Daniel Ventas López

Ingeniero técnico en informática por la UCLM.

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2014-01-14 Tutorial visitado 8.079 veces Descargar en PDF

Tutorial holamundo con Spring MVC, Hibernate y aspectos.

0. Índice de contenidos.

1. Entorno

Este tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil Mac Book Pro 17" (2,93 Ghz Intel Core 2 Duo, 8 GB DDR3)
- Sistema Operativo: Mac OS X Mavericks 10.9
- Spring Framework 3.0
- IDE: Eclipse Kepler + plugin m2e

2. Introducción.

En este tutorial vamos a hacer un holamundo para probar las distintas tecnologías como son Spring MVC, maven para gestionar las dependencias fácilmente, Hibernate para persistencia, crearemos un aspecto a modo de holamundo para ver cómo funciona y por último crearemos un test de prueba con JUnit.

Para el ejemplo, vamos a crear una base de datos en la que vamos a tener unos libros y los vamos a mostrar por pantalla, y teniendo una opción para introducir nuevos libros.

3. Preparación del entorno en eclipse con Maven.

EL primer paso para empezar el tutorial, es crear un nuevo proyecto en eclipse. Voy a suponer que ya está instalado todo lo necesario como maven y el plugin de Spring.

En este tutorial podemos ver cómo crear el proyecto para SpringMVC.

El pom de nuestro proyecto maven va a ser el siguiente, para que se descargen las dependencias necesarias:

```
view plain  print  ?
01.    <!--
02.
03.    /pom.xml
04.
05.     -->
06.
07. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
08.    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
09.    <modelVersion>4.0.0</modelVersion>
10.    <groupId>com.autentia.libreria</groupId>
11.    <artifactId>libreria</artifactId>
12.    <version>1.0.0-SNAPSHOT</version>
13.    <packaging>war</packaging>
```


Últimas Noticias

» Curso JBoss de Red Hat

» Si eres el responsable o líder técnico, considérate desafortunado. No puedes culpar a nadie por ser gris

» Portales, gestores de contenidos documentales y desarrollos a medida

» Comentando el libro Start-up Nation, La historia del milagro económico de Israel, de Dan Senor & Salu Singer

» Screencasts de programación narrados en Español

Histórico de noticias

Últimos Tutoriales

» Tutorial Apple Watch

» Analizando WSO2 Business Rules Server

» Hooks en Cordova:Desarrollo de aplicaciones móviles multiplataforma con Apache

```
14.
15.    <properties>
16.      <spring.version>3.2.3.RELEASE</spring.version>
17.      <hibernate.version>4.2.2.Final</hibernate.version>
18.      <javax_servlet.version>2.5</javax_servlet.version>
19.      <jstl.version>1.2</jstl.version>
20.      <hsql.version>2.2.9</hsql.version>
21.      <aspectj.version>1.7.3</aspectj.version>
22.      <junit.version>4.11</junit.version>
23.    </properties>
24.
25.    <dependencies>
26.      <!-- Spring -->
27.      <dependency>
28.        <groupId>org.springframework</groupId>
29.        <artifactId>spring-context</artifactId>
30.        <version>${spring.version}</version>
31.      </dependency>
32.      <dependency>
33.        <groupId>org.springframework</groupId>
34.        <artifactId>spring-webmvc</artifactId>
35.        <version>${spring.version}</version>
36.      </dependency>
37.      <dependency>
38.        <groupId>org.springframework</groupId>
39.        <artifactId>spring-orm</artifactId>
40.        <version>${spring.version}</version>
41.      </dependency>
42.          <dependency>
43.        <groupId>org.springframework</groupId>
44.        <artifactId>spring-test</artifactId>
45.        <version>${spring.version}</version>
46.      </dependency>
47.
48.      <!-- Hibernate -->
49.      <dependency>
50.        <groupId>org.hibernate</groupId>
51.        <artifactId>hibernate-core</artifactId>
52.        <version>${hibernate.version}</version>
53.      </dependency>
54.
55.      <!-- Java EE -->
56.      <dependency>
57.        <groupId>javax.servlet</groupId>
58.        <artifactId>servlet-api</artifactId>
59.        <version>${javax_servlet.version}</version>
60.        <scope>provided</scope>
61.      </dependency>
62.      <dependency>
63.        <groupId>jstl</groupId>
64.        <artifactId>jstl</artifactId>
65.        <version>${jstl.version}</version>
66.      </dependency>
67.
68.      <!-- Hypersonic Database -->
69.      <dependency>
70.        <groupId>org.hsqldb</groupId>
71.        <artifactId>hsqldb</artifactId>
72.        <version>${hsql.version}</version>
73.        <scope>runtime</scope>
74.      </dependency>
75.
76.      <dependency>
77.        <groupId>org.aspectj</groupId>
78.        <artifactId>aspectjrt</artifactId>
79.        <version>${aspectj.version}</version>
80.      </dependency>
81.      <dependency>
82.        <groupId>org.aspectj</groupId>
83.        <artifactId>aspectjweaver</artifactId>
84.        <version>${aspectj.version}</version>
85.      </dependency>
86.
87.      <!-- JUnit -->
88.      <dependency>
89.        <groupId>junit</groupId>
90.        <artifactId>junit</artifactId>
91.        <version>${junit.version}</version>
92.      </dependency>
93.    </dependencies>
94.
95.    <build>
96.      <plugins>
97.        <plugin>
98.          <groupId>org.apache.maven.plugins</groupId>
99.          <artifactId>maven-compiler-plugin</artifactId>
100.         <version>3.1</version>
101.         <configuration>
102.           <source>1.6</source>
103.           <target>1.6</target>
104.         </configuration>
105.       </plugin>
106.     </plugins>
107.   </build>
108.
109. </project>
```

