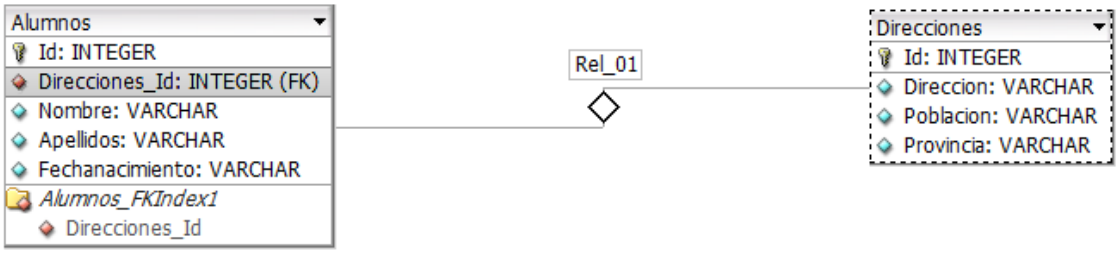

Figura 79. E/R Alumnos y Direcciones

Empezaremos creando un proyecto Spring MVC al que llamaremos relaciones OneToOne. Una vez creado el proyecto como hacemos normalmente utilizaremos Maven para incluir las dependencias necesarias para su correcto funcionamiento.

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-test</artifactId>
	<version>${org.springframework-version}</version>
</dependency>

<dependency>
  <groupId>commons-dbcp</groupId>
  <artifactId>commons-dbcp</artifactId>
  <version>1.4</version>
</dependency>
    <dependency>
       <groupId>org.hibernate</groupId>
       <artifactId>hibernate-entitymanager</artifactId>
       <version>3.4.0.GA</version>
    </dependency>
             <!-- Spring -->
                    <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>${org.springframework-version}</version>
            <type>jar</type>
            <scope>compile</scope>
        </dependency>
```

```
            <dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>${org.springframework-version}</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
    <dependency>
  <groupId>taglibs</groupId>
  <artifactId>standard</artifactId>
  <version>1.1.2</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
    <dependency>
  <groupId>commons-digester</groupId>
  <artifactId>commons-digester</artifactId>
  <version>2.1</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
    <dependency>
  <groupId>commons-collections</groupId>
  <artifactId>commons-collections</artifactId>
  <version>3.2.1</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
    <dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>3.3.2.GA</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
    <dependency>
  <groupId>javax.persistence</groupId>
  <artifactId>persistence-api</artifactId>
  <version>1.0</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
    <dependency>
  <groupId>c3p0</groupId>
  <artifactId>c3p0</artifactId>
  <version>0.9.1.2</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib-nodep</artifactId>
  <version>2.2</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
<dependency>
```

```
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-annotations</artifactId>
    <version>3.4.0.GA</version>
    <type>jar</type>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>jboss</groupId>
    <artifactId>javassist</artifactId>
    <version>3.7.ga</version>
    <type>jar</type>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.14</version>
    <type>jar</type>
    <scope>compile</scope>
  </dependency>
          <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-context</artifactId>
                <version>${org.springframework-version}</version>
                <exclusions>
                      <!-- Exclude Commons Logging in favor of SLF4j -->
                      <exclusion>
                            <groupId>commons-logging</groupId>
                            <artifactId>commons-logging</artifactId>
                       </exclusion>
                </exclusions>
          </dependency>
          <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-webmvc</artifactId>
                <version>${org.springframework-version}</version>
          </dependency>

          <!-- AspectJ -->
          <dependency>
                <groupId>org.aspectj</groupId>
                <artifactId>aspectjrt</artifactId>
                <version>${org.aspectj-version}</version>
          </dependency>

          <!-- Logging -->
          <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-api</artifactId>
                <version>1.6.1</version>
          </dependency>
          <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>jcl-over-slf4j</artifactId>
                <version>1.6.1</version>
                <scope>runtime</scope>
          </dependency>
```

```
<dependency>
	<groupId>org.slf4j</groupId>
	<artifactId>slf4j-log4j12</artifactId>
	<version>1.6.1</version>
	<scope>runtime</scope>
</dependency>
<dependency>
	<groupId>log4j</groupId>
	<artifactId>log4j</artifactId>
	<version>1.2.15</version>
	<exclusions>
		<exclusion>
			<groupId>javax.mail</groupId>
			<artifactId>mail</artifactId>
		</exclusion>
		<exclusion>
			<groupId>javax.jms</groupId>
			<artifactId>jms</artifactId>
		</exclusion>
		<exclusion>
			<groupId>com.sun.jdmk</groupId>
			<artifactId>jmxtools</artifactId>
		</exclusion>
		<exclusion>
			<groupId>com.sun.jmx</groupId>
			<artifactId>jmxri</artifactId>
		</exclusion>
	</exclusions>
	<scope>runtime</scope>
</dependency>

<!-- @Inject -->
<dependency>
	<groupId>javax.inject</groupId>
	<artifactId>javax.inject</artifactId>
	<version>1</version>
</dependency>

