Librería:
Spring MVC
Tiles 3
Hibernate

Marco Antonio Fernández Heras
2º Desarrollo de aplicaciones multiplataforma
I.E.S Arroyo Harnina
2016

Tabla de contenidos

1. Requisitos previos.
1.1 Herramientas.

I. JAVA IDE.

Para el desarrollo de nuestro código java vamos a utilizar IntelliJ IDEA en su versión Ultimate como editor y depurador.
Lo podemos descargar desde la web del fabricante:
https://www.jetbrains.com/idea/download/
Usaremos la versión Ultimate por su integración con los diversos frameworks que vamos a utilizar.

II. Servlet Container

Como contenedor de servlets ("*Servidor web*" a partir de aquí) usaremos Apache Tomcat 8, que podemos descargar desde la web del fabricante:
http://tomcat.apache.org/download-80.cgi

III. RDBMS

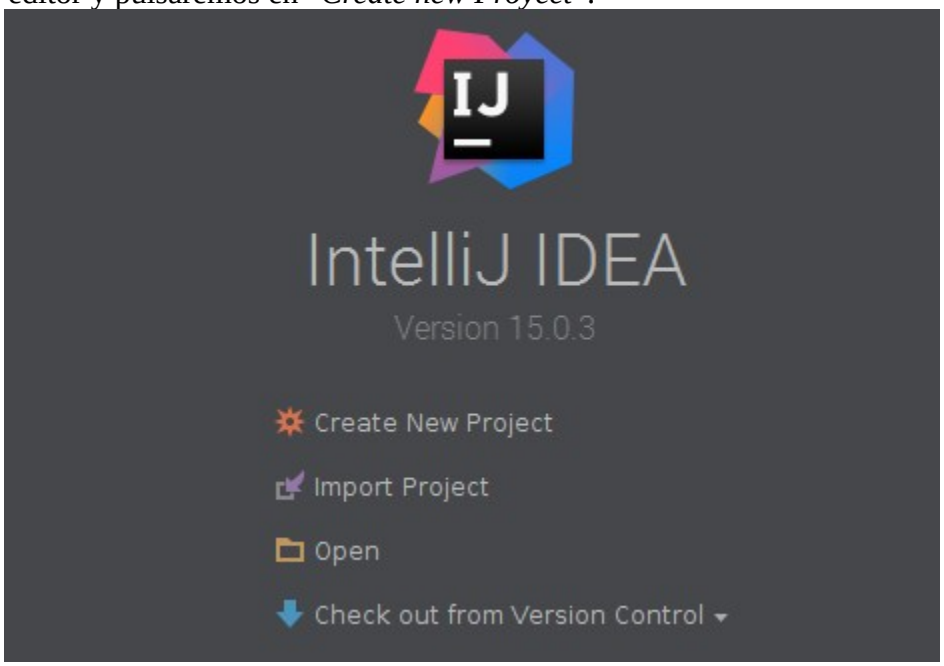
Como sistema de manejo de bases de datos relacionales ("*base de datos*" a partir de aquí) usaremos MySql o su reemplazo open source MariaDB.
MySql podemos descargarlo desde http://dev.mysql.com/downloads/mysql/.
MariaDB podemos descargarlo desde https://downloads.mariadb.org/.
Tambien podemos optar por utilizar XAMPP, el cual trae MariaDB.

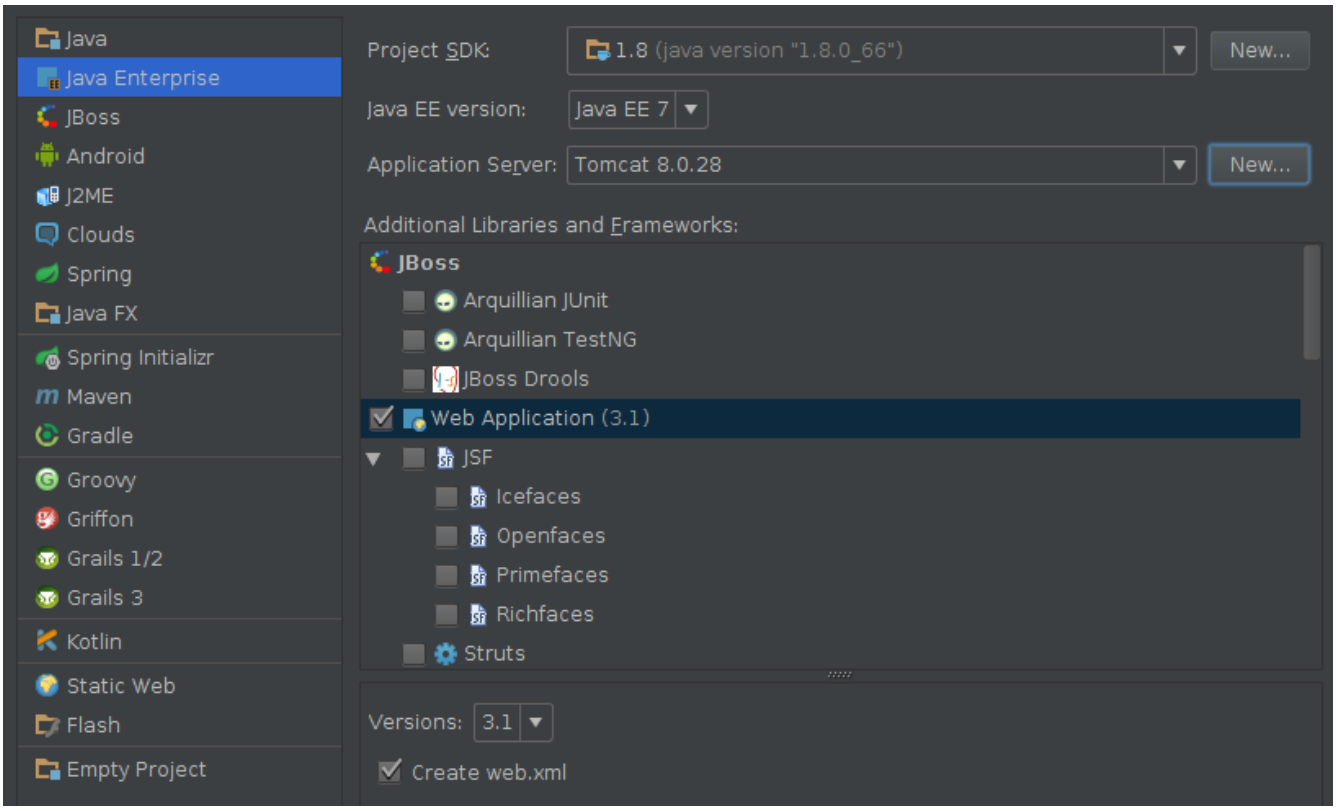
IV. SQL IDE.

Para la gestión y visualización de la base de datos durante el proceso de desarrollo, usaremos HeidiSql como editor.
Podemos descargarlo desde http://www.heidisql.com/download.phpmysql:mysql-connector-java:5.1.36

1.2 Creando el proyecto.

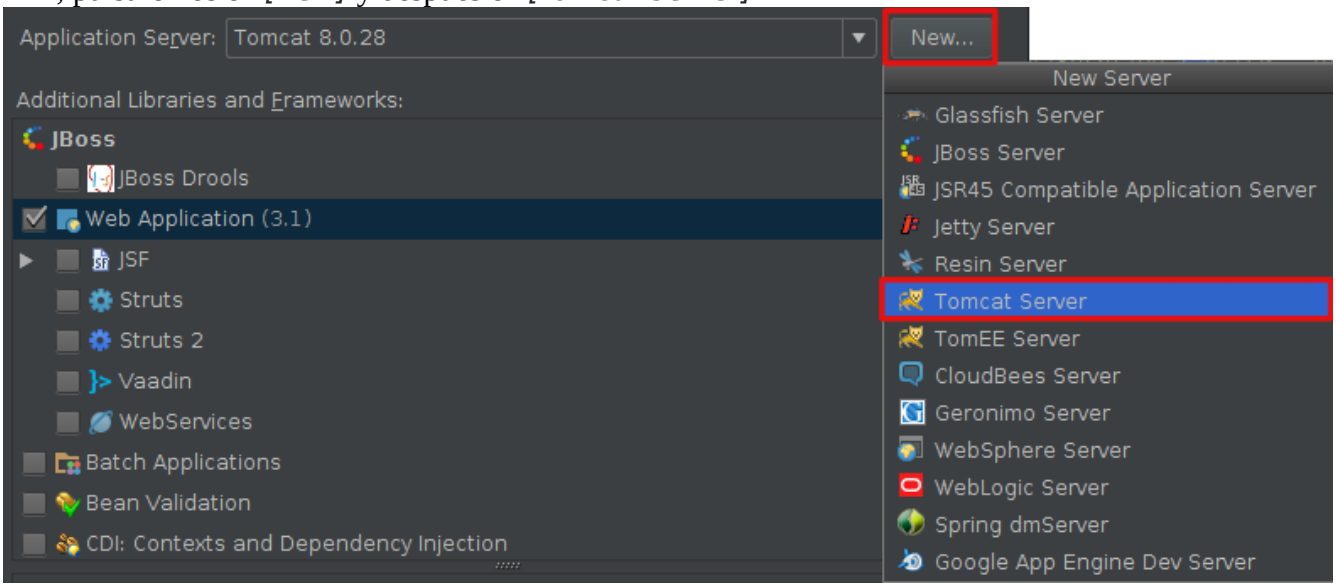
I. Abriremos el editor y pulsaremos en "*Create new Proyect*":



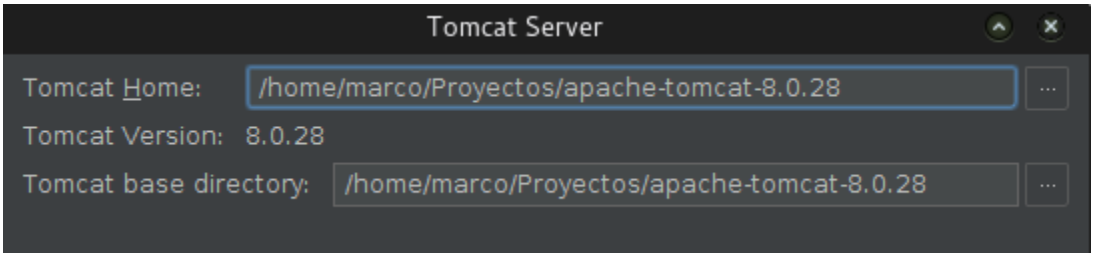


II. (Opcional): Configurar el *Java Development Kit* (JDK): Si no tenemos el JDK todavía configurado, pincharemos en [*New*] y después en *[JDK].* En a ventana emergente seleccionaremos la ruta de instalacion de nuestro JDK local.

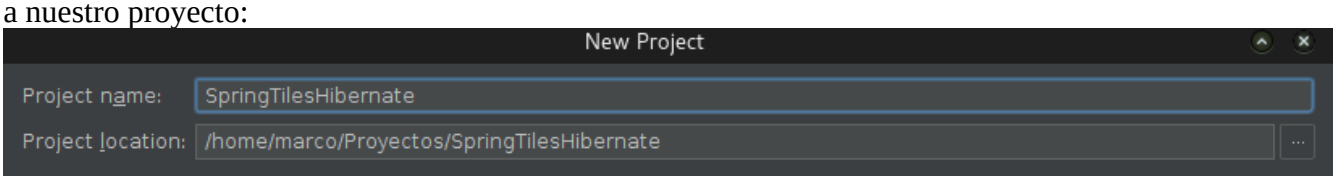
III. (Opcional): Configurar el *Servidor web:* Si no hemos configurado nuestro *Servidor web* en el IDE, pulsaremos en *[New]* y despues en *[Tomcat Server]:*



Y especificaremos la ruta de instalacion:



IV. Después de de configurar el JDK y el Servidor Web, pulsaremos en siguiente y le daremos nombre a nuestro proyecto:



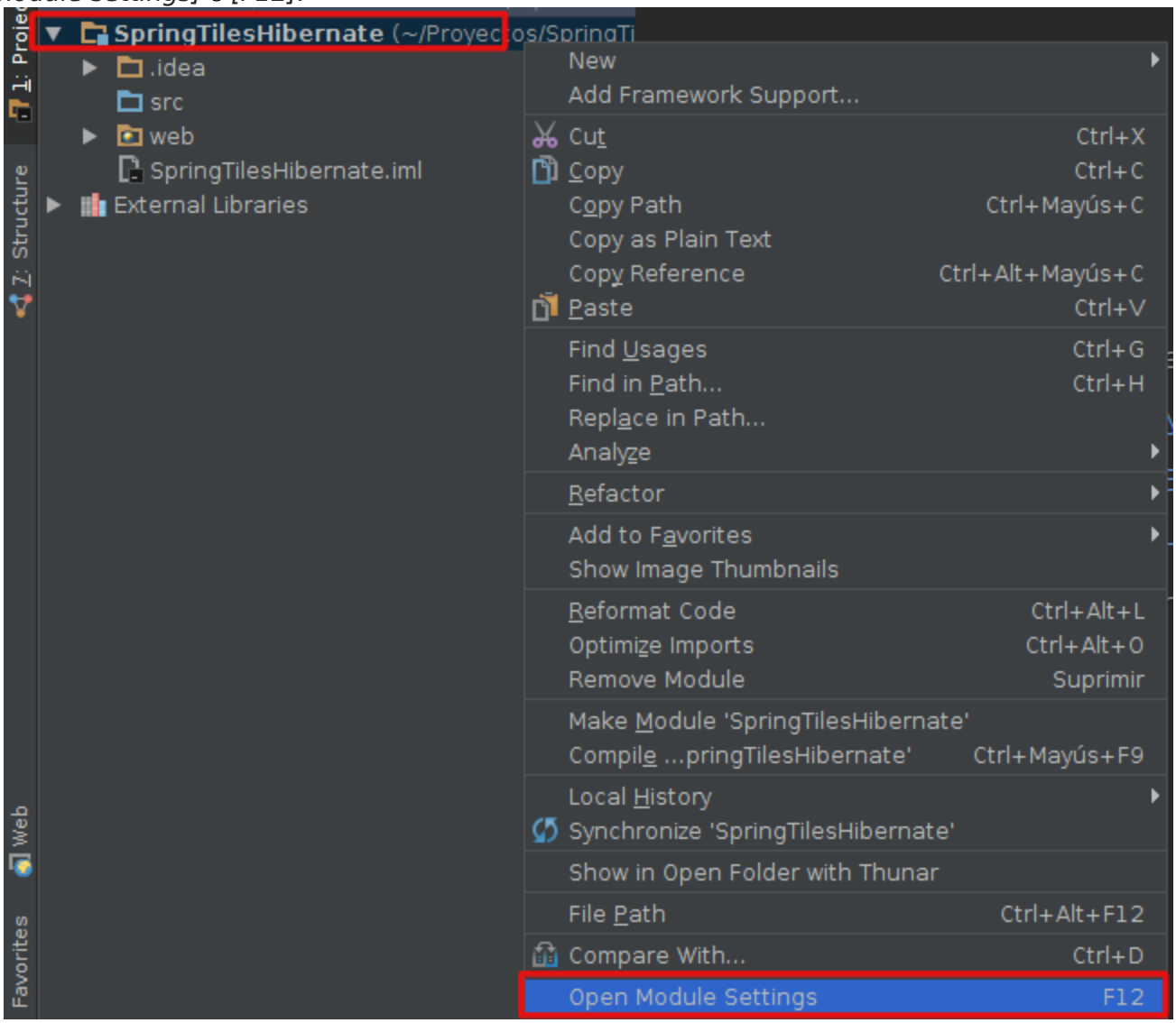
y pulsaremos en *[Finish]*.

2. Dependencias.

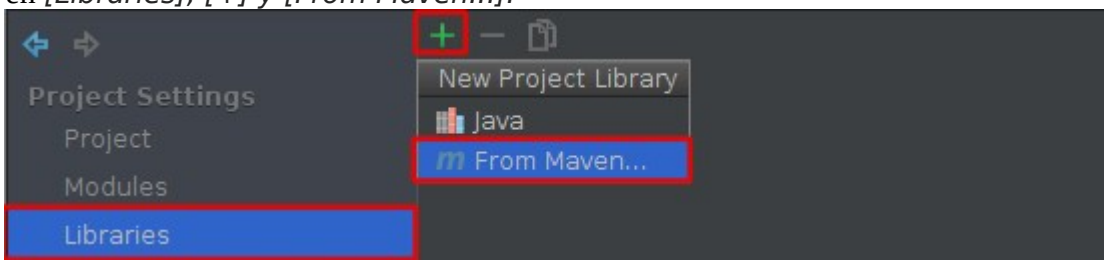
Una vez que tenemos nuestro proyecto listo, pasaremos a instalar y configurar las dependencias necesarias para poder utilizar Spring y Tiles3. Pospondremos la instalación y configuración de Hibernate para capítulos posteriores, una vez tengamos la capa de presentación operativa.