Y al final del tutorial el proyecto tiene que tener una estructura como la siguiente:

Cordova utilizando AngularJS, Ionic y ngCordova

» Paradigma publish/subscribe con Spring Data Redis

» Creación paso a paso de un webscript Alfresco

Últimos Tutoriales del Autor

» Depurar Tomcat en remoto.

» Firewall de alta disponibilidad con balanceo de carga (Clúster)

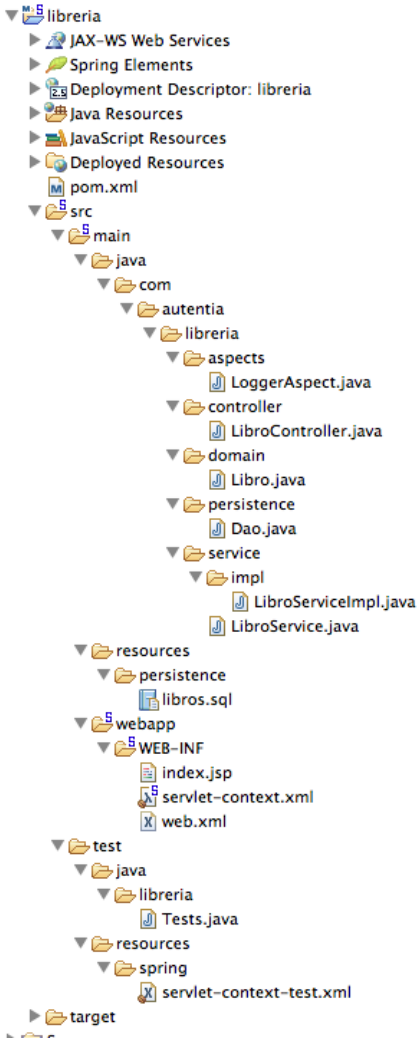
» Máquina de estados con Apache SCXML

» App iOS para conectar con periférico bluetooth 4.0

» Cómo montar un raid1 en una máquina corriendo debian.

Categorías del Tutorial

- JSF
- Spring
- Hibernate
- Java Estándar
- Maven



Y otra cosa a tener en cuenta es que la aplicación correrá bajo un tomcat, que se sale del tutorial explicar cómo instalarlo en eclipse, aunque no tiene ninguna complicación.

Configuración Spring MVC, Hibernate y Aspectos.