<!-- Servlet -->
<dependency>
	<groupId>javax.servlet</groupId>
	<artifactId>servlet-api</artifactId>
	<version>2.5</version>
	<scope>provided</scope>
</dependency>
<dependency>
	<groupId>javax.servlet.jsp</groupId>
	<artifactId>jsp-api</artifactId>
	<version>2.1</version>
	<scope>provided</scope>
</dependency>
<dependency>
	<groupId>javax.servlet</groupId>
	<artifactId>jstl</artifactId>
	<version>1.2</version>
</dependency>
```

```
<!-- Test -->
<dependency>
	<groupId>junit</groupId>
	<artifactId>junit</artifactId>
	<version>4.7</version>
	<scope>test</scope>
</dependency>
```

Listado 64. Dependencias para relaciones OneToOne.

Estas dependencias son las que incluiremos en todos los proyectos en los que utilicemos Spring MVC, hibernate y JPA por lo que serán las mismas para los siguiente tutoriales. Una vez guardado el proyecto podemos empezar a crear los ficheros necesarios.

Empezaremos por el fichero de persistencia al que llamaremos "persistence.xml", el cual crearemos en la carpeta META-INF en este fichero tendremos todas las opciones de configuración de nuestro ORM, como puede ser la URL de conexión a la base de datos, el password, el usuario, si queremos que nos muestre por consola las consultas SQL que realiza, el dialecto de la base de datos, etc.…

```
<?xml versión="1.0" encoding="UTF-8"?>
<persistence  xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd "
version="2.0">
	<persistence-unit name="JpaPersistenceUnit" transaction-
type="RESOURCE_LOCAL" >

		<class>upv.pfc.jc.Alumnos</class>
		<class>upv.pfc.jc.Direcciones</class>

		 <properties>
		<property name="hibernate.hbm2ddl.auto" value="update"/>
		<property name="hibernate.archive.autodetection" value="class,
hbm"/>
		<property name="hibernate.show_sql" value="true"/>
		<property name="hibernate.connection.driver_class"
value="com.mysql.jdbc.Driver"/>
		<property name="hibernate.connection.password" value="root"/>
		<property name="hibernate.connection.url"
value="jdbc:mysql://localhost/docencia"/>
		<property name="hibernate.connection.username" value="root"/>
		<property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
		<property name="hibernate.c3p0.min_size" value="5"/>
		<property name="hibernate.c3p0.max_size" value="20"/>
		<property name="hibernate.c3p0.timeout" value="300"/>
		<property name="hibernate.c3p0.max_statements" value="50"/>
		<property name="hibernate.c3p0.idle_test_period" value="3000"/>
	  </properties>
	</persistence-unit>
</persistence>
```

Listado 65. Fichero persistence.xml.

Además como podemos observar también hemos declarado las dos clases que vamos a utilizar Alumnos y Direcciones que se encuentra dentro del paquete upv.pfc.jc.

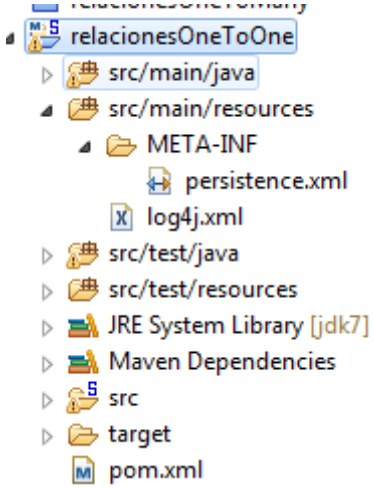

Figura 80. Directorios con fichero persistence.xml.

Ahora crearemos el fichero de configuración "hibernate-context.xml" que tendrá nuestra configuración para la aplicación.

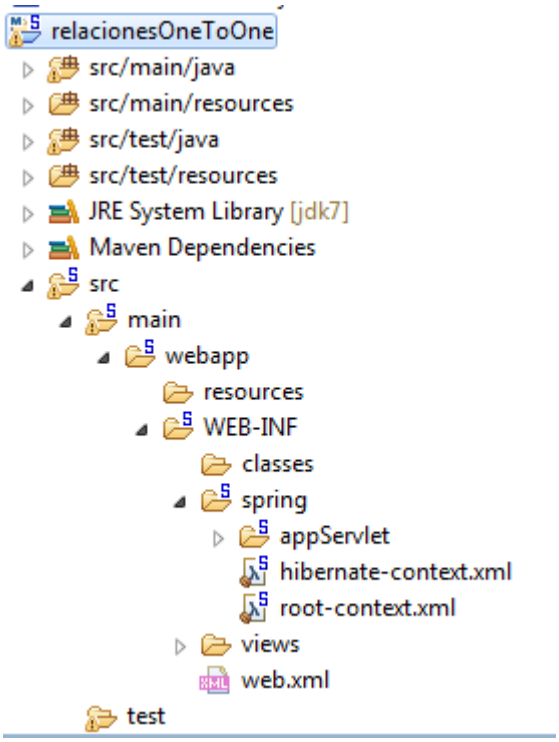

Figura 81. Directorios con fichero hibernate-context.xml.

En este archivo debemos crear y declarar nuestro DataSource que es el que va a utilizar nuestra aplicación.