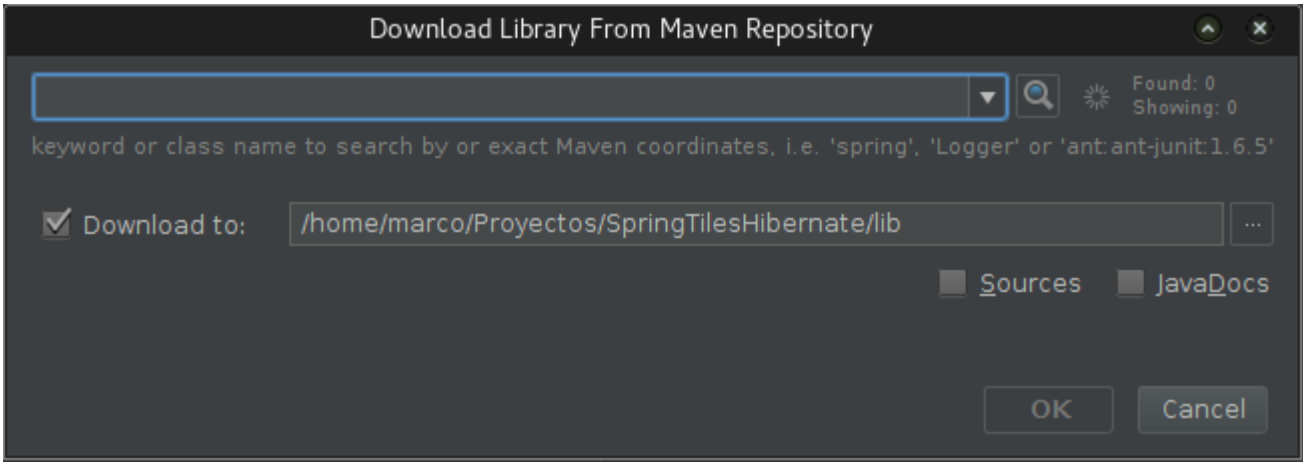
2.1 Instalando librerías.

I. Pincharemos sobre la carpeta de proyecto con el *[Botón derecho de ratón]* y después en *[Open module settings]* o *[F12].*



Después en *[Libraries], [+] y [From Maven...]:*





En la ventana emergente iremos buscando y descargando las siguientes librerias:
Para Spring:

- org.springframework:spring-context:4.1.6.RELEASE
- org.springframework:spring-context-support:4.1.6.RELEASE
- org.springframework:spring-webmvc:4.1.6.RELEASE
- org.springframework:spring-web:4.1.6.RELEASE
- org.springframework:spring-beans:4.1.6.RELEASE
- org.springframework:spring-orm:4.1.6.RELEASE

Para Tiles:

- org.apache.tiles:tiles-jsp:3.0.3
- org.apache.tiles:tiles-core:3.0.3
- org.apache.tiles:tiles-api:3.0.3
- org.apache.tiles:tiles-servlet:3.0.3
- org.apache.tiles:tiles-template:3.0.3

Loging:

- org.slf4j:slf4j-api:1.6.6
- org.slf4j:jcl-over-slf4j:1.6.6
- org.slf4j:slf4j-log4j12:1.6.6
- log4j:log4j:1.2.15

Servlet:

- javax.servlet:javax.servlet-api:3.1.0
- javax.servlet.jsp:jsp-api:2.1
- javax.servlet:jstl:1.2

Mysql:

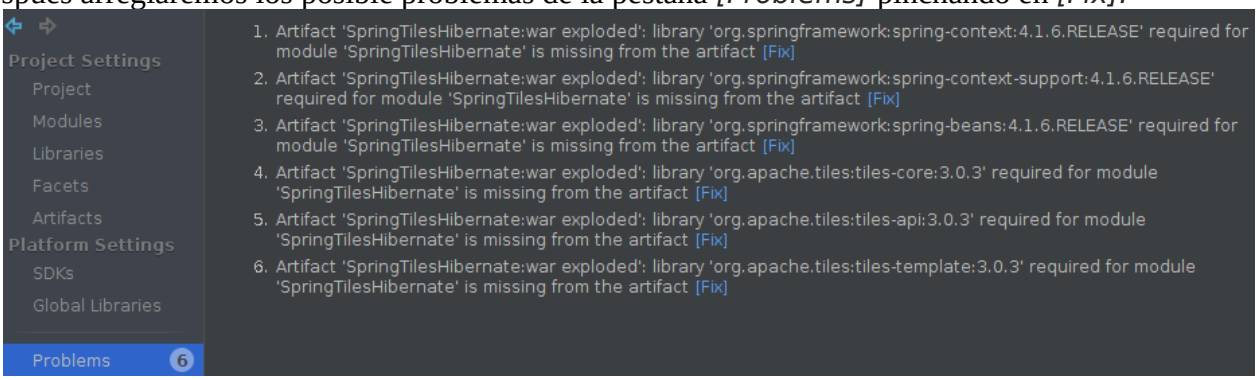
- mysql:mysql-connector-java:5.1.36
-

Una vez descargadas todas, debemos borrar las siguientes:

- commons-logging-1.2.jar
- jcl-over-slf4j-1.5.8.jar
- slf4j-api-1.5.8.jar.

Puesto que entran en conflicto con sus versiones mas modernas.

Después arreglaremos los posible problemas de la pestaña *[Problems]* pinchando en *[Fix]*.



2.2 Configuración inicial.

Una vez las librerías están descargadas, pasaremos a la configuración inicial de Spring en *web.xml*:

```
1  <!-- The definition of the Root Spring Container shared by all Servlets -->
2  <context-param>
3      <param-name>contextConfigLocation</param-name>
4      <param-value>/WEB-INF/spring/root-context.xml</param-value>
5  </context-param>
6  <!-- Creates the Spring Container shared by all Servlets and Filters -->
7  <listener>
8        <listener-class>
9              org.springframework.web.context.ContextLoaderListener
10       </listener-class>
11 </listener>
12 <!-- Processes application requests -->
13 <servlet>
14     <servlet-name>appServlet</servlet-name>
```

```
15      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
16      <init-param>
17          <param-name>contextConfigLocation</param-name>
18          <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
19      </init-param>
20      <load-on-startup>1</load-on-startup>
21  </servlet>
22  <!-- Map all *.form urls to Spring servlet -->
23  <servlet-mapping>
24      <servlet-name>appServlet</servlet-name>
25      <url-pattern>*.form</url-pattern>
26  </servlet-mapping>
```

Líneas 2-5: Definimos el archivo de configuración global de Spring (que crearemos después).
Líneas 7-11: Configuramos el arranque del contenedor de Spring para todos los servlets y filtros.
Líneas 13-21: Declaramos el servlet de Spring MVC y configuramos donde se encontrará el archivo de configuración (que crearemos después).
Líneas 23-26: Mapeamos todas las rutas terminadas en .form al servlet de Spring MVC.

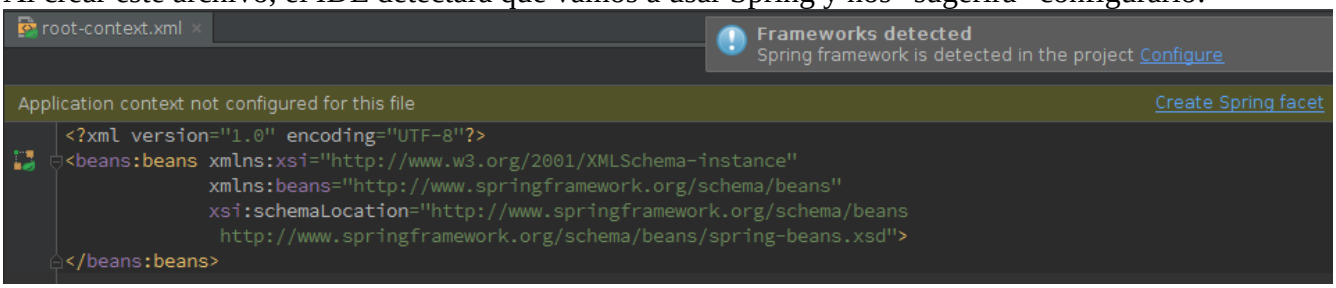
Para evitar futuros problemas de codificación de caracteres UTF-8 (acentos, eñes, etc), usaremos un filtro de Spring para forzar la codificación:

```
 1  <!-- UTF-8 Encoding Filter -->
 2  <filter>
 3      <filter-name>EncodingFilter</filter-name>
 4      <filter-class>
 5  org.springframework.web.filter.CharacterEncodingFilter
 6      </filter-class>
 7      <init-param>
 8          <param-name>encoding</param-name>
 9          <param-value>UTF-8</param-value>
10      </init-param>
11      <init-param>
12          <param-name>forceEncoding</param-name>
13          <param-value>true</param-value>
14      </init-param>
15  </filter>
16
17  <!-- Encoding filter mapping -->
18  <filter-mapping>
19      <filter-name>EncodingFilter</filter-name>
20      <url-pattern>*</url-pattern>
21  </filter-mapping>
```

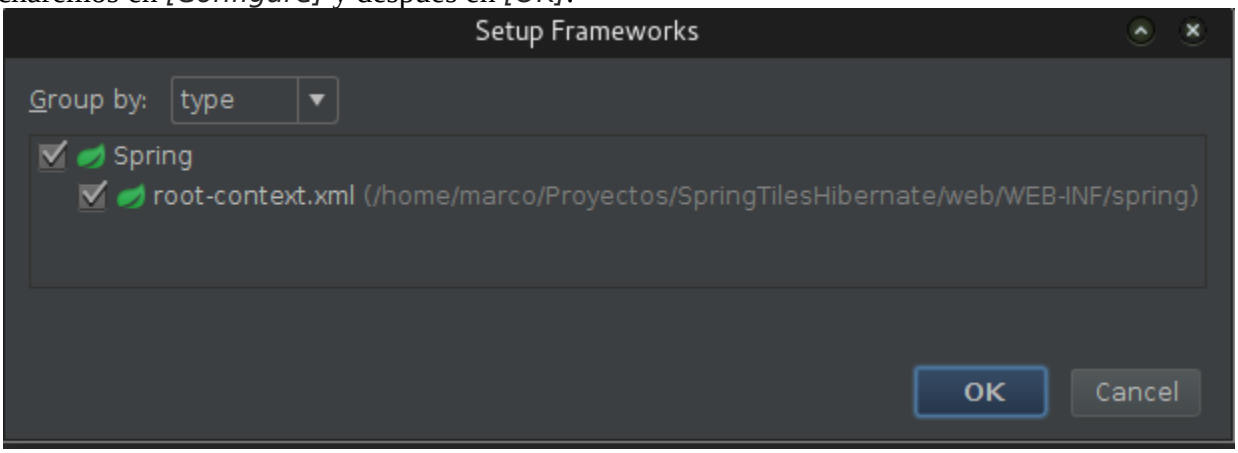
Creamos a continuación el archivo *root-context.xml* dentro de */WEB-INF/spring/* :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3              xmlns:beans="http://www.springframework.org/schema/beans"
4              xsi:schemaLocation="http://www.springframework.org/schema/beans
5               http://www.springframework.org/schema/beans/spring-beans.xsd">
6 </beans:beans>
```

Al crear este archivo, el IDE detectará que vamos a usar Spring y nos “sugerirá” configurarlo:



Pincharemos en *[Configure]* y después en *[OK]*:



Pasamos a configurar el servlet de Spring MVC en el archivo *servlet-context.xml* dentro de *WEB-INF/spring/appServlet*:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:mvc="http://www.springframework.org/schema/mvc"
4        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5        xmlns:context="http://www.springframework.org/schema/context"
```

```
6         xmlns:tx="http://www.springframework.org/schema/tx"
7         xsi:schemaLocation="
8          http://www.springframework.org/schema/beans
9          http://www.springframework.org/schema/beans/spring-beans.xsd
10         http://www.springframework.org/schema/mvc
11         http://www.springframework.org/schema/mvc/spring-mvc.xsd
12         http://www.springframework.org/schema/tx
13         http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
14         http://www.springframework.org/schema/context
15         http://www.springframework.org/schema/context/spring-context.xsd">
16
17     <!-- Configure MVC to use annotations -->
18     <mvc:annotation-driven />
19
20     <!-- Resolves views selected for rendering
21         by @Controllers to .jsp resources in the /WEB-INF/views directory -->
22     <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
23         <property name="prefix" value="/WEB-INF/views/" />
24         <property name="suffix" value=".jsp" />
25     </bean>
26
27     <!-- Tiles configuration -->
28     <bean id="tilesConfigurer"
29           class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
30         <property name="definitions">
31             <list>
32                 <value>/WEB-INF/tiles/tiles-definitions.xml</value>
33             </list>
34         </property>
35     </bean>
36 </beans>
```

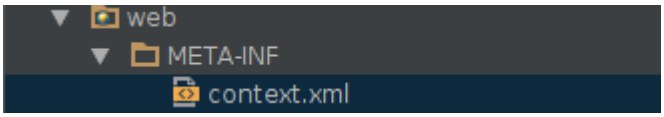
Línea 18: Configuramos el servlet para que use *configuración por anotaciones*.
Líneas 22-25: Configuramos las vistas, especificando que estarán dentro de *WEB-INF/views/* y que tendrán extensión *jsp*.
Líneas 28-35: Declaramos donde se encontrará el archivo de definiciones que usará Tiles3 para componer nuestras vistas.

Por último crearemos el archivo *tiles-definitions.xml* dentro de *WEB-INF/tiles*:

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <!DOCTYPE tiles-definitions PUBLIC
3         "-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
4         "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
5 <tiles-definitions>
6     <definition name="defaultTemplate"
7                 template="/WEB-INF/template/default/template.jsp">
8     </definition>
9 </tiles-definitions>
```

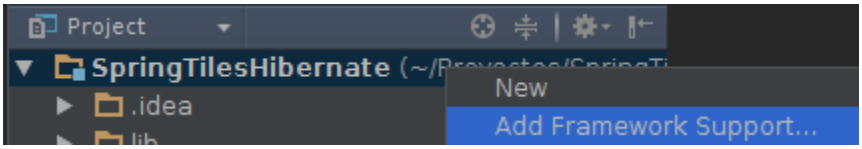
Líneas 6-7: Definimos donde se encuentran las plantillas que serán la base de nuestras vistas.

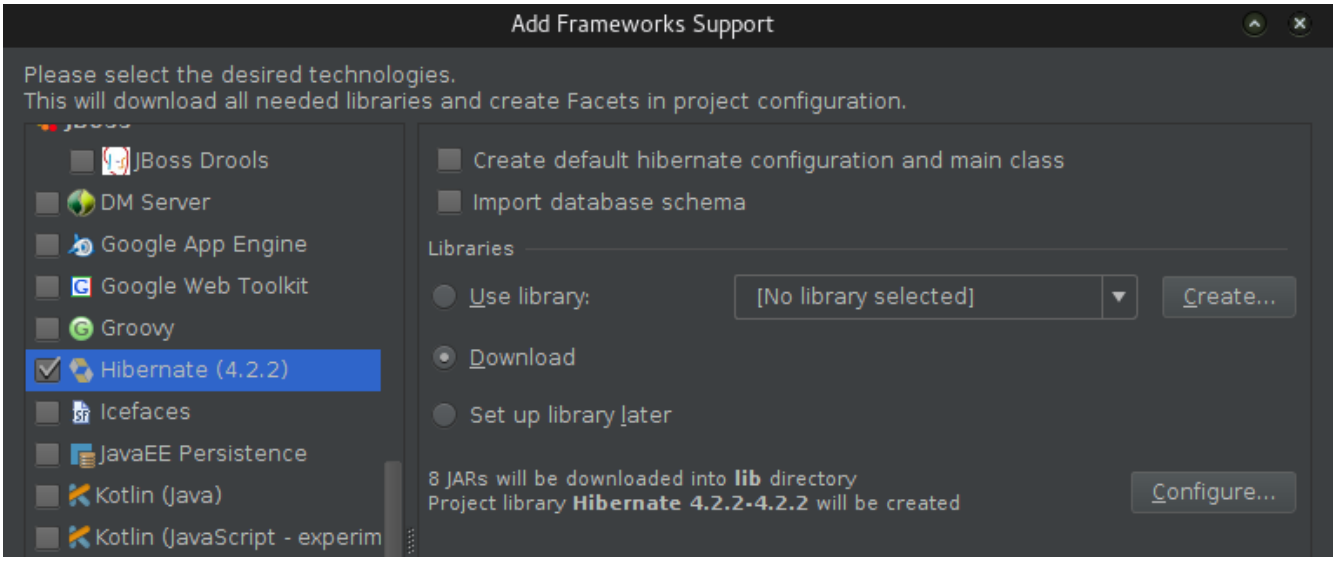
Pasamos ahora a instalar el soporte de *Hibernate* a nuestro proyecto.
Primero configuraremos el pool de conexiones en */META-INF/context.xml:*