Para empezar nos creamos en la ruta: /webapp/WEB-INF/ un archivo xml que en mi caso le he llamado servlet-context.xml que será donde estén todas las beans y configuraciones de hibernate y aspectos. Al ser un pequeño ejemplo no se ha modularizado que es lo conveniente.

Voy a comentar en el mismo código las cosas más significativas:

```
view plain print ?
<!--

src/main/webapp/WEB-INF/servlet-context.xml

 -->


lt;beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:jdbc="http://www.springframework.org/schema/jdbc"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd
  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.2.xsd
  http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd">



<!-- Con esto Spring MVC reconoce las anotaciones como RequestMapping que nos harán falta más tarde.-->
<mvc:annotation-driven/>

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

<!-- Escanea el classpath buscando anotaciones (eg: @Service, @Repository etc) -->
<context:component-scan base-package="com.autentia.libreria.*" />


<!-- Configuración de la base de datos embebida. -->
<!-- libros.sql es el archivo que contiene el script de la base de datos. -->
<jdbc:embedded-database id="dataSource" type="HSQL">
  <jdbc:script location="classpath:persistence/libros.sql" />
</jdbc:embedded-database>

<!-- Hibernate Session Factory -->
<bean id="mySessionFactory"
  class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="packagesToScan">
    <array>
      <value>com.autentia.libreria</value>
    </array>
  </property>
  <property name="hibernateProperties">
    <value>
      hibernate.dialect=org.hibernate.dialect.HSQLDialect
    </value>
  </property>
</bean>


  <!-- Para reconocer aspectos -->
<aop:aspectj-autoproxy/>

<!-- Bean donde se encuentra el aspecto, para que lo incluya Spring -->
<bean id="imprimeLog" class="com.autentia.libreria.aspects.LoggerAspect" />


<!-- Hibernate Transaction Manager -->
<bean id="transactionManager"
  class="org.springframework.orm.hibernate4.HibernateTransactionManager">
  <property name="sessionFactory" ref="mySessionFactory" />
</bean>

<!-- Activates annotation based transaction management -->
<tx:annotation-driven transaction-manager="transactionManager" />

</beans>
```

Ahora vamos a explicar de la misma manera el fichero web.xml, que se encuentra en la misma ruta que el anterior y es en el que configuraremos los servlets.

```
view plain print ?
01.  <!--
02.
03.  src/webapp/WEB-INF/web.xml
04.
05.   -->
06.
07.
08.
09. <web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
10.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
11.   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
12.
13.
14. <!-- Aquí lo importante es el DispatcherServlet, que coge la configuración del fichero servlet-context.xml que definimos antes. --
    >
15.   <servlet>
16.     <servlet-name>appServlet</servlet-name>
17.     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
18.     <init-param>
19.       <param-name>contextConfigLocation</param-name>
20.       <param-value>/WEB-INF/servlet-context.xml</param-value>
21.     </init-param>
22.     <load-on-startup>1</load-on-startup>
23.   </servlet>
24.
25.   <!-- Aquí indicamos la ruta donde el dispatcher estará escuchando, en este caso la raíz de la web(/). -->
26.
27.   <servlet-mapping>
28.     <servlet-name>appServlet</servlet-name>
29.     <url-pattern>/</url-pattern>
30.   </servlet-mapping>
31.
32. </web-app>
```

Con esto tenemos la configuración mínima para funcionar, nos queda crear nuestras clases, el script de la base de datos, el aspecto y la vista que lo haremos en la siguiente sección.

Creando la lógica de la aplicación.