```
	<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
	<property name="driverClassName" value="com.mysql.jdbc.Driver"/>
```

```
        <property name="url" value="jdbc:mysql://localhost/docencia"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>
```

Listado 66. Declaración del DataSource para OneToOne.

Ahora debemos declarar nuestro EntityManagerFactory y le debemos agregar el dataSource que hemos creado anteriormente además de la unidad de persistencia que hemos creado anteriormente llamada JpaPersistenceUnit y le declaramos nuestro jpaVendorAdapter agregándole como propiedades el tipo de gestor de base de datos y la opción de mostrar el SQL.

```
      <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
              <property name="persistenceUnitName"
value="JpaPersistenceUnit"/>
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
            <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
            <property name="database" value="MYSQL" />
            <property name="showSql" value="true" />
            </bean>
        </property>
    </bean>
```

Listado 67. Declaración del entityManagerFactory.

Ahora debemos crear nuestro transactionManager y añadirle como propiedades nuestro DataSource y nuestro entityManagerFactory.

```
          <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
            <property name="entityManagerFactory"
ref="entityManagerFactory"/>
            <property name="dataSource" ref="dataSource" />
      </bean>
```

Listado 68. Declaración del transactionManager.

Bien después de todo esto nuestro fichero debe quedar de la siguiente manera.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans       xmlns="http://www.springframework.org/schema/beans"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:p="http://www.springframework.org/schema/p"
             xmlns:tx="http://www.springframework.org/schema/tx"
             xmlns:context="http://www.springframework.org/schema/context"
             xsi:schemaLocation="http://www.springframework.org/schema/beans
                   http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd
                   http://www.springframework.org/schema/tx
                   http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
                   http://www.springframework.org/schema/context
                   http://www.springframework.org/schema/context/spring-
context-3.0.xsd">
    <tx:annotation-driven/>
```

```
    <tx:jta-transaction-manager/>
    <tx:annotation-driven transaction-manager="transactionManager" />

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

      <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
              <property name="persistenceUnitName"
value="JpaPersistenceUnit"/>
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
            <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
            <property name="database" value="MYSQL" />
            <property name="showSql" value="true" />
            </bean>
        </property>
    </bean>
          <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
             <property name="entityManagerFactory"
ref="entityManagerFactory"/>
             <property name="dataSource" ref="dataSource" />
      </bean>


</beans>
```

Listado 69. Archivo hibernate-context.xml

Ahora antes de crear nuestras entidades debemos añadir al web.xml una referencia a nuestro hibernate-context.xml para que sea cargado por la aplicación.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
    /WEB-INF/spring/root-context.xml
    /WEB-INF/spring/hibernate-context.xml
    </param-value>
</context-param>
```

Figura 82. Con la etiqueta "context-param" se indica la ubicación de los ficheros de configuración de hibernate.

Ahora ya podemos empezar a crear nuestras entidades con anotaciones, primero crearemos en src/main/java dentro del paquete upv.pfc.jc la clase "Alumnos.java", después de añadirle el código quedara de la siguiente manera.

```
package upv.pfc.jc;

import java.io.Serializable;
```

```java
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.OneToOne;

@Entity
@Table(name="ALUMNOS")
public class Alumnos implements Serializable {
	private static final long serialVersionUID = -5527566248002296042L;
	@Id
	@Column(name="ID")
	@GeneratedValue
	private Integer id;

	@Column(name="Nombre")
	private String nombre;

	@Column(name="Apellidos")
	private String apellidos;

	@Column(name="FechaNacimiento")
	private String fechanacimiento;

	@OneToOne(fetch=FetchType.EAGER,cascade =CascadeType.ALL)
	@JoinColumn(name="DIRECCIONES_ID")
	private Direcciones Direccion;

	public Integer getId() {
		return id;
	}

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

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String getApellidos() {
		return apellidos;
	}

	public void setApellidos(String apellidos) {
		this.apellidos = apellidos;
	}
```

```
	public String getFechanacimiento() {
		return fechanacimiento;
	}

	public void setFechanacimiento(String fechanacimiento) {
		this.fechanacimiento = fechanacimiento;
	}

	public Direcciones getDireccion() {
		return Direccion;
	}

	public void setDireccion(Direcciones direccion) {
		Direccion = direccion;
	}

	public static long getSerialversionuid() {
		return serialVersionUID;
	}

	@Override
	public String toString() {
		return "Alumnos [id=" + id + ", nombre=" + nombre + ",
apellidos="
				+ apellidos + ", fechanacimiento=" +
fechanacimiento
				+ ", Direccion=" + Direccion + "]";
	}