```
 1 <Context>
 2     <Resource name="jdbc/PoolDB" auth="Container" type="javax.sql.DataSource"
 3               factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
 4               testWhileIdle="true" testOnBorrow="true" testOnReturn="false"
 5               validationQuery="SELECT 1" validationInterval="30000"
 6               timeBetweenEvictionRunsMillis="30000" maxActive="100"
 7               minIdle="10" maxWait="10000" initialSize="10"
 8               removeAbandonedTimeout="60" removeAbandoned="true"
 9               logAbandoned="true" minEvictableIdleTimeMillis="30000"
10               jmxEnabled="true"
11               jdbcInterceptors=
12               "org.apache.tomcat.jdbc.pool.interceptor.ConnectionState;
13                 org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer"
14               username="root"
15               password="root"
16               driverClassName="com.mysql.jdbc.Driver"
17               url="jdbc:mysql://localhost:3306/LibreriaSpring"/>
18 </Context>
```

Después añadiremos el soporte para *Hibernate*:

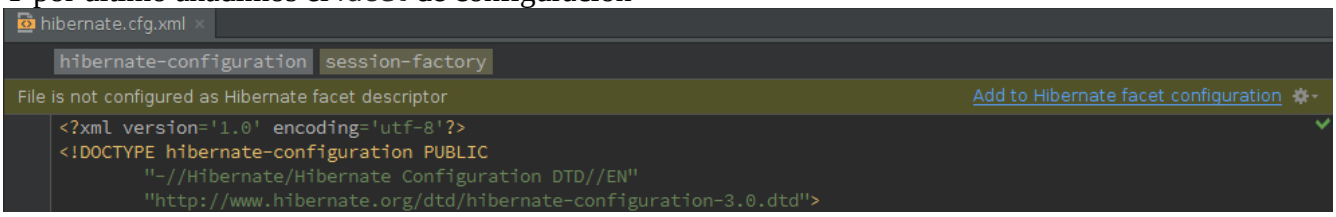




Crearemos el archivo de configuración de Hibernate en *src/hibernate.cfg.xml*

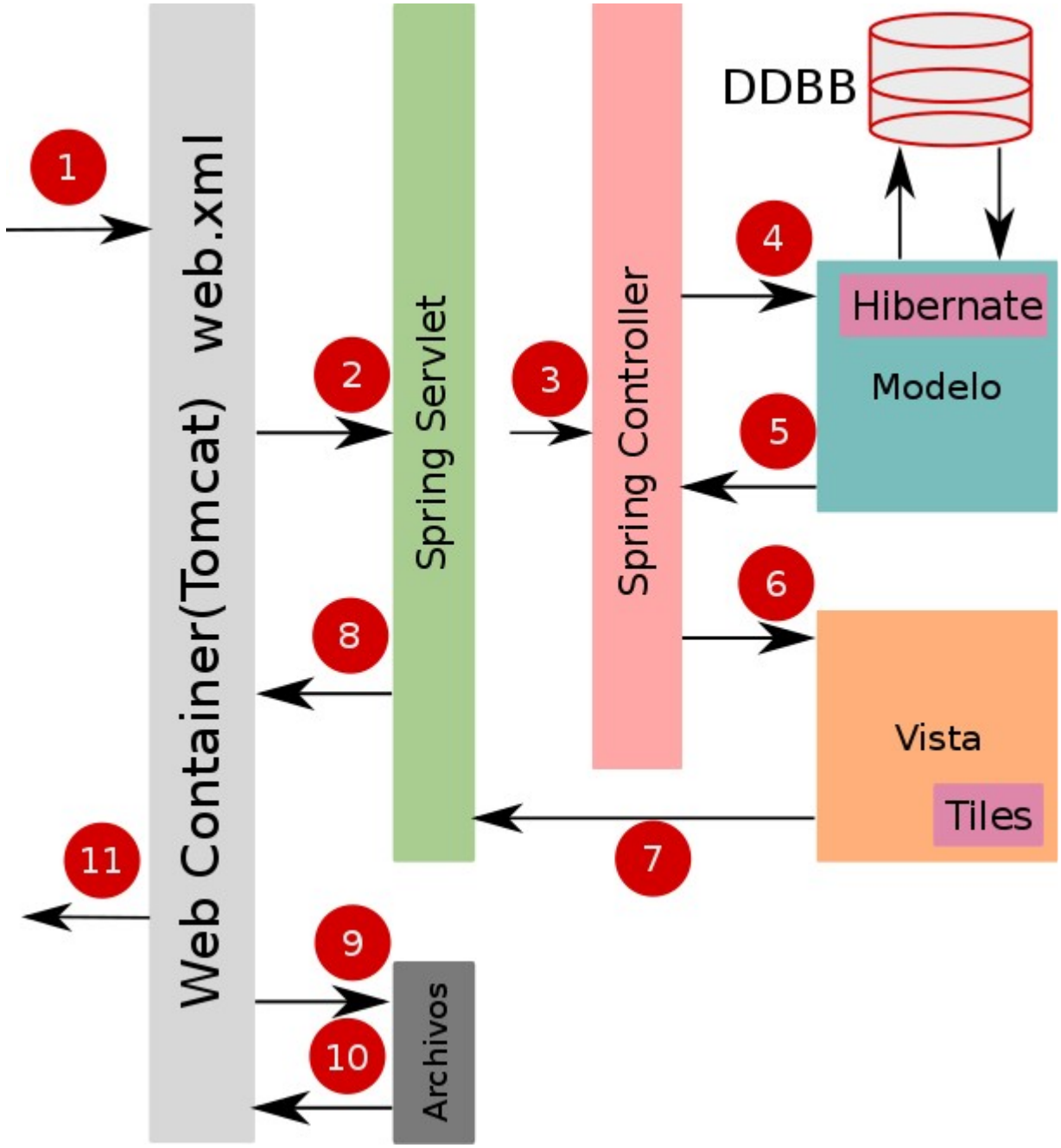
```
1  <?xml version='1.0' encoding='utf-8'?>
2  <!DOCTYPE hibernate-configuration PUBLIC
3          "-//Hibernate/Hibernate Configuration DTD//EN"
4          "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
5  <hibernate-configuration>
6      <session-factory>
7          <property name="connection.datasource">java:comp/env/jdbc/PoolDB</property>
8          <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9          <property name="show_sql">true</property>
10     </session-factory>
11 </hibernate-configuration>
```

Y por último añadimos el *facet* de configuración



3. Entendiendo la estructura.

3.1 Ciclo de vida de una petición.

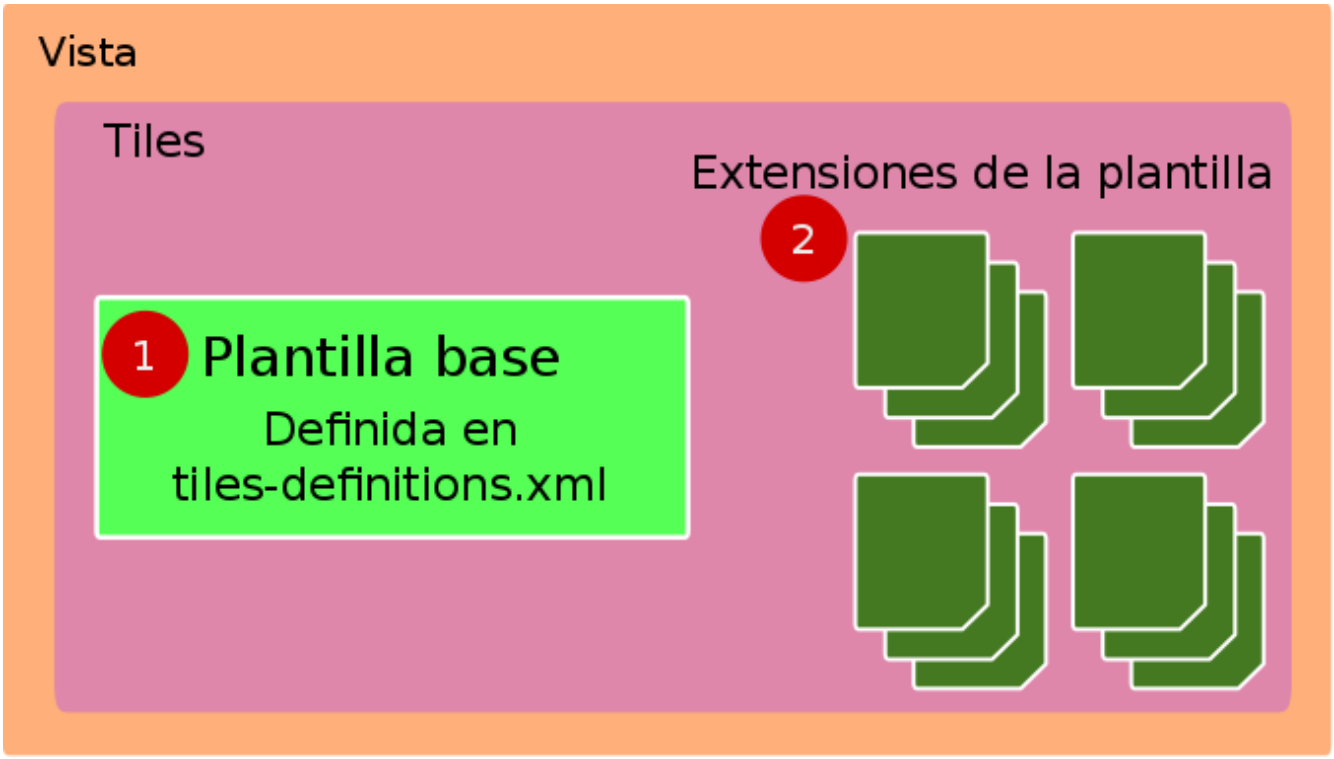


1.- La petición HTTP llega al servidor web desde el cliente.
2.- Si la petición acaba en .form es enrutada al servlet de Spring.
3.- El servlet de Spring mapea la petición al controlador pertinente.
4.- El controlador solicita al modelo los datos.
5.- El modelo, consulta a la base de datos usando hibernate, devuelve los datos al controlador.
6.- El controlador pasa los datos a la vista para generar el html.
7.- La vista, una vez generado el html la devuelve a al servlet de Spring.
8.- El servlet le devuelve el html al servidor web.

9.- Si la petición no acaba en .form es enrutada directamente a los ficheros dell sistema.
10.- El fichero se procesa y es devuelto al servidor web.

11.- El servidor web devuelve la respuesta al cliente.

3.2 Ciclo de vida de la vista.



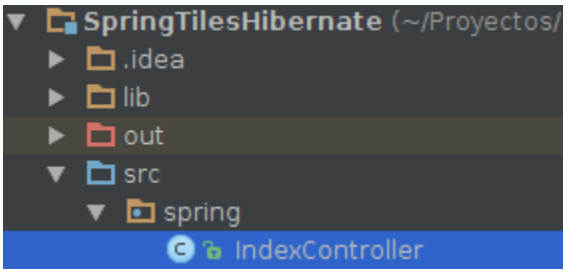
A la hora de crear las vistas, tendremos una plantilla base, definida en *tiles-definitions.xml* (1) donde vamos a definir las partes comunes de nuestra aplicación web. En esta plantilla definiremos también las zonas variables para habilitar que las extensiones definan estas zonas y muestren el contenido especifico.
Después definiremos la vista especifica (2) de cada parte de nuestra web, extendiendo la plantilla base, re definiendo las zonas variables.

4. Hello Spring.

En este capítulo, crearemos nuestro primer controlador, definiremos la plantilla base de nuestro sitio web y añadiremos la vista del index.

4.1 Nuestro primer controlador.

Empezaremos creando un nuevo paquete al que llamaremos *spring* y dentro de este crearemos una nueva clase *IndexController*:




```
1  package spring;
2
3  import org.springframework.stereotype.Controller;
4  import org.springframework.web.bind.annotation.RequestMapping;
5
6  @Controller
7  public class IndexController {
8
9      @RequestMapping(value = {"/index"})
10     public String index(){
11         return "index";
12     }
13 }
```

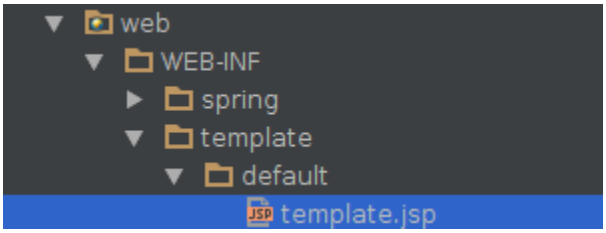
Línea 6: Anotación de Spring MVC para indicar que esta clase en un controlador.
Línea 9: Anotación de Spring MVC para indicar que este método responde a la rutas especificada.
Línea 10-12: Método que devuelve un String, que será la vista que deseamos devolver al cliente.

Para que Spring MVC encuentre este controlador, debemos indicar en *servlet-context.xml* cual es el package donde se encuentran nuestros controladores. Por lo tanto, añadiremos las siguientes lineas depués de *<mvc:annotation-driven />*:

```
1 <!-- Configure controller's package -->
2 <context:component-scan base-package="spring" />
```

4.2 La plantilla de nuestra web.

Una vez hemos definido nuestro primer controlador, debemos definir la vista *index* que especificamos como retorno. Pero para ello, primero definir la plantilla base de nuestras vistas, creando el archivo */WEB-INF/template/default/template.jsp* tal y como especificamos durante la fase de configuración en el archivo */WEB-INF/tiles/tiles-definitions.xml*:



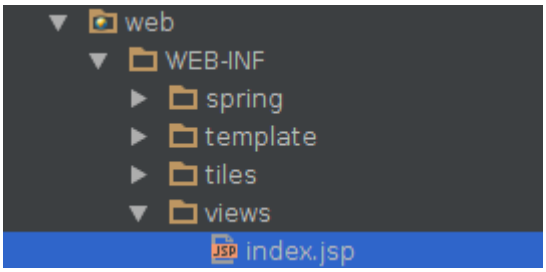
El contenido de la plantilla será el siguiente:

```
1  <%@ page contentType="text/html; charset=UTF-8"%>
2  <%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
3  <!DOCTYPE html>
4  <html lang="es">
5          <head>
6              <meta charset="UTF-8">
7              <title>
8                  <tiles:insertAttribute name="title"
9                  ignore="true" defaultValue="Spring Tiles"/>
10             </title>
11         </head>
12         <body>
13             <tiles:insertAttribute name="content" />
14         </body>
15 </html>
```

Línea 1: Definimos la codificación del archivo a *UTF-8* para poder usar acentos, eñes, etc.
Línea 2: Importamos la librería de etiquetas de Tiles3, usando como prefijo *tiles.*
Líneas 8-9: Definimos una zona dinámica opcional para el título, así cada vista podrá redefinirlo.
Líneas 13: Definimos una zona dinámica obligatoria para el contenido.

4.3 La vista index.

Ahora que tenemos la plantilla base de nuestra web, definiremos la vista *index* creando el archivo */WEB-INF/views/index.jsp*:



El contenido de la vista será el siguiente:

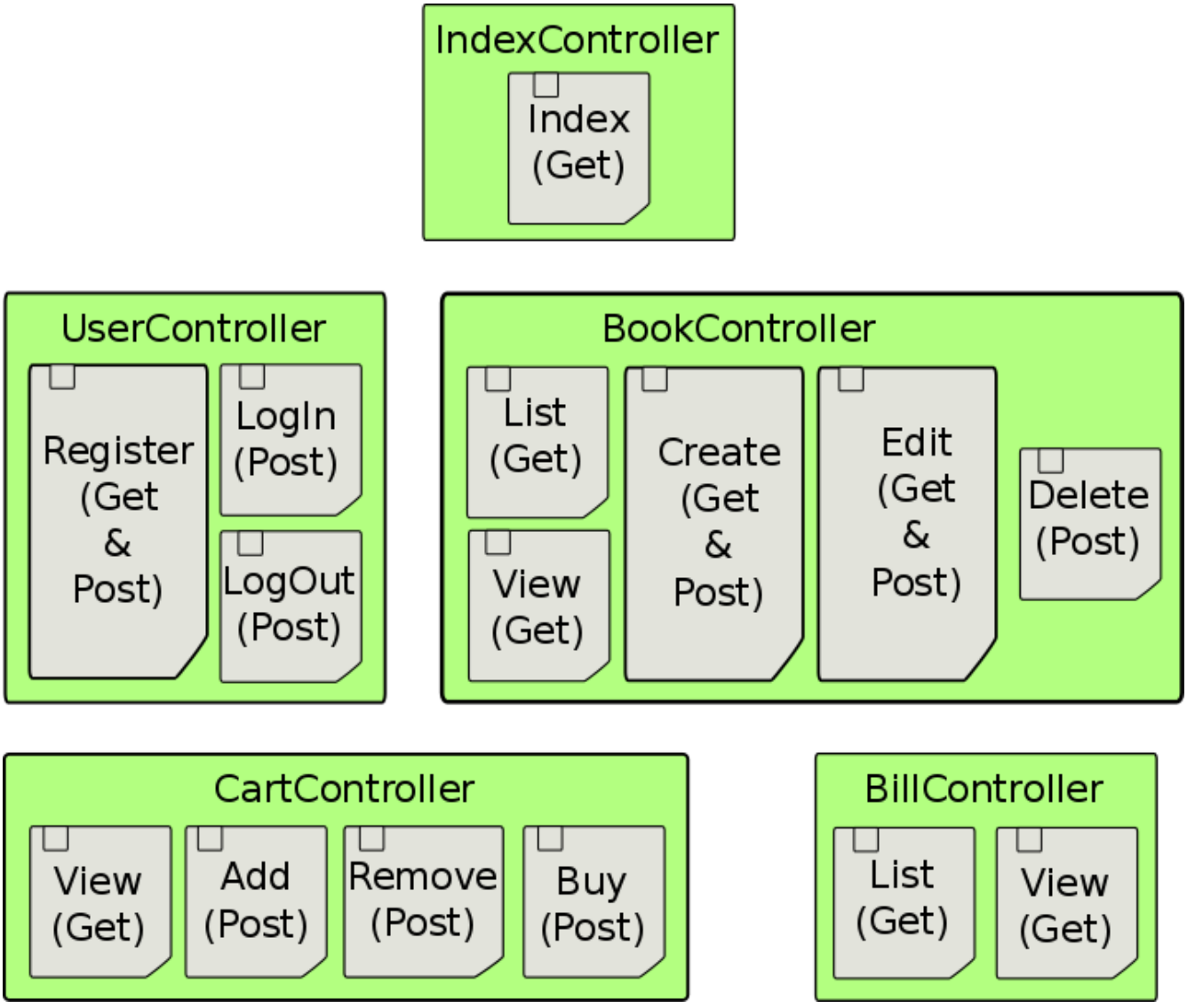
```
1  <%@ page contentType="text/html; charset=UTF-8"%>
2  <%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
3  <tiles:insertDefinition name="defaultTemplate">
4
5      <tiles:putAttribute name="title">Librería Spring - Tiles</tiles:putAttribute>
6
7      <tiles:putAttribute name="content">
8          Hello from Spring & Tiles3
9      </tiles:putAttribute>
10
11 </tiles:insertDefinition>
```

Línea 1: Definimos la codificación del archivo a *UTF-8* para poder usar acentos, eñes, etc.
Línea 2: Importamos la librería de etiquetas de Tiles3, usando como prefijo *tiles.*
Línea 3: Insertamos el contenido de la plantilla base.
Línea 5: Redefinimos la zona dinámica llamada *title*.
Línea 7-9: Redefinimos la zona dinámica llamada *content*.

Si ejecutamos y vamos a *http://localhost:8080/index.form* vemos como se renderiza la vista, incluyendo la plantilla base.

5. Estructurando nuestro proyecto.

5.1 La capa del controlador.



En la capa del controlador encontraremos 5 clases diferentes: *IndexController, UserController, BookController, CartController y BillController.*

IndexController:

Este será el controlador encargado de gestionar nuestra pagina de inicio. Contará con un único método *index* que responderá a la ruta */index.form (GET).*

UserController:

Este será el controlador de la gestión de usuario, su registro y su posterior acceso y salida del sistema. Contará con cuatro métodos:

1. *registerView:* Renderizará la vista de registro de usuarios.
   Responderá a la ruta */user/register.form (GET).*
2. *register:* Procesará los datos de registro de usuarios.
   Responderá a la ruta */user/register.form (POST).*
3. *login:* Verificará los datos proporcionados y proporcionará acceso al usuario si son correctos.
   Responderá a la ruta */user/login.form (POST).*
4. *logout:* Borrará los datos de sesión del usuario.
   Responderá a la ruta */user/logout.form (POST)*.

BookController:

Este controlador estará encargado del las operaciones CRUD (Create, Retrieve, Update, Delete) de los libros. Contará con los siguientes métodos:

1. *createView:* Renderizará la vista de registro de libros.
   Responderá a la ruta */book/create.form (GET).*
2. *create:* Procesará los datos de registro del nuevo libro.
   Responderá a la ruta */book/create.form (POST).*
3. *editView:* Renderizará la vista de edición de libros.
   Responderá a la ruta */book/edit.form (GET).*
4. *edit:* Procesará los datos de registro del nuevo libro.
   Responderá a la ruta */book/edit.form (POST).*
5. *list:* Renderizará una vista con la lista de libros registrados.
   Responderá a la ruta */book/list.form (GET).*
6. *view:* Renderizará una vista con los detalles de un libro concreto.
   Responderá a la ruta */book/view.form (GET).*
7. *delete:* Borrara un libro concreto.
   Responderá a la ruta */book/delete.form (POST).*

CartController:

Esta controlador gestionará el carro de la compra de cada sesión. Contará con cuatro métodos:

1. *view*: Renderizará la vista con el contenido actual del carro de la compra.
   Responderá a la ruta */cart/view.form (GET).*
2. *add:* Añadirá libros al carro de la compra.
   Responderá a la ruta */cart/add.form (Post).*

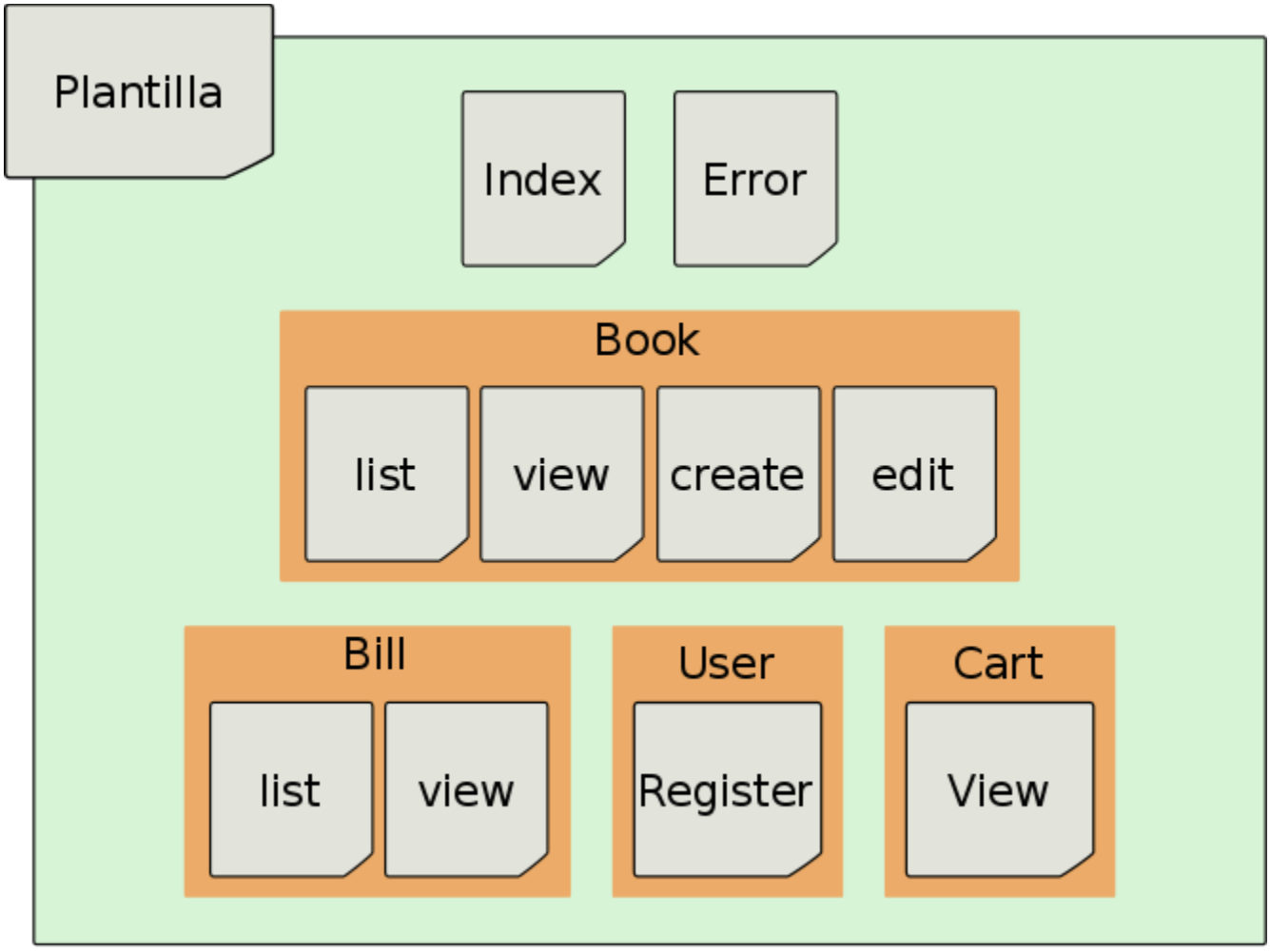
3. *remove:* Eliminará libros al carro de la compra.
   Responderá a la ruta */cart/remove.form (Post).*
4. *buy:* Procesará el carro de compra actuál y creará una factura.
   Responderá a la ruta */cart/buy.form (Post).*

BillController:

Esta controlador gestionará las facturas del usuario. Tendrá solo dos métodos:

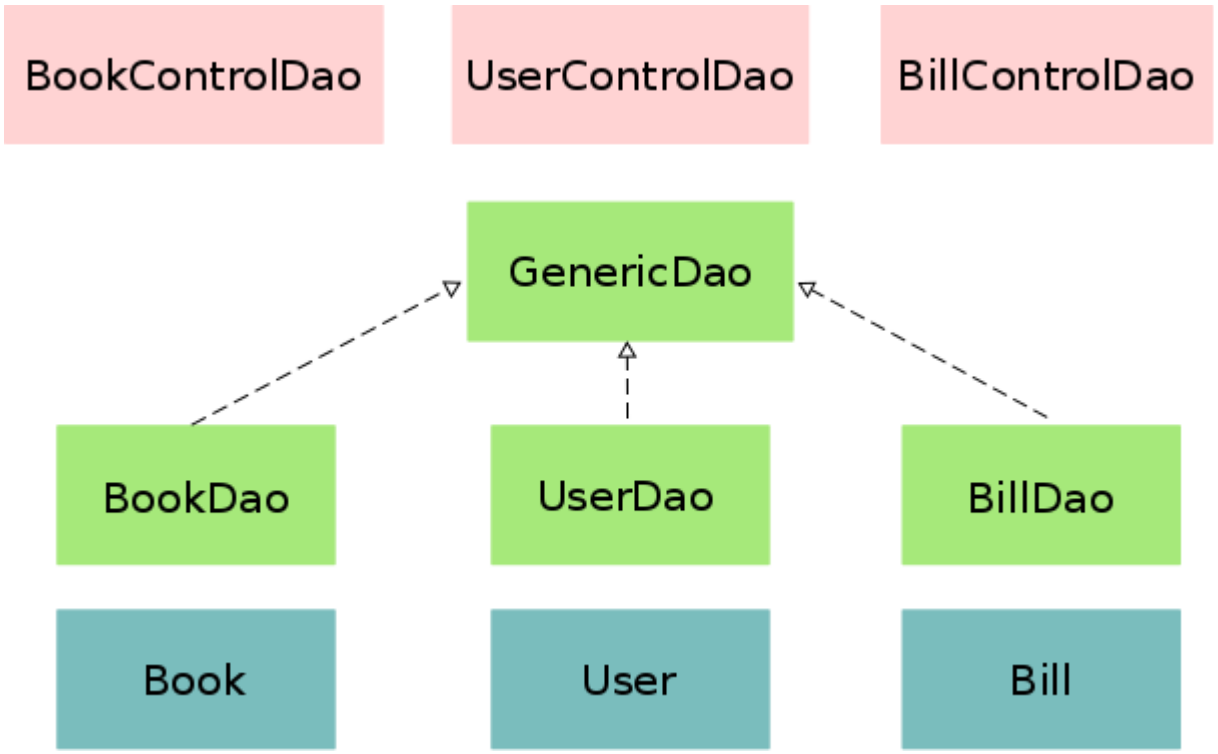
1. *list:* Mostrará la lista de facturas del cliente actual.
   Responderá a la ruta */bill/list.form (Post).*
2. *view:* Mostrará los detalles de una factura concreta.
   Responderá a la ruta */bill/view.form (Post).*

5.2 La capa de la vista.



En la capa de la vista tendremos una pagina *jsp* por cada petición *GET* a la que responde la capa de controlador.
Además tendremos una vista *error* a la que redireccionaremos en caso de error.

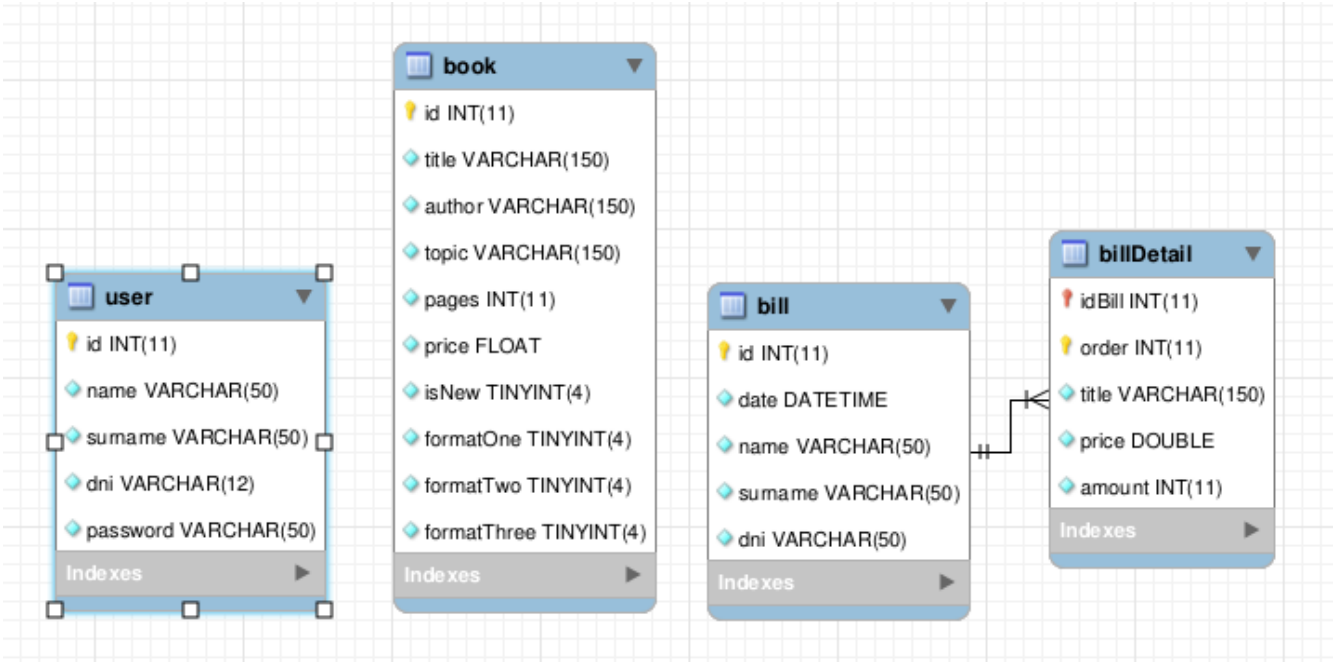
5.3 La capa del modelo.



En la capa del modelo, tendremos los beans de *Book, User, y Bill*. También tendremos los objetos de acceso a datos *BookDao, UserDao y BillDao*, que implementarán la interfaz *GenericDao*, y a modo de fachada las clases de control *BookControlDao, UserControlDao y BillControlDao*.

6. Creando la base de datos.

6.1 Diagrama entidad relación.

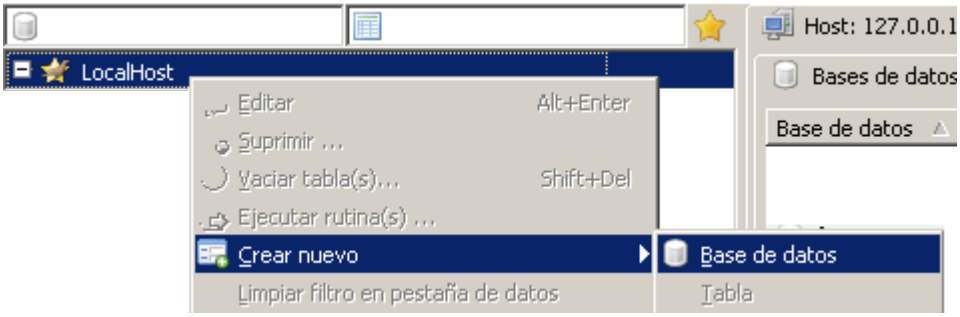


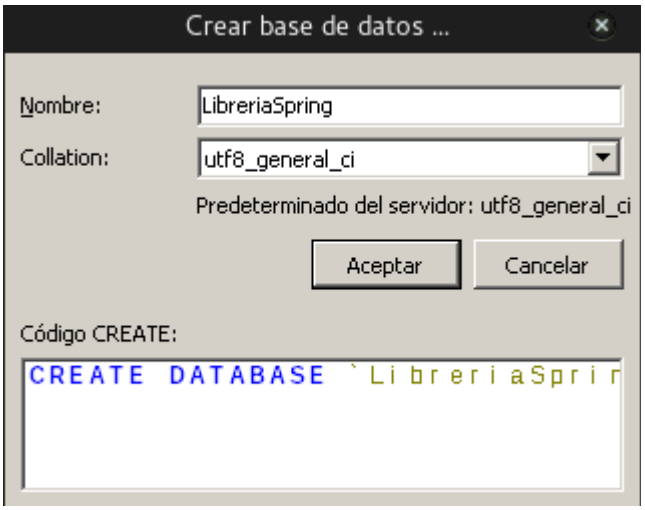
Observamos que tenemos las siguientes entidades:

- libro: Entidad que representa un libro. Tendrá un numero de id y un título únicos.
- usuario: Entidad que representa a los usuarios registratos en el sitio web. Tendrá un numero de id y un dni únicos.
- facturaCabecera y facturaDetalle: la entidad Factura la dividiremos en dos para evitar redundancia. Tendremos pues facturaCabecera que contendran los datos generales de la factura y por otro lado facturaDetalle que contendrá todos y cada uno de los libros que forman parte de la misma factura. Tendrán una relación de uno a muchos.

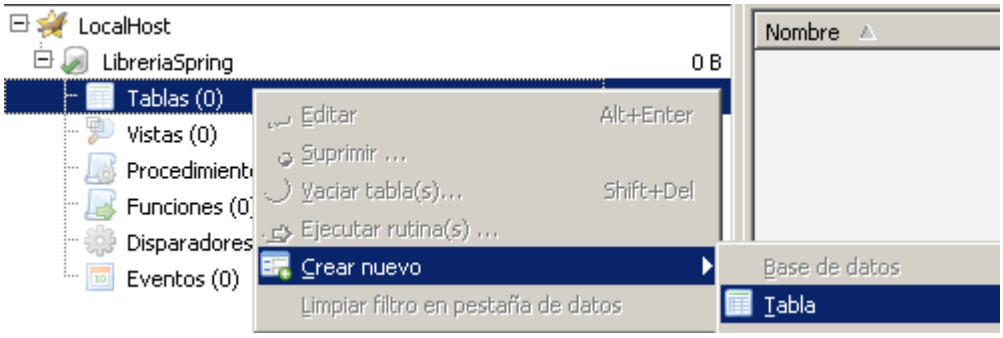
6.2 Implementación en MySql.

Con la base de datos en ejecución, ejecutaremos HeidiSql y procederemos a crear nuestra base de datos y sus tablas:

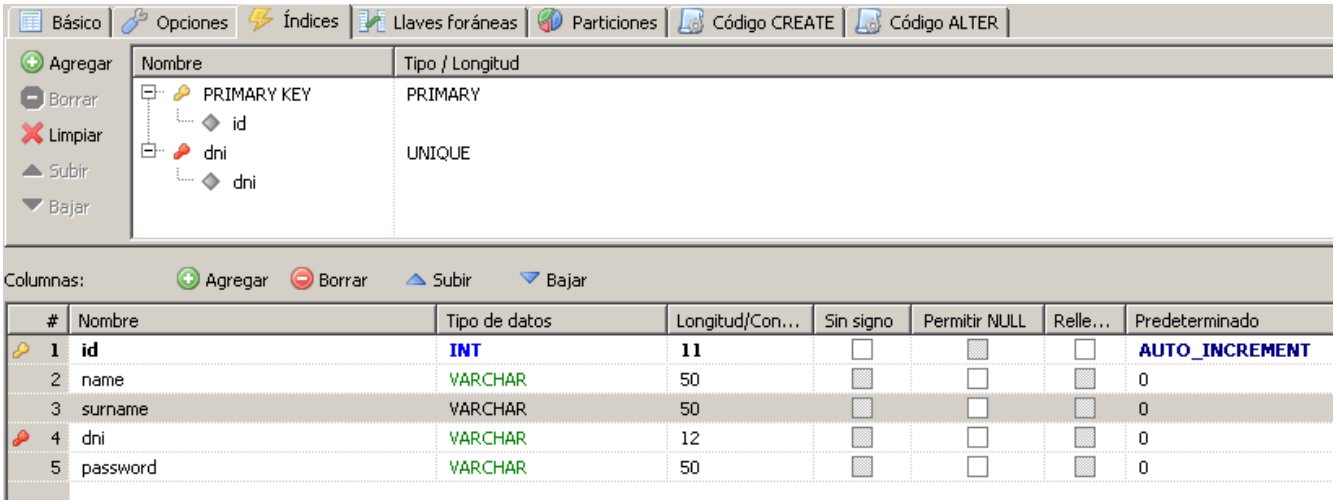




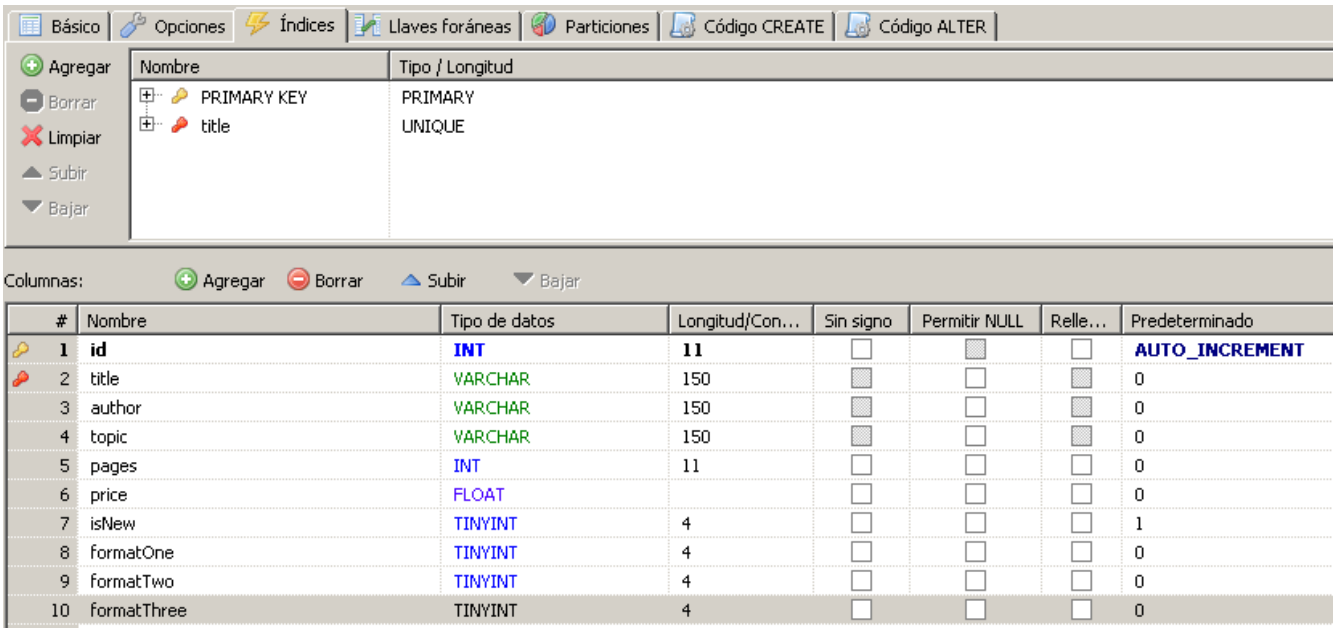
Y empezamos a crear nuestras tablas:



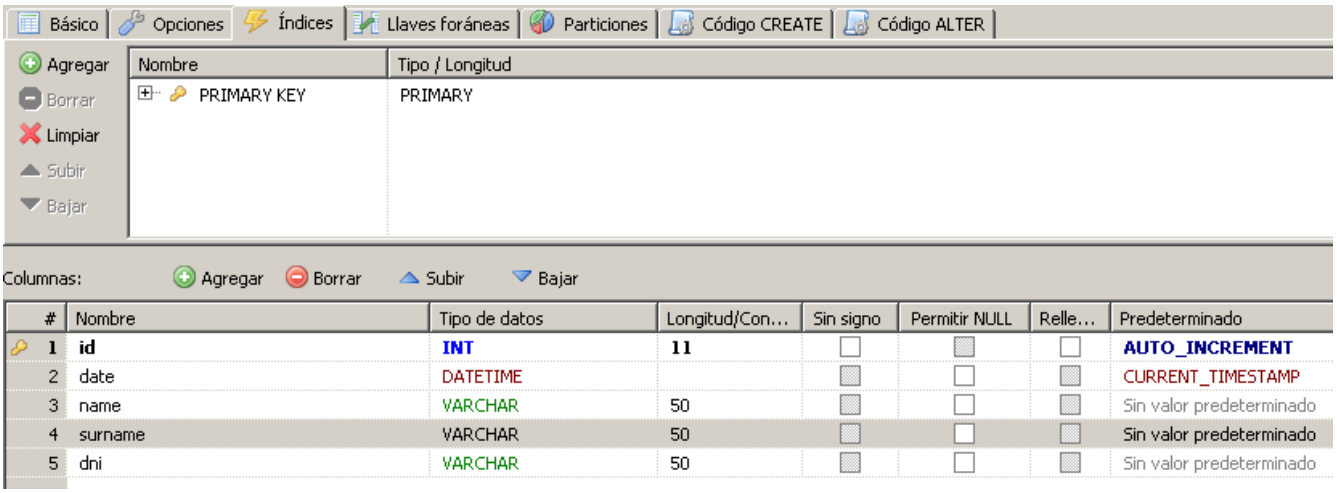
Creamos la tabla *user*:



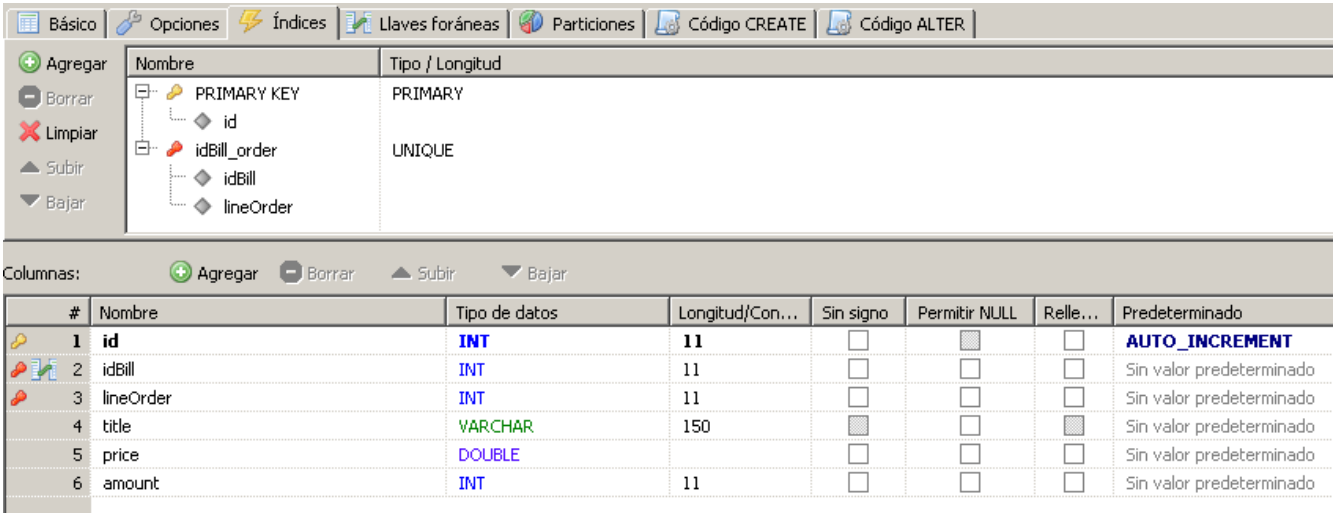
Creamos la tabla *book*:



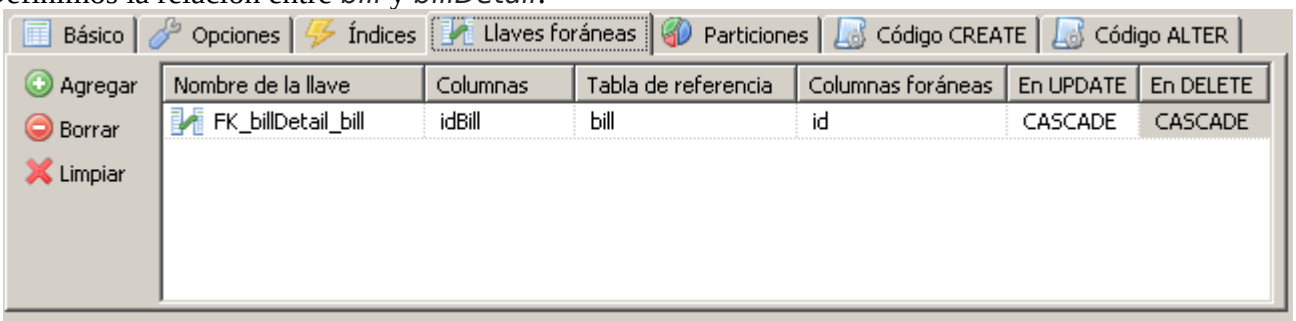
Creamos la tabla *bill*:



Y la tabla *billDetail*:



Definimos la relación entre *bill* y *billDetail*:



7. Creando la capa del modelo.

Una vez tenemos nuestra base de datos lista, vamos a reflejarla en nuestro codido java mediante un mapeo objeto-relacional, para ello utilizaremos el mapeo por anotaciones de Hibernate.

7.1 Mapeando la base de datos: Hibernate.

Para empezar, crearemos un paquete nuevo *model* donde iremos creando las clases necesarias.
La primera entidad que mapearemos sera *User*:

```java
@Entity
@Table(name = "user")
public class User {
    private int id;
    private String name;
    private String surname;
    private String dni;
    private String password;
    private transient boolean valid = false;

    @Id @Column(name = "id", nullable = false)
    @GeneratedValue(strategy=GenerationType.AUTO)
    public int getId() { return id; }

    @Basic @Column(name = "name", nullable = false, length = 50)
    public String getName() { return name; }

        @Basic @Column(name = "surname", nullable = false, length = 50)
    public String getSurname() { return surname; }

    @Basic @Column(name = "dni", nullable = false, length = 12)
    public String getDni() { return dni; }

    @Basic @Column(name = "password", nullable = false, length = 50)
    public String getPassword() { return password; }

    /* Settters, equals & hashCode omited */
}
```

Línea 1: Definimos la clase. Utilizamos la anotacion *@Entity* para indicar que se trata de una clase que debe ser persistida.
Línea 2: Especificamos el nombre de la tabla en la base de datos.
Línea 9: Creamos una propiedad adicional para saber si el usuario inicio sesion. No se guardará en la base de datos, por eso la marcamos como *transient*.
Línea 11-25: Los getters de las propiedades deberán llevar la anotación @Column para indicar de que columna de la base se deben extraer los datos. Indicamos tambien si dicha propiedad puede ser nula su longitud (si procede).
Línea 11: Anotamos la propiedad *Id* como clave primaria con *@Id*.
Línea 12: Especificamos que la propiedad id será generada por la base de datos.
Usando estas anotaciones, el mapeo de la entidad *Book* en la clase de su mismo nombre será trivial.

Continuaremos mapeando la entidad *Bill* y su relación con *billDetail*:

```
 1 @Entity
 2 @Table(name = "bill")
 3 public class Bill {
 4     private int id;
 5     private Date date;
 6     private String name;
 7     private String surname;
 8     private String dni;
 9     private Set<BillDetail> details = new HashSet<>();
10 
11     @Id @Column(name = "id")
12     @GeneratedValue(strategy=GenerationType.AUTO)
13     public int getId() { return id; }
14 
15     @Basic @Column(name = "date", insertable = false)
16     @GeneratedValue(strategy=GenerationType.AUTO)
17     public Date getDate() { return date; }
18 
19     @Basic @Column(name = "name", nullable = false, length = 50)
20     public String getName() { return name; }
21 
22     @Basic @Column(name = "surname", nullable = false, length = 50)
23     public String getSurname() { return surname; }
24 
25     @Basic @Column(name = "dni", nullable = false, length = 50)
26     public String getDni() { return dni; }
27 
28     @OneToMany(mappedBy="bill", cascade = {CascadeType.ALL})
29     public Set<BillDetail> getDetails() { return details; }
30 
31     /* Settters, equals & hashCode omited */
32 }
```

```
 1 @Entity
 2 @Table(name = "billDetail")
 3 public class BillDetail {
 4     private int id;
 5     private Bill bill;
 6     private int lineOrder;
 7     private String title;
 8     private double price;
 9     private int amount;
10 
11     @Id @Column(name = "id")
12     @GeneratedValue(strategy=GenerationType.AUTO)
13     public int getId() { return id; }
14 
15     @Basic @Column(name = "lineOrder", nullable = false)
16     public int getLineOrder() { return lineOrder; }
17 
18     @Basic @Column(name = "title", nullable = false, length = 150)
19     public String getTitle() { return title; }
20 
21     @Basic @Column(name = "price", nullable = false, precision = 0)
22     public double getPrice() { return price; }
23 
24     @Basic @Column(name = "amount", nullable = false)
25     public int getAmount() { return amount; }
26 
27     @ManyToOne @JoinColumn(name="idBill")
28     public Bill getBill() { return bill; }
29 
30     /* Settters, equals & hashCode omited */
31 }
```

Línea 28 (Bill): Establecemos la relación entre *Bill* y *BillDetail* como uno a muchos con la anotación *@OneToMany* especificando que será mapeada por la propiedad *bill* de *BillDetail.* También especificamos que se deben realizar todas las operaciones que re realicen sobre un objeto *Bill* en todos los objetos *BillDetail* de su propiedad *details* con *cascade = {CascadeType.ALL}*.

Línea 27 (BillDetail): Mapeamos la otra parte de la relación entre *Bill* y *BillDetail* que, desde el punto de vista de *BillDetail* es de muchos a uno, con la anotación de *@ManyToOne*. Especificamos la clave foránea con *@JoinColumn*.

Cuando tengamos nuestras tablas mapeadas, indicaremos en el archivo *hibernate.cfg.xml* que hemos realizado dichos mapeos:

```
1 <mapping class="model.Book"/>
2 <mapping class="model.User"/>
3 <mapping class="model.Bill"/>
4 <mapping class="model.BillDetail"/>
```

7.2 Capa de acceso a datos.

Una vez tenemos los *POJOs* preparados, pasaremos a crear la capa de acceso a datos. Para ello crearemos las clases *BookDao*, *UserDao* y *BillDao* que implementarán la interfaz *GenericDao:*

```
1 public interface GenericDao<T> {
2     List<T> all();
3     Optional<T> search(Serializable id);
4     void insert(T object);
5     void update(T object);
6     void delete(T object);
7 }
```

Empezaremos creando UserDao:

```
 1 public class UserDao implements GenericDao<User> {
 2     @Autowired private SessionFactory sessionFactory;
 3 
 4     @Override @SuppressWarnings("unchecked")
 5     public List<User> all() {
 6         return sessionFactory.getCurrentSession().createCriteria(User.class).list();}
 7 
 8     @Override
 9     public Optional<User> search(Serializable id) {
10         return Optional.of(
11                 (User) sessionFactory.getCurrentSession().byId(User.class).load(id));
12     }
13 
14     @Override
15     public void insert(User object) {
16         sessionFactory.getCurrentSession().save(object);
17     }
18 
19     @Override
20     public void update(User object) {
21         sessionFactory.getCurrentSession().merge(object);
22     }
23 
24     @Override
25     public void delete(User object) {
26         sessionFactory.getCurrentSession().delete(object);
27     }
```

Línea 2: Definimos una propiedad de tipo *SessionFactory* con la que obtendremos la sesión actual de Hibernate. Anotamos esta propiedad con *@Autowired* para que Spring inyecte este objeto.
Línea 4-6: Usando la sesion de Hibernate, obtenemos todos los usuarios.
Línea 8-12: Usando la sesion de Hibernate, obtendremos un usuario por su *id*.
Línea 14-17: Usando la sesion de Hibernate, insertaremos un nuevo usuario.
Línea 19-22: Usando la sesion de Hibernate, actualizaremos un usuario.
Línea 24-27: Usando la sesion de Hibernate, borraremos un usuario.
Teniendo en cuenta esta implementación, la creación de la clase *BookDao* será trivial.
Veamos ahora la implementación de *BillDao*:

```
1  public class BillDao implements GenericDao<Bill> {
2      @Autowired
3      private SessionFactory sessionFactory;
4      @Autowired private User user;
5
6      @Override @SuppressWarnings("unchecked")
7      public List<Bill> all() {
8          Criteria criteria = sessionFactory.getCurrentSession()
9                                            .createCriteria(Bill.class);
10         criteria.add(Restrictions.eq("dni", user.getDni()));
11         return criteria.list();
12     }
13
14     @Override @SuppressWarnings("unchecked")
15     public Optional<Bill> search(Serializable id) {
16         Criteria criteria = sessionFactory.getCurrentSession()
17                                           .createCriteria(Bill.class);
18         criteria.add(Restrictions.eq("dni", user.getDni()));
19         criteria.add(Restrictions.eq("id", id));
20         return criteria.list().stream().findAny();
21     }
22
23     @Override
24     public void insert(Bill object) {
25         object.getDetails().forEach(d -> d.setBill(object));
26         sessionFactory.getCurrentSession().save(object);
27     }
28
29     @Override
30     public void update(Bill object) {throw new UnsupportedOperationException();}
31     @Override
32     public void delete(Bill object) {throw new UnsupportedOperationException();}
```

Como podemos observar, la clase *BillDao* es muy similar a la de *UserDao*, solo que esta tiene en cuenta el *dni* del usuario para obtener las facturas correspondientes, limitando así el acceso de un usuario a las facturas de otro distinto.
También podemos ver en la **linea 25** como se establece la relación entre una factura y sus detalles.
Por último eliminamos la posibilidad de actualizar/eliminar una factura en las **lineas 29-32**.

7.3 Capa de control: la fachada.

Una vez terminada la capa de acceso a datos, pasaremos a implementar la capa de control, que actuara como fachada entre la capa del modelo y la capa del controlador. También se encargará de la traducción de excepciones.
Empezaremos creando la excepción *ModelException* que será el único tipo que la capa del modelo lanzará:

```
1  public class ModelException extends Exception {
2      private final Throwable cause;
3      public ModelException(Throwable cause) { this.cause = cause; }
4
5      @Override
6      public String getMessage() {
7          String message = "Error desconocido: " + cause.getMessage();
8          if (cause.getClass().equals(com.mysql.jdbc.exceptions.jdbc4
9                                       .CommunicationsException.class))
10             message = "Problemas en la conexión de la base de datos";
11
12         if (cause.getClass().equals(ArithmeticException.class))
13             message = "Excepción de operacion aritmetica";
14
15         if (cause.getClass().equals(IOException.class))
16             message = "Excepción de entrada/salida";
17
18         if (cause.getClass().equals(NumberFormatException.class))
19             message = "Excepción de formato incorrecto";
20
21         if (cause.getClass().equals(ClassNotFoundException.class))
22             message = "Excepción de clase no encontrada";
23         return message;
24     }
25 }
```

Definiremos la interfaz ControlDao:

```
1  public interface ControlDao<T> {
2
3      List<T> all() throws ModelException;
4      Optional<T> search(Serializable id) throws ModelException;
5      void insert(T object) throws ModelException;
6      void update(T object) throws ModelException;
7      void delete(T object) throws ModelException;
8      default List<T> query(Predicate<T> predicate) throws ModelException{
9          return all().stream().filter(predicate).collect(Collectors.toList());
10     }
11 }
```

Y crearemos la clase *UserControlDao* implementando esta interfaz:

```
1  public class UserControlDao implements ControlDao<User>{
2      @Autowired
3      private GenericDao<User> dao;
4
5      @Override
6      public List<User> all() throws ModelException {
7          try { return dao.all(); }
8          catch (Exception e){ throw new ModelException(e); }
9      }
10
11     @Override
12     public Optional<User> search(Serializable id) throws ModelException {
13         try { return dao.search(id); }
14         catch (Exception e){ throw new ModelException(e); }
15     }
16
17     @Override
18     public void insert(User object) throws ModelException {
19         try { dao.insert(object); }
20         catch (Exception e){ throw new ModelException(e); }
21     }
22
23     @Override
24     public void update(User object) throws ModelException {
25         try { dao.update(object); }
26         catch (Exception e){ throw new ModelException(e); }
27     }
```

```
28
29      @Override
30      public void delete(User object) throws ModelException {
31          try { dao.delete(object); }
32          catch (Exception e){ throw new ModelException(e); }
33      }
34  }
```

Como podemos observar, la clase de control delega el proceso a la capa de acceso a datos, recogiendo las excepciones y traduciéndolas a *ModelException*, reduciendo así el acoplamiento con las demás capas.
La creación de las clases *BookControlDao* y *BillControlDao* será identica a esta.

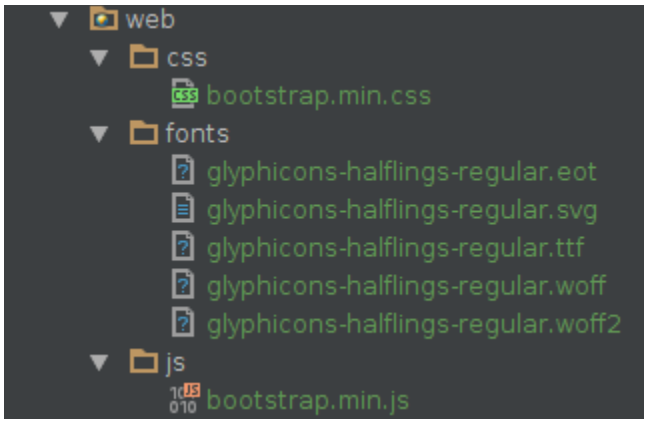
8. Creando la capa de la vista.

Una vez lista la capa del modelo, nos centraremos ahora en crear la capa de la vista, que será con la que interactuará directamente el usuario final.

8.1 Css is awesome: Bootstrap.

Bootstrap es un framework o conjunto de herramientas de código abierto para diseño de sitios y aplicaciones web. Contiene plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos de diseño basado en HTML y CSS, así como, extensiones de JavaScript opcionales adicionales.
Podemos descargar la versión por defecto en http://getbootstrap.com/ o personalizarlo para adaptarlo a las necesidades de nuestro sitio web en http://getbootstrap.com/customize/.

Una vez descargado, extraeremos el contenido en la carpeta *web* de nuestro proyecto:



Realizaremos los enlaces pertinentes en *WEB-INF/template/default/template.jsp* (nuestra plantilla base):

1. Dentro del head:

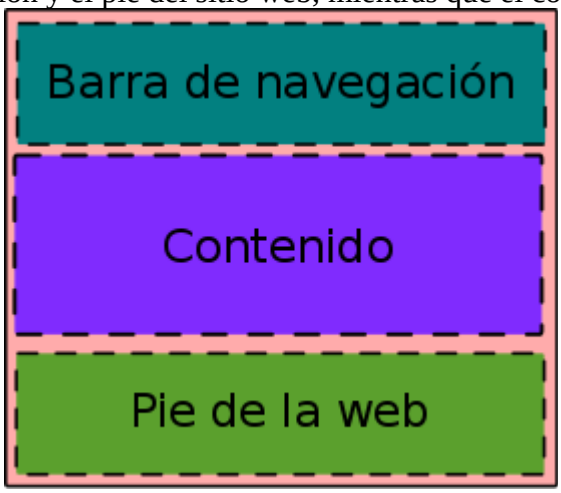
```
<link rel="stylesheet" href="$
{pageContext.request.contextPath}/css/bootstrap.min.css" />
```

2. Antes del cierre del body:

```
<script src="https://code.jquery.com/jquery-1.12.2.min.js"
            integrity="sha256-lZFHibXzMHo3GGeehn1hudTAP3Sc0uKXBXAzHX1sjtk="
            crossorigin="anonymous"></script>
    <script src="${pageContext.request.contextPath}/js/bootstrap.min.js">
</script>
```

8.2 Plantilla base.

En nuestra plantilla base definiremos las partes comunes de nuestro sitio web. Todas las vistas tendrán en común la barra de navegación y el pie del sitio web, mientras que el contenido será lo que variará:



- La barra de navegación:


```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed"
        data-toggle="collapse"
        data-target="#navbar-collapse" aria-expanded="false">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand"
      href="${pageContext.request.contextPath}/index.form">Inicio</a>
    </div>

    <div class="collapse navbar-collapse" id="navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a href="/book/list.form">Ver Libros</a></li>
      </ul>
```

```
20        <c:choose>
21          <c:when test="${not user.valid}">
22            <form class="navbar-form navbar-right" method="post"
23            action="${pageContext.request.contextPath}/user/login.form">
24              <div class="form-group">
25                <input type="text" class="form-control"
26                  placeholder="Usuario" name="username">
27                <input type="password" class="form-control"
28                  placeholder="Password" name="password">
29              </div>
30              <button type="submit" class="btn btn-default">Login</button>
31              <a class="btn btn-primary"
32                 href="${pageContext.request.contextPath}/user/register.form">
33                Registrarse
34              </a>
35            </form>
36          </c:when>
37
38          <c:otherwise>
39            <form class="navbar-form navbar-right" method="post"
40              action="${pageContext.request.contextPath}/user/logout.form">
41              <button type="submit" class="btn btn-default">Logout</button>
42            </form>
43          </c:otherwise>
44        </c:choose>
45      </div>
46    </div>
47  </nav>
```

Línea 12-13: Definimos el enlace al inicio.
Línea 18: Definimos el enlace a la lista de los libros.
Línea 20-44: Definimos la zona de login/registro o de logout, basándonos en la propiedad *valid* del usuario actual en la sesión. Si la propiedad es false (el usuario no a iniciado sesión) mostraremos los controles para iniciar sesión y un enlace a la vista de registro. En caso de que fuese verdadera, mostraremos la opción de terminar la sesión.

- Pie de la web:

```
1  <footer class="footer">
2      <div class="container text-center">
3          <p class="text-muted">Marco A. Fernández Heras<sup>©</sup>2015-2016</p>
4      </div>
5  </footer>
```

8.3 Mensajes.

Una parte importante de un sitio web es el feedback de eventos al usuario y como es algo que tendrán gran parte de las vistas, vamos a extrerla en una vista parcial que utilizaremos para mostrar dichos mensajes.
Para ello crearemos el archivo *WEB-INF/views/messages.jsp*, que luevo podremos importar en las diferentes vistas:

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:if test = "${message != null}">
    <div class="col-xs-12 col-md-10 col-md-offset-1
                alert alert-dismissible alert-success">
        <button type="button" class="close" data-dismiss="alert">×</button>
        <strong>¡Todo Correcto!</strong> <c:out value="${message}" />
    </div>
</c:if>
<c:if test="${error != null}">
    <div class="col-xs-12 col-md-10 col-md-offset-1
                alert alert-dismissible alert-danger">
        <button type="button" class="close" data-dismiss="alert">×</button>
        <strong>¡Error!</strong> <c:out value="${error}"/>
    </div>
</c:if>
```

Como se puede apreciar, si *message* o *error* están definidos, mostrará un *div* con el contenido del mensaje o error.