Vamos a empezar con las clases de persistencia, primeramente creamos el POJO Libro con las anotaciones de Hibernate.

```
view plain print ?
01.  /*
02.
03.   src/main/java/com/autentia/libreria/domain/Libro.java
04.
05.   */
06.
07.  //Con entity y table estamos diciendo cómo se mapea la clase en la base de datos.
08.  @Entity
09.  @Table(name = "Libro")
10.  public class Libro {
11.
12.  //Id indica a hibernate que es la primary key
13.  //Indica con qué columna de la base de datos se corresponde cada variable.
14.  //Es buena práctica llamar a las variables igual que la columna en base de datos.
15.  @Id
16.  @Column(name="id")
17.  private Integer id;
18.
19.  @Column(name="titulo")
20.  private String titulo;
21.
22.  @Column(name="isbn")
23.  private String isbn;
24.
25.  @Column(name="autor")
26.  private String Autor;
27.
28.  /*
29.
30.   Adds getters and setters...
31.
32.   */
33.
34. }
```

Ahora vamos a definir el dao, que será muy sencillo, como toda la aplicación.

```
view plain print ?
01. /*
02.
03. src/main/java/com/autentia/libreria/persistence/Dao.java
04.
05. */
06.
07. //Repository indica a Spring que es un beans y que forma parte del modelo.
08. //Es equivalente a @Component, pero viene bien especificar a que parte pertenece.
09. @Repository
10. @SuppressWarnings({ "unchecked", "rawtypes" })
11. public class Dao {
12.
13. //Autowired indica a Spring que sessionFactory tiene que inyectarlo
14. @Autowired
15. private SessionFactory sessionFactory;
16.
17. //Transaccional indica que la transacción o se completa entera o se hace rollback
18. //En los select no es necesario, pero ya que se delega Hibernate a Spring hace falta.
19. @Transactional
20. public List<Libro> findAll() {
21.
22.    Session session = sessionFactory.getCurrentSession();
23.    List pizzas = session.createQuery("from Libro").list();
24.    return pizzas;
25. }
26.
27. @Transactional
28. public void insertBook(Libro libro) {
29.
30.    sessionFactory.getCurrentSession().save(libro);
31.
32. }
```

Ya que estamos con la parte de persistencia, ahora dejo el script de creación de la base de datos y algún dato de ejemplo.

-- src/main/resources/persistence/libros.sql

```
CREATE  TABLE LIBRO (
 id INT NOT NULL ,
 titulo VARCHAR(45) NOT NULL ,
 isbn VARCHAR(50) ,
 autor VARCHAR(50) ,
 PRIMARY KEY (id) );


INSERT INTO libro VALUES ('0','Java','1234','Manuel');
INSERT INTO libro VALUES ('1','C','4567','Ricardo');
INSERT INTO libro VALUES ('2','Python','1010','Jaime');
```

No voy a explicar nada del script ya que es básico. Ahora seguimos con la parte de vista. Primero describiré el service, que será llamado por el controller de la vista y por último describiré la vista, para dejar para el final el aspecto.

En el servicio vamos a implementarlo con una interfaz, que es como se debe hacer todo, pero al ser un ejemplo sencillo solo contaremos con esta interfaz, en vuestras manos dejo implementar correctamente los demás métodos.

```
view plain print ?
01.  /*
02.
03. src/main/java/com/autentia/libreria/service/LibroService.java
04.
05.    */
06.
07.  public interface LibroService {
08.
09.    public List<Libro> findAll();
10.
11.    public void insert(Libro libro);
12.
13. }
```

Y ahora la implementación del servicio, que implementará la interfaz que acabamos de crear.

```
view plain print ?
01.  /*
02.
03. src/main/java/com/autentia/libreria/service/impl/LibroServiceImpl.java
04.
05.  */
06.
07. @Service
08. public class LibroServiceImpl implements LibroService {
09.
10.   @Autowired
11.    private Dao dao;
12.
13.    public List<Libro> findAll(){
14.       return dao.findAll();
15.    }
16.
17.    public void insert(Libro libro){
18.       dao.insertBook(libro);
19.    }
20.
21. }
```