}
```

Listado 70. Clase Alumnos.java

Ahora vamos explicar las diferentes anotaciones que se encuentran en esta clase.

Para empezar la clase debe implementar la interface "Serializable", antes de declarar la clase vemos la anotación @Entity esta es para informar al proveedor de persistencia que cada instancia de esta clase es una entidad. La anotación @Table simplemente la utilizamos para definir el nombre de la tabla con la que se está mapeando la clase, sino se utiliza el nombre de la tabla corresponderá al nombre de la clase.

@Id: especifica que es la clave principal.

@Column: Especifica una columna

@GeneratedValue: especifica que el valor de esta columna se generara automáticamente.

Ahora vamos a ver la anotación @OneToOne, esta anotación indica que existe una relación de 1:1 con otra entidad en este caso Direcciones, los atributos que le hemos especificado como fetch EAGER especifican el tipo de lectura, existen dos tipos de lectura Lazy(perezosa o retardada) y Eager en las relaciones 1:1 el tipo por defecto es EAGER esto quiere decir que cuando consultemos una entidad nos traeremos todas sus entidades relacionadas también, en las relaciones de 1:1 esto puede ser asumible pero en las relaciones tipo de N:N es inasumible por lo que el tipo por defecto de estas es Lazzy, es decir que cuando consultamos una entidad solo nos traemos esa entidad y nos traeremos las entidades relacionadas solamente cuando las consultemos, en cuanto a cascade=CascadeType.ALL , significa que para cada borrado, actualización o inserción se hará en cascada , también podemos especificar que solo se haga el borrado en cascada CascadeType.REMOVE , etc..

@JoinColumn simplemente sirve para indicar el nombre del campo que hará referencia a la relación, sino lo hacemos la aplicación generar uno por nosotros.

Bueno después solo nos quedara generar los getters y setters correspondientes y ya tenemos nuestra primera entidad.

Ahora vamos a implementar la entidad Direcciones.

```
package upv.pfc.jc;

import java.io.Serializable;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.OneToOne;

@Entity
@Table(name="DIRECCIONES")
public class Direcciones implements Serializable {

	@Id
	@Column(name="ID")
	@GeneratedValue
	private Integer id;

	@Column(name="Direccion")
    private String direccion;

	@Column(name="Poblacion")
	private String poblacion;

	@Column(name="Provincia")
	private String provincia;

	@OneToOne(cascade =CascadeType.ALL,mappedBy="Direccion")
	private Alumnos alumnos;

	public Integer getId() {
		return id;
	}

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

	public String getDireccion() {
		return direccion;
	}

	public void setDireccion(String direccion) {
		this.direccion = direccion;
	}
```

```
	public String getPoblacion() {
		return poblacion;
	}

	public void setPoblacion(String poblacion) {
		this.poblacion = poblacion;
	}

	public String getProvincia() {
		return provincia;
	}

	public void setProvincia(String provincia) {
		this.provincia = provincia;
	}

	public Alumnos getAlumnos() {
		return alumnos;
	}

	public void setAlumnos(Alumnos alumnos) {
		this.alumnos = alumnos;
	}

	@Override
	public String toString() {
		return "Direcciones [id=" + id + ", direccion=" + direccion
				+ ", poblacion=" + poblacion + ", provincia=" +
provincia
				+ ", alumnos=" + alumnos + "]";
	}
}
```

Listado 71. Clase Direcciones.java

Si no especificamos en Direcciones ninguna anotación sobre la relación, esta sería unidireccional, es decir que solo Alumnos sabría de la existencia de Direcciones y no al contrario, en este caso vamos a hacer una relación de 1:1 bidireccional ya que las dos entidades tendrán una referencia a la otra.
La única anotación nueva que vemos aquí es un atributo en la relación llamado mappedBy, este atributo indica que este campo está relacionado con el campo Direcciones de la clase Alumnos , al meter este atributo en Direcciones estamos haciendo que la clase Alumnos sea la dueña de la relación.

Una vez finalizada la creación de nuestras entidades vamos a construir las clases que harán uso de estas entidades, a las clases las llamaremos "AlumnoDAO" y "DireccionesDAO".

Vamos a ver el archivo "AlumnoDAO".

```
package upv.pfc.jc;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Query;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
```

```
@Repository
public class AlumnoDAO {

	@Autowired
	protected EntityManagerFactory emf;
	public void setEntityManagerFactory(EntityManagerFactory emf)
	{
		this.emf=emf;
	}

	public List<Alumnos> getAll()
	{
	EntityManager em=this.emf.createEntityManager();
     Query query=em.createQuery("FROM Alumnos");
    @SuppressWarnings("unchecked")
     List<Alumnos>al=query.getResultList();
    return al;
	}

	public void add(Alumnos alumno )
	{
	EntityManager em=this.emf.createEntityManager();
	em.getTransaction().begin();
	em.persist(alumno);
	em.getTransaction().commit();
	}

	public void delete(int id)
	{
		EntityManager em=this.emf.createEntityManager();

		Alumnos alumno=null;
		em.getTransaction().begin();
		alumno=em.find(Alumnos.class, id);
		em.remove(alumno);
		em.getTransaction().commit();
	}

	public Alumnos getAlumnos(int id)
	{
		Alumnos alumno=null;
		EntityManager em=this.emf.createEntityManager();
		em.getTransaction().begin();
		alumno=em.find(Alumnos.class, id);
		em.getTransaction().commit();
		return alumno;
	}

	public void editAlumno(Alumnos alumno)
	{
		EntityManager em=this.emf.createEntityManager();
		Alumnos alumno1=null;
		em.getTransaction().begin();
		alumno1=em.find(Alumnos.class,alumno.getId());
		em.getTransaction().commit();
```