8.4 CRUD del libro.

Ahora que tenemos nuestra plantilla lista, empezaremos a crear las vistas necesarias para el CRUD (**C**reate, **R**ead, **U**pdate, **D**elete) de los libros. Primero crearemos una vista de lista, desde la cual enlazaremos a las demás en el archivo *WEB-INF/views/book/list.jsp*:

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<tiles:insertDefinition name="defaultTemplate">

 <tiles:putAttribute name="title">Librería Spring - Tiles</tiles:putAttribute>
```

```
 7
 8  <tiles:putAttribute name="content">
 9   <h2>Lista de libros:</h2>
10   <div class="col-xs-12">
11    <c:set var="user" value="${sessionScope.get('user')}"/>
12    <c:if test="${books.isEmpty() }">
13     <p>No hay Libros que mostrar</p>
14    </c:if>
15    <c:if test="${not books.isEmpty() }">
16    <table class="table table-striped centered">
17     <tbody>
18      <c:forEach var="book" items="${books}">
19       <tr>
20        <td>${book.title}</td>
21        <c:if test="${user.isValid()}">
22         <td class="pull-right">
23          <form action="${pageContext.request.contextPath}/book/delete.form"
24             method="post">
25           <input type="hidden" name="id" value="${book.id}" >
26           <button type="submit" class="btn btn-sm btn-danger">
27            <span class="glyphicon gtruelyphicon-remove"></span>
28           </button>
29          </form>
30         </td>
31         <td class="pull-right">
32          <a class="btn btn-primary btn-sm"
33            href="${pageContext.request.contextPath}/book/edit.form?id=${book.id}">
34             <span class="glyphicon glyphicon-pencil"></span>
35          </a>
36         </td>
37        </c:if>
38        <td class="pull-right">
39         <a class="btn btn-primary btn-sm"
40           href="${pageContext.request.contextPath}/book/view.form?id=${book.id}">
41            <span class="glyphicon glyphicon-eye-open"></span>
42         </a>
43         </td>
44       </tr>
45      </c:forEach>
46     </tbody>
47    </table>
48    </c:if>
```

```
49    <c:if test="${user.isValid()}">
50     <a type="button" class="btn btn-primary"
51       href="${pageContext.request.contextPath}/book/create.form">Añadir libro</a>
52    </c:if>
53   </div>
54  </tiles:putAttribute>
55
56 </tiles:insertDefinition>
```

Línea 12-14: En caso de que no hubiese ningún libro en la base de datos, informaremos al usuario.
Línea 15-48: Mostramos la lista de libros en una tabla.
Línea 21-37: Si el usuario es válido, mostramos los botones para editar y borrar cada libro.
Línea 38-43: Mostramos un botón que nos enlaza a la vista de detalles del libro.
Línea 21-37: Si el usuario es válido, mostramos un botón para añadir nuevos libros.

8.3.1 Datos del libros.

Tanto a la hora de crear un libro como de editarlo, los datos que debe rellenar el usuario son los mismo, lo que nos lleva a la conclusión que podemos extraer esa parte en una vista separada y reutilizarla en las otras dos.
Crearemos el archivo *WEB-INF/views/book/editbookData.jsp*:

```
 1 <table class="table-striped">
 2     <tbody>
 3     <tr>
 4         <td>Título</td>
 5         <td>
 6             <form:input path="title"
 7                   cssClass="form-control"
 8                   type="text" value="${ title }"/>
 9         </td>
10     </tr>
11     <tr>
12         <td>Autor</td>
13         <td>
14             <form:input path="author"
15                   cssClass="form-control"
16                   type="text"  value="${ author }"/>
17         </td>
18     </tr>
19     <tr>
20         <td>Tema</td>
```

```
21          <td>
22              <form:select path="topic" cssClass="form-control" >
23                  <form:option value="Accion" label="Accion" />
24                  <form:option value="Aventuras" label="Aventuras" />
25                  <form:option value="Biografía" label="Biografía" />
26                  <form:option value="Ciencia" label="Ciencia" />
27                  <form:option value="Ciencia Ficción" label="Ciencia Ficción" />
28                  <form:option value="Cine" label="Cine" />
29                  <form:option value="Economía" label="Economía" />
30                  <form:option value="Gastronomía" label="Gastronomía" />
31                  <form:option value="Historia" label="Historia" />
32                  <form:option value="Informática" label="Informática" />
33                  <form:option value="Medicina" label="Medicina" />
34                  <form:option value="Misterio" label="Misterio" />
35                  <form:option value="Naturaleza" label="Naturaleza" />
36                  <form:option value="Policíaco" label="Policíaco" />
37                  <form:option value="Política" label="Política" />
38                  <form:option value="Romántica" label="Romántica" />
39                  <form:option value="Teatro" label="Teatro" />
40                  <form:option value="Terror" label="Terror" />
41              </form:select>
42          </td>
43      </tr>
44      <tr>
45          <td>Numero de páginas</td>
46          <td>
47              <form:input cssClass="form-control"
48                    type="number" path="pages"
49                    value="${ pages }"/>
50          </td>
51      </tr>
52      <tr>
53          <td>Cartone</td>
54
55          <td><form:checkbox path="formatOne" value="${ !formatOne }" /></td>
56      </tr>
57      <tr>
58          <td>Rustica</td>
59          <td><form:checkbox path="formatTwo" value="${ !formatTwo }" /></td>
60      </tr>
61      <tr>
62          <td>Tapa dura</td>
```

```
63          <td><form:checkbox path="formatThree" value="${ !formatThree }" /></td>
64      </tr>
65      <tr>
66          <td>Estado</td>
67          <td>
68              <form:radiobutton path="isNew" value="true" label="Novedad"/>
69              <form:radiobutton path="isNew" value="false" label="Reedicion"/>
70          </td>
71      </tr>
72      <td>Precio</td>
73      <td>
74          <form:input cssClass="form-control"
75                type="number" step="0.01" path="price"
76                value="${ price }"/>
77      </td>
78      </tbody>
79  </table>
```

Usando la librería de etiquetas de spring, creamos los controles necesarios para editar los datos del libro. Usaremos *form:input* para las propiedades de tipo *String*, *form:checkbox* para las de tipo *boolean*. Usaremos un control de tipo *form:radiobutton* para la propiedad *isNew*, para que el usuario decida entre *Nuevo* o *Reedición.*
Cada *form*, tiene una propiedad *path* que indica a que propiedad del libro está vinculada además del *value* inicial. Para darle estilo a los *forms*, se utiliza la propiedad *cssClass*.

8.3.2 Creando libros.

Utilizando la vista parcial de introducción de datos, creamos la vista de creación de libros en el archivo *WEB-INF/views/book/create.jsp*:

```
1  <%@ page contentType="text/html; charset=UTF-8"%>
2  <%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
3  <%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
4  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
5
6  <tiles:insertDefinition name="defaultTemplate">
7      <tiles:putAttribute name="title">Librería Spring - Tiles
8      </tiles:putAttribute>
9
10     <tiles:putAttribute name="content">
11         <c:import url="../messages.jsp" />
12         <div class="col-xs-12 col-md-6 col-md-offset-3">
```

```
13              <h2 class="text-center">Nuevo libro</h2>
14              <h3>Datos:</h3>
15              <form:form
16                  action="${pageContext.request.contextPath}/book/create.form"
17                  method="post" modelAttribute="book">
18                  <%-- Importamos la tabla del libro --%>
19                  <c:import url="editbookData.jsp"/>
20                  <input class="btn btn-success pull-right" type="submit"
21                         value="Guardar"/>
22              </form:form>
23              <a type="button" class="btn btn-default"
24                 href="${pageContext.request.contextPath}/book/list.form">
25                 Volver
26              </a>
27          </div>
28      </tiles:putAttribute>
29  </tiles:insertDefinition>
```

Línea 15-22: Creamos el formulario para hacer *POST* a */book/create.form*.

8.3.3 Editando libros.

La vista para editar libros es prácticamente la misma que la de creación, cambiando solo la acción del formulario y un campo extra para el id del libro que estamos editando:

```
1  <form:form action="${pageContext.request.contextPath}/book/edit.form"
2        method="post" modelAttribute="book">
3      <%-- Importamos la tabla del libro --%>
4      <c:import url="editbookData.jsp"/>
5      <form:input type="hidden" value="${ book.id }" path="id">
6      <input class="btn btn-success pull-right" type="submit" value="Guardar"/>
7  </form:form>
```

Línea 1: Especificamos la acción del formulario para hacer *POST* a */book/edit.form*.
Línea 5: Creamos un campo oculto donde guardamos el id del libro que estamos actualizando.

8.3.4 Viendo libros.

Crearemos el archivo de la vista de libro, donde mostraremos las propiedades de un libro.
/WEB-INF/views/book/view.jsp:

```
<tiles:insertDefinition name="defaultTemplate">
    <tiles:putAttribute name="title">Librería Spring - Tiles</tiles:putAttribute>

    <tiles:putAttribute name="content">
        <c:import url="../messages.jsp" />
        <div class="col-xs-12 col-md-6 col-md-offset-3">
            <h2 class="text-center">${book.title}</h2>
            <h3>Datos:</h3>
            <table class="table-striped">
                <tbody>
                <tr>
                    <td>Título</td>
                    <td>${ book.title }</td>
                </tr>
                <tr>
                    <td>Autor</td>
                    <td>${ book.author }</td>
                </tr>
                <tr>
                    <td>Tema</td>
                    <td>${book.topic}</td>
                </tr>
                <tr>
                    <td>Numero de páginas</td>
                    <td>${ book.pages }</td>
                </tr>
                <tr>
                    <td>Cartone</td>
                    <td><form:checkbox path="book.formatOne"
                               value="${ !book.formatOne }" disabled="true" /></td>
                </tr>
                <tr>
                    <td>Rustica</td>
                    <td><form:checkbox path="book.formatTwo"
                               value="${ !book.formatTwo }"  disabled="true" /></td>
                </tr>
                <tr>
                    <td>Tapa dura</td>
                    <td><form:checkbox path="book.formatThree"
```

```
40                                  value="${ !book.formatThree }"  disabled="true"/></td>
41                 </tr>
42                 <tr>
43                     <td>Estado</td>
44                     <td>
45                         <form:radiobutton path="book.isNew"
46                               value="true" label="Novedad" disabled="true"/>
47                         <form:radiobutton path="book.isNew"
48                               value="false" label="Reedicion" disabled="true"/>
49                     </td>
50                 </tr>
51                 <td>Precio</td>
52                 <td>${ book.price }</td>
53                 </tbody>
54             </table>
55             <a type="button" class="btn btn-default"
56                href="${pageContext.request.contextPath}/book/list.form">Volver</a>
57         </div>
58     </tiles:putAttribute>
59 </tiles:insertDefinition>
```

Como podemos apreciar, lo único que hacemos en esta vista es mostrar las propiedades del libro.

8.5 Registro de usuarios.

Una vez listas las vistas necesarias para hacer el CRUD de los libros, crearemos la vista para el registro de usuarios. Para ello crearemos el archivo *WEB-INF/views/user/register.jsp*:

```
1  <%@ page contentType="text/html; charset=UTF-8"%>
2  <%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
3  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
4  <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
5
6  <tiles:insertDefinition name="defaultTemplate">
7      <tiles:putAttribute name="title">Librería Spring - Tiles</tiles:putAttribute>
8
9      <tiles:putAttribute name="content">
10         <c:import url="../messages.jsp" />
11         <h2>Registro de usuarios:</h2>
12         <div class="col-xs-12 col-md-6 col-md-offset-3">
13             <form action="${pageContext.request.contextPath}/user/register.form"
```

```
14                    method="post">
15                <div class="form-group">
16                    <label class="control-label">Nombre:</label>
17                    <form:input path="user.name" type="text"
18                          cssClass="form-control"/>
19                </div>
20                <div class="form-group">
21                    <label class="control-label">Apellidos:</label>
22                    <form:input path="user.surname" type="text"
23                          cssClass="form-control"/>
24                </div>
25                <div class="form-group">
26                    <label class="control-label">DNI:</label>
27                    <form:input path="user.dni" type="text"
28                          cssClass="form-control"/>
29                </div>
30                <hr/>
31                <label class="control-label">Password:</label>
32                <form:input path="user.password" type="password"
33                      cssClass="form-control"/>
34                <hr/>
35                <input class="btn btn-success pull-right" type="submit"
36                       value="Registrarse"/>
37            </form>
38        </div>
39    </tiles:putAttribute>
40 </tiles:insertDefinition>
```

En cuanto a gestión de usuarios, está sera la única vista que necesitaremos, ya que el inicio/find de sesión está definido en la plantilla base.

Utilizando las etiquetas vistas hasta ahora, podremos crear sin problemas las vistas necesarias tanto para la gestión del carro de la compra como para la gestión de facturas.

9. Creando la capa del controlador.

Ahora que tenemos listas tanto la capa de la vista como la capa del modelo, procederemos a crear la capa del controlador, que unirá las dos anteriores y dará respuesta a las peticiones efectuadas por los usuarios.

9.1 BookController.

Empezaremos creando el controlador encargado de canalizar las operaciones CRUD de los libros desde el usuario a la base de datos. Para ello creamos una nueva clase dentro del paquete *spring* llamada *BookController*:

```
1 @Controller
2 @RequestMapping("/book")
3 public class BookController {
4
5     @Autowired
6     private ControlDao<Book> dao;
7 }
```

Líneas 1 y 2: Usamos la anotacion *@Controller* y *@RequestMapping* tal como vimos en el capítulo 4, salvo que esta vez vamos a mapear todos los métodos del controlador a la ruta base */book*.
Línea 6: Pediremos la inyección automática de un objecto de tipo *ControlDao<Book>* usando la anotacion *@Autowired*, tal como hicimos anteriormente en nuestros *GenericDaos* y *ControlDaos*.

Una vez tenemos la clase creada, iremos añadiendo los métodos necesarios para efectuar el CRUD completo de los libros.

9.1.1 Lista de libros.

```
 1 @RequestMapping(value = "/list" , method = RequestMethod.GET)
 2 public ModelAndView list(){
 3     ModelAndView model = new ModelAndView("book/list");
 4     try {
 5         List<Book> books = dao.all();
 6         model.addObject("books", books);
 7     } catch (ModelException ex) {
 8         ex.printStackTrace();
 9         model.addObject("error", ex);
10     }
11     return model;
12 }
```

Línea 1: Usamos la anotación *@RequestMapping* para mapear este método a la ruta */list*, por lo que la ruta de acceso a este método sería */book/list.* También especificamos que esta ruta solo responde al método *GET*.
Línea 2: El tipo de retorno de este método es del tipo *ModelAndView*, que representa una vista en conjunto con los datos necesarios para su renderizado.
Línea 2: Creamos un nuevo *ModelAndView*, pasandole en nombre de la vista que queremos utilizar.
Líneas 5 y 6: Usamos el objeto de control para obtener la lista completa de los libros, la que pasamos posteriormente al *ModelAndView* como datos para su renderizado.
Línea 9: En caso de excepción, añadimos la excepción al *ModelAndView*, para mostar el mensaje de error en la zona de mensajes de las vistas.

9.1.2 Ver libro.

```
1  @RequestMapping(value = "/view" , method = RequestMethod.GET)
2  public ModelAndView view(@RequestParam int id){
3      ModelAndView model = new ModelAndView("book/view");
4      try {
5          Optional<Book> bookOptional = dao.search(id);
6          if(bookOptional.isPresent()){
7              model.addObject("book", bookOptional.get());
8          }
9          else{
10             model.addObject("error", "Libro no encontrado");
11         }
12     } catch (ModelException ex) {
13         ex.printStackTrace();
14         model.addObject("error", ex);
15     }
16     return model;
17 }
```

El procedimiento es prácticamente idéntico al de la lista de libros, mapear el método a la ruta deseada, crear un nuevo *ModelAndView*, añadir los datos necesarios utilizando el objeto de control y retornarlo. Lo único que diferencia este método del anterior son los *parámetros*, que en este caso se necesita el *id* del libro que queremos visualizar. Utilizando la anotación *@RequestParam*, indicamos que esperamos recibir dicho *parámetro* a través de los parámetros de la petición *Http*.
En las **líneas 6-11**, podemos ver como si el *id* recibido no coincidiese con ningún libro registrado actualmente, mostraríamos un error de *Libro no encontrado*

9.1.3 Crear libro.