El controlador será el siguiente:

```
view plain print ?
01. /*
02.
03. src/main/java/com/autentia/libreria/controller/LibroController.java
04.
05. */
06.
07. //Igual que hemos visto antes con repository, pero esta vez para la parte de controlador.
08. //RequestMapping forma parte de las anotaciones de springMVC que su funcionamiento se basa en recoger las peticiones
09. //que se hacen a la url relativa, en este caso la raíz, e indica a Spring que esta es la clase que maneja la vista.
10. @Controller
11. @RequestMapping("/")
12. public class LibroController {
13.
14.   //empezamos en 3 por no complicar la clase sacando de la base de datos el siguiente id, y al ser un holamundo saber
15.    private static int num = 3;
16.
17.    @Autowired
18.    private LibroService libroService;
19.
20.    //Aquí además estamos diciendo que el método usado es por petició GET.
21.    @RequestMapping(method = RequestMethod.GET)
22.    public String list(Model model) {
23.
24.    //la función del método es rellenar las variables libros y libro. una con la lista de libros
25.    //sacados de la base de datos, y libro para que pueda ser rellenada con los datos del formulario.
26.
27.      List<Libro> libros = libroService.findAll();
28.      model.addAttribute("libros", libros);
29.      model.addAttribute("libro", new Libro());
30.      return "index";
31.    }
32.
33.    @RequestMapping("/addBook")
34.    public String addBook(@ModelAttribute("libro") Libro libro) {
35.       libro.setId(num);
36.       num++;
37.       libroService.insert(libro);
38.       return "redirect:/";
39.    }
40. }
```

Una vez explicado el controlador, vamos a hacer una vista muy simple para mostrar la información.

```
view plain print ?
01.  <!--
02.
03.  src/webapp/WEB-INF/index.jsf
04.
05.  -->
06.  ...
07.
08.  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
09.  <%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
10.
11.  ...
12.
13.  <body>
14.  <h1>Lista de libros</h1>
15.  <ul>
16.
17.     <!-- Lista con todos los libros que hay en la base de datos,
18.        los saca de la variable libros. -->
19.
20.     <c:forEach var="p" items="${libros}">
21.        <li>Titulo: ${p.titulo} - Autor:  ${p.autor} - ISBN: ${p.isbn}</li>
22.     </c:forEach>
23.  </ul>
24.
25.  <!-- formulario en el que se recogen los datos para crear un nuevo libro,
26.  y al mandar llama a /addbook que será mapeado por nuestro método insertBook
27.  que definimos en LibroController. -->
28.
29.  <form:form method="post" action="addBook" commandName="libro">
30.
31.  <table>
32.  <tr>
33.     <td><form:label path="titulo">Titulo:</form:label></td>
34.     <td><form:input path="titulo" /></td>
35.  </tr>
36.  <tr>
37.     <td><form:label path="autor">Autor:</form:label></td>
38.     <td><form:input path="autor" /></td>
39.  </tr>
40.  <tr>
41.     <td><form:label path="isbn" >ISBN:</form:label></td>
42.     <td><form:input path="isbn" /></td>
43.  </tr>
44.  <tr>
45.     <td colspan="2">
46.        <input type="submit" />
47.     </td>
48.  </tr>
49.  </table>
50.  </form:form>
51.
52.
53.  </body>
```

Y ya sólo nos queda definir el aspecto que será una clase java en la cual simplemente se llama a imprimir en la salida estandar cada vez que se ejecuta un método que se llama list().

```
view plain print ?
01.  /*
02.
03.  src/main/java/com/autentia/libreria/aspects/LoggerAspect.java
04.
05.  */
06.
07.  //Indicamos que es un aspecto
08.  @Aspect
09.  public class LoggerAspect {
10.
11.     //variable que indica numero de veces que se ha llamado
12.     private static int veces = 1;
13.
14.     //Aquí hay varias partes, la anotación Before indica que se tiene que ejecutar antes de el pointcut que indiquemos dentr
15.     //El pointcut es un punto en el programa en el que poder ejecutar un aspecto, en este caso hemos indicado nuestro poi
16.     //método que se llamado desde cualquier parte que se llame list(), con al menos un parámetro.
17.     @Before("execution(* list(..))")
18.     public void imprimirlog(){
19.        System.out.println("Se ha llamado a método list : " + veces);
20.        veces++;
21.     }
22.
23.  }
```

Con esto hemos terminado de definir La lógica de nuestro programa. En resumen, tenemos una base de datos con varios libros, gestionada por hibernate. Spring MVC nos muestra la vista y controla las variables que aparecen en la vista y que la rellenan. Y tenemos un aspecto que se ejecuta cada vez que el método list() es llamado.