```
            em.getTransaction().begin();
            alumno1.setApellidos(alumno.getApellidos());
            alumno1.setNombre(alumno.getNombre());
            alumno1.setFechanacimiento(alumno.getFechanacimiento());
            alumno1.setDireccion(alumno.getDireccion());
            em.merge(alumno1);

            em.getTransaction().commit();

      }
}
```

Listado 72. Clase AlumnosDAO.java

Lo primero en que nos fijamos es en las anotaciones @Repository y @Autowired que están al principio del documento y son muy importantes ya que nos sirven para inyectar el fichero en el contexto de Spring.
Empecemos por definir nuestro entityManagerFactory para después poder crear nuestros EntityManager que son las que nos abrirán y cerraran las transacciones para operar con la base de datos, necesitamos indicarle antes la anotación @Autowired para que lo inyecte en la aplicación.

Despues de esto solo tenemos que crear los métodos para trabajar con la entidad, como add, getAll, delete, ect...
Para el método getAll que nos devuelve todos los alumnos que están en la base de datos no es necesario abrir ninguna transacción, en cambio para los otros métodos si es necesario crear una transacción y cerrarla al acabar de operar con una excepción.
El método modificar necesita de dos transacciones diferentes ya que JPA no deja modificar una entidad conectada a la base de datos, lo que hacemos es abrir una primera transacción para buscar y recuperar la entidad que necesitamos, hasta aquí como en las demás funciones, el problema es que si no cerramos la transacción la entidad todavía está conectada a la base de datos por lo que tenemos que cerrarla, después modificamos la entidad, volvemos a abrir la transacción y hacemos un merge, es decir actualizamos la entidad, así no nos dará ningún error.
Ahora podemos ver que el fichero "DireccionesDAO" es idéntico.

```
package upv.pfc.jc;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Query;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository
public class DireccionesDAO {

      @Autowired
      protected EntityManagerFactory emf;
      public void setEntityManagerFactory(EntityManagerFactory emf)
      {
            this.emf=emf;
      }

      public List<Direcciones> getAll()
      {
            EntityManager em=this.emf.createEntityManager();
            Query query=em.createQuery("FROM Direcciones");
            @SuppressWarnings("unchecked")
            List<Direcciones>dir=query.getResultList();
            return dir;
```

```
	}

	public void add(Direcciones dir)
	{
		EntityManager em=this.emf.createEntityManager();
		em.getTransaction().begin();
		em.persist(dir);
		em.getTransaction().commit();
	}
	public void delete(int id)
	{
	    Direcciones dir=null;
		EntityManager em=this.emf.createEntityManager();
	    em.getTransaction().begin();
	    dir=em.find(Direcciones.class,id);
	    em.remove(dir);
	    em.getTransaction().commit();

	}
	public Direcciones getDirecciones(int id)
	{
		Direcciones dir=null;
		EntityManager em=this.emf.createEntityManager();
		em.getTransaction().begin();
		dir=em.find(Direcciones.class,id);
		em.getTransaction().commit();
		return dir;
	}

	public void editDirecciones(Direcciones dir)
	{
		Direcciones direc=null;
		EntityManager em=this.emf.createEntityManager();
		em.getTransaction().begin();
		direc=em.find(Direcciones.class,dir.getId());
	    direc.setDireccion(dir.getDireccion());
	    direc.setPoblacion(dir.getPoblacion());
	    direc.setProvincia(dir.getProvincia());
		em.getTransaction().commit();
	}

}
```

Listado 73. Clase DireccionesDAO.java

Ahora que ya tenemos los dos ficheros debemos podemos empezar a construir nuestra web como haríamos normalmente, lo único que deberemos inyectar también los ficheros en el controlador para poder interactuar con la base de datos.

6 Tutorial Spring MVC, hibernate y JPA parte 2, Relaciones OneToMany

En este tutorial nos centraremos en las relaciones de uno a muchos poniendo el ejemplo de la relación entre los alumnos y sus emails dado que un alumno puede tener varios emails y un email solo puede pertenecer a un alumno.

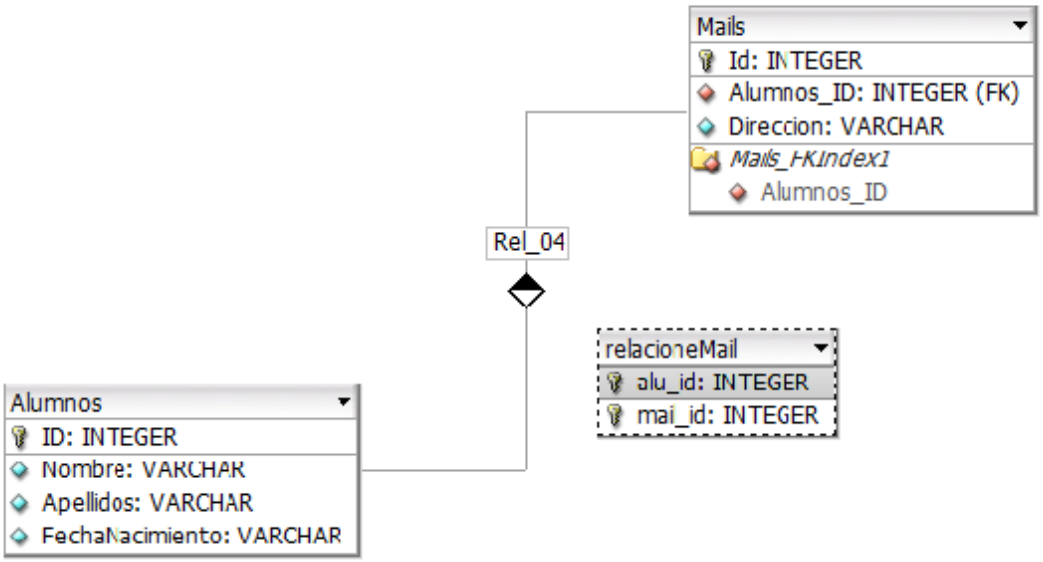

Figura 83. E/R de Alumnos y Mails.

En la imagen podemos ver la relación de las tablas, además hemos incluido a modo de ejemplo la nueva tabla que se creara pues para esta relación hemos decidido usar la opción de crear una tabla adicional con JPA (en la relaciones de uno a muchos es opcional, en la de muchos a muchos esta tabla es obligatoria).
Ahora vamos a ver cómo crear esta tabla, como hemos hecho en el primer tutorial tenemos que crear nuestras clases Alumnos y Mail. Pero antes tenemos que agregar las clases a nuestro fichero de persistencia como hemos hecho en el otro tutorial.

```
<persistence-unit name="JpaPersistenceUnit" transaction-type="RESOURCE_LOCAL" >
    <class>upv.pfc.jc.Alumnos</class>
    <class>upv.pfc.jc.Mail</class>

     <properties>
```

Figura 1. Declaración de las clases en el fichero de persistencia.

Ahora debemos crearnos las clases con todas las anotaciones necesarias.

```
package upv.pfc.jc;

import java.io.Serializable;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinTable;
```

```
import javax.persistence.Table;
import javax.persistence.OneToMany;
import javax.persistence.JoinColumn;


@Entity
@Table (name="ALUMNOS")
public class Alumnos implements Serializable {

	private static final long serialVersionUID = -5527566248002296042L;
	@Id
	@Column (name="ID")
	@GeneratedValue
	private Integer id;

	@Column(name="Nombre",nullable=false)
	private String nombre;

	@Column(name="Apellidos",nullable=false)
	private String apellidos;

	@Column(name="FechaNacimiento")
	private String fechanacimiento;

	@OneToMany(cascade=CascadeType.ALL)
	@JoinTable(name="relacionEmail",joinColumns={@JoinColumn(name="alu_id"
,referencedColumnName="ID")},inverseJoinColumns={@JoinColumn(name="mail_id",r
eferencedColumnName="ID",unique=true)})
	private List<Mail> email;

	public Integer getId() {
		return id;
	}

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

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String getApellidos() {
		return apellidos;
	}

	public void setApellidos(String apellidos) {
		this.apellidos = apellidos;
	}

	public String getFechanacimiento() {
		return fechanacimiento;
	}
```

```
	public void setFechanacimiento(String fechanacimiento) {
		this.fechanacimiento = fechanacimiento;
	}

	public List<Mail> getEmail() {
		return email;
	}

	public void setEmail(List<Mail> email) {
		this.email = email;
	}
}
```

Listado 74. Clase Alumnos de relaciones OneToMany.

Esta clase es prácticamente igual que la del anterior tutorial pero con la diferencia de la relación OneToMany

```
	@OneToMany(cascade=CascadeType.ALL)
	@JoinTable(name="relacionEmail",joinColumns={@JoinColumn(name="alu_id"
,referencedColumnName="ID")},inverseJoinColumns={@JoinColumn(name="mail_id",r
eferencedColumnName="ID",unique=true)})
	private List<Mail> email;
```

Listado 75. Declaración de la relación OneToMany.

La primera anotación @OneToMany (cascade=CascadeType.ALL) es para definir el tipo de relación y la opción de CacadeType.ALL es para decirle que todas las operaciones sobre la relación se realicen en cascada. Después la anotación @JoinTable es para crear una tabla intermedia de la relación con el nombre que le hemos indicado en la opción name, después de esto tenemos la opción joinColumn que sirve para crear la columna de la nueva tabla que se referirá a la tabla Alumnos en este caso le pondremos de nombre a la columna alu_id y se referirá a la columna ID. Despues tenemos la opción inverseJoinColumns que se referiré a la otra tabla que va a formar parte de la relación en este caso Mail y como antes tenemos que indicar el nombre de la columna y a que columna de la tabla se referirá.
Por último debemos crearnos una colección de objetos Mail ya que al ser una relación de uno a muchos tendremos muchos Mail para un mismo alumno.

7 Tutorial Spring MVC, hibernate y JPA parte 3, Relaciones ManyToMany

En este último tutorial vamos a ver las relaciones de muchos a muchos con el ejemplo de Alumnos y Asignaturas, en esta relación es obligatorio el uso de una JoinTable para esta relación.

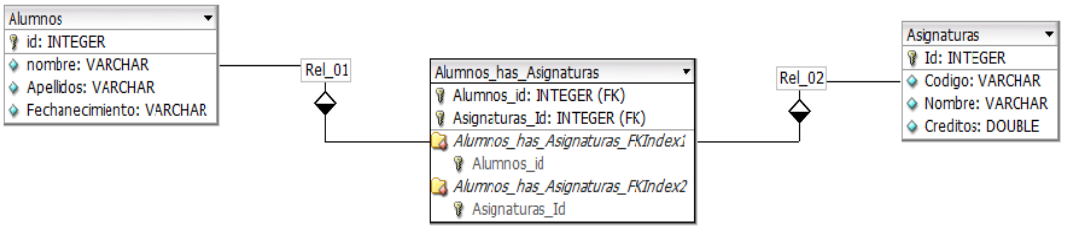

Figura 84. E/R de Alumnos y Asignaturas.

Ahora debemos hacer como en los anteriores tutoriales, incluir en el fichero de persistencia las clases que vamos a utilizar en este caso Asignaturas y Alumnos.