En la siguiente parte veremos cómo hacer un pequeño ejemplo de test, que no probaremos todo, sólo la base de datos como ejemplo.

Ejemplo test para el dao.

Ahora vamos a realizar tres simples test, en el primero comprobamos que en la base de datos existen 3 libros, los cuales introducimos con el script libros.sql. En el segundo probamos que se realiza el insert correctamente. Y en el último comprobamos que se borran libros correctamente.

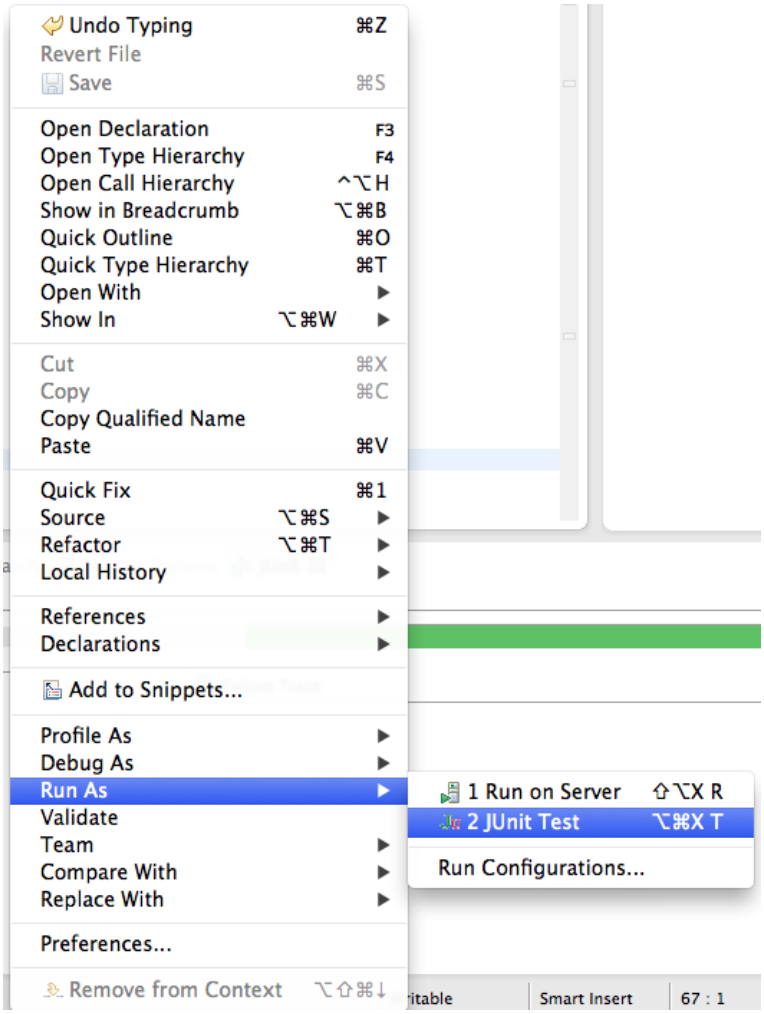
```
view plain print ?
/*

    src/test/java/libreria/Tests.java

*/

//Con RunWith indicamos que sea Spring quien gestione los test y nos provea de funcionalidades de Spring como la DI y
//En contextConfiguration ponemos la ruta donde está nuestro xml con la configuración que necesitamos.
//Es un XML distinto debido a que no nos hace falta levantar todo lo que habíamos levantado en la aplicación.
//Más adelante pondré el XML servlet-context-test.xml.
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:spring/servlet-context-test.xml" })
public class Tests {

    @Resource
    private Dao dao;

    List<Libro> libros;

    Libro libro;

    private static final int LIBROSBD = 3;

    //Con before estamos configurando que este método sea invocado antes de los test
    //Aquí pondremos la inicialización de las instancias que necesitemos en los test.
    //Como ejemplo vamos a creamos un Libro.
    @Before
    public void setUp() {
        libro = new Libro();
        libro.setId(5);
        libro.setAutor("Carlos");
        libro.setTitulo("HTML");
        libro.setIsbn("5555");
    }


    //Empezamos con los test. cada test tiene que tener la anotación de Test
    //En este primer Test probamos que haya 3 libros en la base de datos, que son los libros que introducimos con el scrip
    @Test
    public void testBBDD() {
        libros = dao.findAll();
        Assert.assertTrue("Comprobar que en la base de datos hay 3 libros",
                libros.size() == LIBROSBD);
    }

    //En este test probamos a introducir un libro y le volvemos a sacar de la base de datos para comprobar que es correcto
    @Test
    public void testInsert() {

        dao.insertBook(libro);
        libros = dao.findAll();
        Assert.assertTrue("Comprueba que hay un libro mas",
                libros.size() == LIBROSBD + 1);

        Libro libroAux = libros.get(LIBROSBD);

        Assert.assertTrue("Comprueba que coincide id.",
                libro.getId() == libroAux.getId());
        Assert.assertTrue("Comprueba que coincide el autor.",
                libro.getAutor() == libroAux.getAutor());
        Assert.assertTrue("Comprueba que coincide el titulo.",
                libro.getTitulo() == libroAux.getTitulo());
        Assert.assertTrue("Comprueba que coincide el ISBN.",
                libro.getIsbn() == libroAux.getIsbn());

        dao.deleteBook(libroAux);
    }

    //En este test comprobamos que se borra de la base de datos correctamente.
    @Test
    public void testDelete() {

        dao.insertBook(libro);

        libros = dao.findAll();
        Assert.assertTrue("Comprueba que hay un libro mas.",
                libros.size() == LIBROSBD + 1);

        dao.deleteBook(libro);

        libros = dao.findAll();
        Assert.assertTrue("Comprueba que vuelve a haber los libros iniciales.",
                libros.size() == LIBROSBD);
    }
}
```