```
<persistence-unit name="JpaPersistenceUnit" transaction-type="RESOURCE_LOCAL" >
    <class>upv.pfc.jc.Alumnos</class>
    <class>upv.pfc.jc.Asignaturas</class>
```

Figura 85. Declaración de las clases en el fichero de persistencia para las relaciones ManyToMany.

Ahora debemos declarar las clases, esto será exactamente igual que en anterior tutorial, el de las relaciones OneToMany.

```
package upv.pfc.jc;

import java.io.Serializable;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

@Entity
@Table(name="ALUMNOS")
public class Alumnos implements Serializable {

	private static final long serialVersionUID = -5527566248002296042L;
	@Id
	@Column(name="ID")
	@GeneratedValue
	private Integer id;

	@Column(name="Nombre",nullable=false)
	private String nombre;

	@Column(name="Apellidos",nullable=false)
	private String apellidos;

	@Column(name="FechaNacimiento")
	private String fechanacimiento;

@ManyToMany
	@JoinTable(
			name="Matriculado",
			joinColumns={@JoinColumn(name="ALU_ID",
referencedColumnName="ID")},
			inverseJoinColumns={@JoinColumn(name="ASIG_ID",
referencedColumnName="ID")})
	private List<Asignaturas> asignaturas;

	public Integer getId() {
		return id;
	}
```

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

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String getApellidos() {
		return apellidos;
	}

	public void setApellidos(String apellidos) {
		this.apellidos = apellidos;
	}

	public String getFechanacimiento() {
		return fechanacimiento;
	}

	public void setFechanacimiento(String fechanacimiento) {
		this.fechanacimiento = fechanacimiento;
	}

	public List<Asignaturas> getAsignaturas() {
		return asignaturas;
	}

	public void setAsignaturas(List<Asignaturas> asignaturas) {
		this.asignaturas = asignaturas;
	}
	public  void añadirBorrarAsignatura(Asignaturas as)
	{
		this.asignaturas.remove(as);
	}
}
```

Listado 76. Clase Alumnos.java de relación ManyToMany.

La clase Alumnos es exactamente igual que la anterior exceptuando que ahora la inverseJoinColumns hace referencia a la clase Asignaturas y tenemos una colección de Asignaturas.

Ahora vamos a ver la clase asignaturas.

```
package upv.pfc.jc;

import java.io.Serializable;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
```

```java
import javax.persistence.ManyToMany;

@Entity
@Table(name="ASIGNATURAS")
public class Asignaturas implements Serializable {
	private static final long serialVersionUID = -5527566248002296042L;
	@Id
	@Column(name="ID")
	@GeneratedValue
	private Integer id;

	@Column(name="Nombre",nullable=false,unique=true)
	private String nombre;

	@Column(name="Codigo",nullable=false,unique=true)
	private String codigo;

	@Column(name="Creditos",nullable=false)
	private double creditos;

	@ManyToMany(mappedBy="asignaturas")
	private List<Alumnos> matriculados;

	public Integer getId() {
		return id;
	}

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

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public String getCodigo() {
		return codigo;
	}

	public void setCodigo(String codigo) {
		this.codigo = codigo;
	}

	public double getCreditos() {
		return creditos;
	}

	public void setCreditos(double creditos) {
		this.creditos = creditos;
	}

	public List<Alumnos> getMatriculados() {
		return matriculados;
```

```
	}

	public void setMatriculados(List<Alumnos> matriculados) {
		this.matriculados = matriculados;
	}
}
```

Listado 77. Clase Asignaturas.java

Aquí si hay diferencias respecto al anterior tutorial, en esta clase si hace falta incluir la anotación @ManyToMany por que la clase tiene que tener una colección de Alumnos que van a ser los matriculados en esa asignatura. La opción mappedBy, como hemos explicado antes, es para decirle que la dueña de la relación es la clase alumnos.

Ahora solo nos quedaría implementar los DAO como antes y los controladores.