Ahora vamos con el XML que hemos reducido para levantar únicamente lo que necesitamos para los test.

```
view plain print ?
<!--

src/test/resources/spring/servlet-context-test.xml

 -->

lt;beans xmlns="http://www.springframework.org/schema/beans"
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" xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:jdbc="http://www.springframework.org/schema/jdbc"
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.2.xsd
  http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd">


<!-- Escanea el classpath buscando anotaciones (eg: @Service, @Repository etc) -->
<context:component-scan base-package="com.autentia.libreria.*" />


<jdbc:embedded-database id="dataSource" type="HSQL">
   <jdbc:script location="classpath:persistence/libros.sql" />
</jdbc:embedded-database>

<!-- Hibernate Session Factory -->
<bean id="mySessionFactory"
   class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
   <property name="dataSource" ref="dataSource" />
   <property name="packagesToScan">
      <array>
         <value>com.autentia.libreria</value>
      </array>
   </property>
   <property name="hibernateProperties">
      <value>
         hibernate.dialect=org.hibernate.dialect.HSQLDialect
      </value>
   </property>
</bean>


<!-- Hibernate Transaction Manager -->
<bean id="transactionManager"
   class="org.springframework.orm.hibernate4.HibernateTransactionManager">
   <property name="sessionFactory" ref="mySessionFactory" />
</bean>

<!-- Activates annotation based transaction management -->
<tx:annotation-driven transaction-manager="transactionManager" />


lt;/beans>
```

Como podemos ver hemos quitado la parte de MVC y aspectos ya que no se van a probar en los Test.

Sólo queda decir que para ejecutarlo, por ejemplo, se pulsa en el botón derecho sobre la clase y la ruta : RunAs --> JUnit Test, y vemos que los tres test pasan correctamente. En la siguiente imagen lo muestro.



Conclusiones.

El objetivo del tutorial es mostrar un pequeño ejemplo de todo el sistema mvc, con los spring e hibernate como base, y añadiendo aspectos y tests, para que sea un punto de partida para hacer cosas más interesantes.

Dejo el código del proyecto aquí para que lo podáis descargar.

Para cualquier duda o aclaración, en los comentarios.

Un saludo.