Bibliografía

http://es.wikipedia.org/wiki/Java_EE

http://enciclopedia.us.es/index.php/Java_2_Enterprise_Edition

http://www.proactiva-calidad.com/java/arquitectura/index.html

http://osgg.net/omarsite/resources/proceedings/Introduccion%20a%20J2EE.pdf

http://www.oracle.com/us/corporate/press/330073

http://docs.oracle.com/javaee/5/api/

http://ccia.ei.uvigo.es/docencia/SCS/Tema5-1.pdf

http://es.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones

http://es.wikipedia.org/wiki/Framework

http://www.datuopinion.com/java-ee

http://aprendiendojee.wordpress.com/2010/04/23/%C2%BFque-es-java/

http://"GlassFish".java.net/es/

http://casidiablo.net/documentacion-en-espanol-de-"GlassFish"/

http://"GlassFish".java.net/es/public/users.html

http://www.marlonj.com/blog/2009/10/que-es-"GlassFish"/

http://certified-es.blogspot.com.es/2010/12/arquitectura-java-ee-y-spring-framework.html#!/2010/12/arquitectura-java-ee-y-spring-framework.html

http://www.slideshare.net/cptanalatriste/arquitectura-y-diseo-de-aplicaciones-java-ee

http://www.javadabbadoo.org/cursos/infosintesis.net/javaee/arquitecturajavaee/index.html

http://es.wikipedia.org/wiki/Enterprise_JavaBeans

http://www.jtech.ua.es/j2ee/2003-2004/abierto-j2ee-2003-2004/ejb/sesion01-traspas.pdf

http://www.oracle.com/technetwork/articles/javase/gfbasics-140371.html

http://www.osmosislatina.com/java/ejb.htm

http://es.scribd.com/doc/62097880/15/Contenedor-EJB

http://cea-tamara.blogspot.com.es/2008/01/contenedor-ejb.html

http://www.slideshare.net/oraclemarketing/4b-roger-freixa-oracle-enterprise-glass-fish

https://blogs.oracle.com/jaimecid/entry/presentacion_”GlassFish”_javaee5

https://blogs.oracle.com/jaimecid/resource/JC_”GlassFish”_JavaEE5.pdf

http://docs.oracle.com/cd/E19226-01/821-1335/821-1335.pdf

http://openaccess.uoc.edu/webapps/o2/bitstream/10609/8052/1/rdelo_TFC0611memoria.pdf

http://openaccess.uoc.edu/webapps/o2/bitstream/10609/609/1/40050tfc.pdf

http://www.leandroiriarte.com.ar/spanish/acegi_spring_security.php

http://www.leandroiriarte.com.ar/spanish/web_mvc.php

http://www.sicuma.uma.es/sicuma/independientes/argentina08/Badaracco/index.htm

http://seamcity.madeinxpain.com/archives/comparativa-spring

http://mikiorbe.wordpress.com/2008/11/20/%C2%BFque-es-spring-explicacion-basica-y-descripcion/

http://noinventeslarueda.blogspot.com.es/2010/11/modulos-de-spring.html

http://www.elorigen.com/466/cuales-son-los-modulos-de-spring-java

http://www.summarg.com/foro/threads/493-Spring-Framework

http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/sanchez_r_ma/capitulo3.pdf

http://www.javadabbadoo.org/cursos/infosintesis.net/javaee/spring/introduccion/modulos/index.html

http://noinventeslarueda.blogspot.com.es/2010/11/que-es-spring.html

http://www.it.uc3m.es/mario/si/Introduccion_a_Spring.pdf

http://hop2croft.wordpress.com/2011/09/10/ejemplo-basico-de-spring-mvc-con-maven/

http://static.springsource.org/spring/docs/2.0.x/reference/mvc.html

http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=springide202

http://pdf.rincondelvago.com/desarrollo-de-n-capas.html

http://www.docirs.cl/arquitectura_tres_capas.htm

http://bulma.net/body.phtml?nIdNoticia=734

http://www.springsource.org/downloads/sts

http://www.palermo.edu/ingenieria/downloads/introduccion_spring_framework_v1.0.pdf

http://metodologiasdesistemas.blogspot.com.es/2007/10/que-es-un-orm-object-relational-mapping.html

http://es.wikipedia.org/wiki/Mapeo_objeto-relacional

http://www.srbyte.com/2009/09/que-es-orm.html

http://es.wikipedia.org/wiki/Java_Persistence_API

http://www.coplec.org/?q=book/export/html/240

http://informatica.uv.es/iiguia/DBD/Teoria/jdo.pdf

http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=hibernate

http://blogandshare.blogspot.com.es/2005/10/hibernate-comienza-el-frio.html

http://www.epidataconsulting.com/tikiwiki/tiki-pagehistory.php?page=Hibernate&diff2=21&diff_style=sideview

http://www.dosideas.com/wiki/Hibernate_Con_Spring

http://www.springhispano.org/?q=node/255

http://www.aprendiendojava.com.ar/index.php?topic=54.0

http://oness.sourceforge.net/proyecto/html/ch03s02.html

http://wiki-aplidaniel.wikispaces.com/Modelo+n-capas

http://demetole.blogspot.com.es/2010/04/"GlassFish"-o-tomcat-cual-le-conviene.html

http://esctoexit.wordpress.com/2011/10/31/instalar-hibernate-tools-en-eclipse-indigo/