```java
@RequestMapping(value = "/create" , method = RequestMethod.GET)
public ModelAndView createView(){
    ModelAndView model = new ModelAndView("book/create");
    model.addObject("book", new Book());
    return model;
}

@RequestMapping(value = "/create", method = RequestMethod.POST)
public ModelAndView create(@ModelAttribute("book") Book book){
    ModelAndView model = new ModelAndView("book/create");
    try {
        dao.insert(book);
        model.addObject("message", "Libro creado con éxito");
        model.addObject("book", new Book());
    } catch (ModelException ex) {
        ex.printStackTrace();
        model.addObject("book", book);
        model.addObject("error", ex);
    }
    return model;
}
```

Como podemos ver el proceso de crear un nuevo libro se divide en dos partes. En la primera parte, el usuario solicita la vista de creación de libros y en la segunda parte, el usuario envía los datos de dicha vista de vuelta al servidor para su procesamiento. Por lo tanto, creamos dos métodos que responden a la misma ruta, pero uno de ellos acepta métodos *GET* mientras que el otro solo acepta métodos *POST.*

En el método *createView*, creamos un *ModelAndView* de la vista de crear al cuál le pasamos como objeto de modelo un libro nuevo.
El el método *create*, especificamos que queremos recoger un atributo del modelo con la anotación *@ModelAttribute* que, en este caso, va a ser de tipo libro. Este atributo tendrá los datos introducidos por el usuario en la vista de cración y seran los que, utilizando el objeto de control, introduciremos en la base de datos.

Si todo va bien, se mostrará un mensaje de *“Libro creado con éxito”* y pasaremos un libro nuevo de vuelta para que el usuario pueda insertar otro libro más.
En caso de que hubiese alguna excepción, devolveremos el mismo libro al usuario para que revise los datos introducidos, así como un mensaje de error.

9.1.4 Editar libro.

```java
@RequestMapping(value = "/edit", method = RequestMethod.GET)
public ModelAndView editView(@RequestParam int id){
    ModelAndView model = new ModelAndView("book/edit");
    try {
        Optional<Book> bookOptional = dao.search(id);
        if(bookOptional.isPresent()){
            model.addObject("book", bookOptional.get());
        }
        else{
            model.addObject("error", "Libro no encontrado");
        }
    } catch (ModelException ex) {
        ex.printStackTrace();
        model.addObject("error", ex);
    }
    return model;
}

@RequestMapping(value = "/edit", method = RequestMethod.POST)
public ModelAndView edit(@ModelAttribute("book") Book book){
    ModelAndView model = new ModelAndView("book/edit");
    model.addObject("book", book);
    try {
        dao.update(book);
        model.addObject("message", "Libro actualizado con éxito");
    } catch (ModelException ex) {
        ex.printStackTrace();
        model.addObject("error", ex);
    }
    return model;
}
```

Como podemos ver, la edición de libros sigue la misma dinámica que la creación, es decir, un método que responde a *GET* y que retorna la vista de edición y otro método que responde a *POST* y persiste los datos actualizados.
La única diferencia es que en este caso queremos actualizar unos datos que ya existían, por lo que debemos recuperarlos de la base de datos y mostrarlos en la vista de edición. En casos de que esos datos no existiesen con anterioridad, mostraremos un error de *“Libro no encontrado”*.

9.1.5 Borrar libro.

```java
@RequestMapping(value = "/delete", method = RequestMethod.POST)
public ModelAndView delete(@RequestParam int id){
    ModelAndView model = new ModelAndView("book/list");

    try {
        Optional<Book> book = dao.search(id);
        if(book.isPresent()){
            dao.delete(book.get());
            model.addObject("message", "Libro borrado con éxito");
        }
    } catch (ModelException ex) {
        model.addObject("error", ex);
    }

    try {
        List<Book> books = dao.all();
        model.addObject("books", books);
    } catch (ModelException ex) {
        ex.printStackTrace();
        model.addObject("error", ex);
    }
    return model;
}
```

En el caso del borrado de libros, solo existirá un método *delete* que aceptará peticiones *POST*. Este método recibirá como parámetro el *id* del libro a borrar y retornará la vista de lista de vista de libros.

9.2 UserController.

El controlador *UserController* se encarga del registro, autenticación, ingreso y salida de los usuarios. Crearemos la clase dentro del paquete *spring* y le añadimos un campo de tipo *ControlDao<User>* y otro campo de tipo *User*:

```
1  @Controller
2  @RequestMapping("/user")
3  public class UserController {
4
5      @Autowired
6      private ControlDao<User> dao;
7
8      @Autowired
9      private User user;
10 }
```

9.2.1 Registro de usuarios.

Del mismo modo que la creación y edición de libros, el proceso de registro de un usuario contra de dos partes, la solicitud de la vista y el procesamiento de los datos:

```
1  @RequestMapping(value = "/register" , method = RequestMethod.GET)
2  public ModelAndView registerView(){
3      ModelAndView modelAndView = new ModelAndView("user/register");
4      modelAndView.addObject("user", new User());
5      return modelAndView;
6  }
7
8  @RequestMapping(value = "/register" , method = RequestMethod.POST)
9  public ModelAndView register(@ModelAttribute("user") User user){
10     ModelAndView model = new ModelAndView("user/register");
11     try {
12         dao.insert(user);
13         model.addObject("message", "Registro completado");
14         model.addObject("user", new User());
15     } catch (ModelException ex) {
16         ex.printStackTrace();
17         model.addObject("user", user);
18         model.addObject("error", ex);
19     }
20     return model;
21 }
```

En el método *registerView*, devolvemos al usuario la vista para que introduzca los datos necesarios para el registro de un nuevo usuario.
En el método *register*, procesamos los datos introducidos por el usuario utilizando el objecto de control de dato para inserta un nuevo usuario en la base de datos.

9.2.2 Inicio de sesión.

Para el inicio de sesión, crearemos otro método, *login*, que aceptará solo peticiones *POST*.

```
1  @RequestMapping(value = "/login" , method = RequestMethod.POST)
2  public String login(HttpServletRequest request, @RequestParam String dni,
3                                          @RequestParam String password)
4  {
5      try {
6          List<User> users = dao.query(u -> u.getDni().equalsIgnoreCase(dni)
7                                      && u.getPassword().equals(password));
8          if(!users.isEmpty()){
9              User _user = users.get(0);
10             request.getSession().setAttribute("actualUser", _user);
11             user.setId(_user.getId());
12             user.setName(_user.getName());
13             user.setSurname(_user.getSurname());
14             user.setDni(_user.getDni());
15             user.setValid(true);
16             _user.setValid(true);
17         }
18     } catch (ModelException ex) {
19         ex.printStackTrace();
20     }
21     return "index";
22 }
```

Este método, aceptará como parámetros un objeto de tipo *HttpServletRequest* así como los String de nombre de usuario y contraseña.
Mediante el objecto de control, intentaremos recuperar algún usuario que coincida con los datos proporcionados, y en caso de encontrarlo, usaremos el objeto *HttpServletRequest* para añadirlos a la sesión.

9.2.3 Fin de sesión.

Para el fin de sesión, añadiremos un método más al controlador, llamado logout, que aceptará peticiones POST e invalidará la sesión actual:

```
@RequestMapping(value = "/logout" , method = RequestMethod.POST)
public String logout(HttpServletRequest request){
    request.getSession().invalidate();
    this.user.setValid(false);
    return "index";
}
```

Al invalidar la sesión, se destruyen todos los datos que hallamos ido guardando en la misma.
Una vez que los datos han sido limpiados, mostraremos al usuario la página de inicio.

9.3 CartController.

Para la gestión del carro de la compra, crearemos otro controlador llamado *CartController*:

```
@Controller
@RequestMapping("/cart")
public class CartController {
    @Autowired
    Bill bill;

    @Autowired
    private ControlDao<Book> dao;

    @Autowired
    private ControlDao<Bill> billDao;
}
```

Como el controlador del carro de la compra se va a encargar de añadir/quitar libros del carro y tambien de la compra y creación de la factura, necesitaremos un objecto de control de libros y otro objeto de control de facturas. Tambien necesitaremos una factura tempolar que usaremos a modo de almacen de datos.

9.3.1 Ver el carrito.

Para mostrar un listado de lo que tenemos en el carro, crearemos el método *view* donde simplemente retornaremos la vista de ver carro.

```
@RequestMapping(value = "/view", method = RequestMethod.GET)
public String view(){
    return "cart/view";
}
```

9.3.2 Añadir libro al carrito.

Para añadir libros al carro de la compra, crearemos el metodo *add* que responderá a peticiones POST y nos redirigirá a la vista de ver carro:

```
1  @RequestMapping(value = "/add", method = RequestMethod.POST)
2  public String add(@RequestParam int id){
3      try {
4          Optional<Book> bookOptional = dao.search(id);
5          if(bookOptional.isPresent()){
6              Book book = bookOptional.get();
7              Optional<BillDetail> detail = bill.getDetails().stream()
8                      .filter(d -> d.getTitle().equalsIgnoreCase(book.getTitle()))
9                      .findFirst();
10             if(detail.isPresent()){
11                 detail.get().setAmount(detail.get().getAmount() + 1);
12             }
13             else{
14                 BillDetail billDetail = new BillDetail();
15                 billDetail.setAmount(1);
16                 billDetail.setTitle(book.getTitle());
17                 billDetail.setPrice(book.getPrice());
18                 bill.getDetails().add(billDetail);
19             }
20         }
21     } catch (ModelException e) {
22         e.printStackTrace();
23     }
24     return "cart/view";
25 }
```

Este método recibe el *id* de un libro. Después de asegurarnos que ese id corresponde realmente a un libro (**linea 5**), comprobamos si dicho libro ya se encuentra en el carro actualmente (**linea 10**). En caso de estar, aumentamos la cantidad en uno (**linea 11**). Si por el contrario el libro no estuviese, introducimos una nueva linea a la factura temporal (**lineas 14-18**).

9.3.3 Quitar libro del carrito.

Igual que tenemos un método para añadir libros al carrro, necesitamos un método para eliminarlos en el caso de que el usuario cambie de opinión y decida que ya no quiere el libro en cuestion. Añadiremos pues el método *remove* al contolador de carro:

```
1  @RequestMapping(value = "/remove", method = RequestMethod.POST)
2  public String remove(@RequestParam String title) {
3      Optional<BillDetail> detail = bill.getDetails().stream()
4              .filter(d -> d.getTitle().equalsIgnoreCase(title))
5              .findFirst();
6      if (detail.isPresent()) {
7          detail.get().setAmount(detail.get().getAmount() - 1);
8          bill.getDetails().removeIf(c -> c.getAmount() == 0);
9      }
10     return "cart/view";
11 }
```

Este método, que recibe por parámetro el título del libro que queremos eliminar, busca en los libros actuales si el titulo se encuentra realmente entre los introducidos con anterioridad (**lineas 3-6**). En caso de encontrarlo, disminuimos su cantidad en uno (**linea 7**) y si la cantidad llegase a 0, lo eliminamos completamente (**linea 8**).
Por último, tal como hicimos en le método de añadir libro, redirigimos al usuario a la vista de ver carro.

9.3.4 Comprar.

Una vez que el usuario ha accedido al sistema, ha decidido los libros que desea comprar y los ha introducido en el carro de la compra, el siguiente paso es la compra propiamente dicha.
Para ello, añadiremos un último método al controlador de carro, llamado *buy*:

```
1  @RequestMapping(value = "/buy", method = RequestMethod.POST)
2  public String buy(HttpServletRequest request){
3      try {
4          int i = 1;
5          for (BillDetail detail: bill.getDetails()) {
6              detail.setLineOrder(i);
7              i++;
8          }
9          User user = (User) request.getSession().getAttribute("actualUser");
10         bill.setName(user.getName());
11         bill.setSurname(user.getSurname());
12         bill.setDni(user.getDni());
13         billDao.insert(bill);
14         bill.setId(0);
15         bill.getDetails().clear();
16     } catch (ModelException e) { e.printStackTrace(); }
17     return "redirect:/bill/list.form";
18 }
```

En este método, lo primero que hacemos es asignarle a cada linea de pedido su número de orden (**lineas 5-8**). Después, establecemos el nombre, apellido y d.n.i. del usuario que está realizando la compra (**líneas 10-12**). Este usuario es el que inició sesión con anterioridad, por lo que lo recuperaremos de la session.
Usando el objecto de control de las facturas, insertamos la nueva factura (**linea 13**).
Por último reseteamos los valores de la factura temporal para que el usuario pueda realiza una nueva compra.

9.4 BillController.

Ahora que el usuario puede realizar compras en nuestro sitio web, crearemos un controlador que permitirá al usuario consultar su facturas.
Para ello crearemos una nueva clase llamada *BillController*:

```
1  @Controller
2  @RequestMapping("/bill")
3  public class BillController {
4
5      @Autowired
6      private ControlDao<Bill> dao;
7
8      @Autowired
9      private User user;
10 }
```

Esta clase necesitará un objecto de control de facturas así como un objecto usuario del cuál recuperar las facturas.
Esta clase también contará con dos método que informarán al usuario del listado de sus facturas y de los detalles de una factura en particular. Empezaremos creando el primero de ellos:

```
1  @RequestMapping(value = "/list" , method = RequestMethod.GET)
2  public ModelAndView list(){
3      ModelAndView model = new ModelAndView("bill/list");
4      try {
5          List<Bill> bills = dao.query(bill -> bill.getDni()
6                                  .equalsIgnoreCase(user.getDni()));
7          model.addObject("bills", bills);
8      } catch (ModelException ex) {
9          model.addObject("error", ex);
10     }
11     return model;
12 }
```

Aquí seguimos el mismo procedimiento que en los listados anteriores, utilizando el objeto de control, recuperamos los datos pertinentes (en este caso las facturas del usuario actual) y se las pasamos a la vista por medio de un *ModelAndView*.

```
@RequestMapping(value = "/view" , method = RequestMethod.GET)

public ModelAndView view(@RequestParam int id){
    ModelAndView model = new ModelAndView("bill/view");
    try {
        Optional<Bill> billOptional = dao.search(id);
        if(billOptional.isPresent() &&
           billOptional.get().getDni().equalsIgnoreCase(user.getDni()))
        {
            model.addObject("bill", billOptional.get());
        }
        else{
            model.addObject("error", "Factura no encontrada");
        }
    } catch (ModelException ex) {
        ex.printStackTrace();
        model.addObject("error", ex);
    }
    return model;
}
```

En la vista de detalle, intentamos recuperar la factura del id solicitado, siempre asegurándonos que dicha factura corresponde al usuario actual. Si todo es correcto le mostraremos los detalles de la factura, en caso contrario mostraremos un mensaje de error.

10. Uniendo las capas: Spring Container.

10.1 ¿Qué es el Spring Container?.

El *framework Spring* se compone de varios módulos, todos ellos giran entorno al *Spring Core* y más concretamente al *Spring Container*, el cual hace uso intensivo del patrón de inyección de Dependencias o de Inversión de Control, por lo tanto, para trabajar con bien con *Spring* es imprescindible conocer cómo funciona el contenedor de *Spring* y en qué consiste la *Inyección de Dependencias.*

El contenedor de *Spring* es uno de los puntos centrales de *Spring*, se encarga de crear los objetos, conectarlos entre si, configurarlos y además controla los ciclos de vida de cada objeto mediante el patrón de *Inyección de Dependencias (Dependency Injection* ó *DI)*.

Podemos personalizar el contenedor de *Spring* mediante configuración XML o programáticamente (con anotaciones).

Los *Bean* se pueden declarar mediante anotaciones en *POJO's* (*Plain Old Java Object* , objetos normales de Java) o mediante XML.

Los *beans* son la manera que tiene de denominar *Spring* a los objetos Java de los que se encarga, es decir aquellos que se encuentren en el contenedor de *Spring*.

En el contenedor Spring se suelen crear y almacenar objetos de servicio, DAO's, y objetos que nos permitan conectarnos con otras partes del sistema como Bases de Datos, Sistemas de Colas de Mensaje, etc…

10.1.1 Inyección de dependencias.

El patrón de Inyección de Dependencias, también conocido como de Inversión de Control es un patrón que tiene como finalidad conseguir un código mas desacoplado, que nos facilitará las cosas a la hora de hacer Tests y además nos permite cambiar partes del sistema más fácilmente en caso de que fuese necesario.

Tener el código desacoplado nos permite cambiar las dependencias en tiempo de ejecución basándonos en cualquier factor que considerásemos, para ello necesitaríamos un Inyector o Contenedor que sería el encargado de inyectar las dependencias correctas en el momento necesario.

Siguiendo el patrón de Inyección de Dependencias (DI, Dependency Injection) los componentes declaran sus dependencias, pero no se encargan de conseguirlas, ahí es donde entra el Contenedor de Spring, que en nuestras aplicaciones de Spring será el encargado de conseguir e inyectar las

dependencias a los objetos.

Spring permite la inyección mediante parámetros en el constructor o también permite inyectar la dependencia mediante los Setter (métodos set*()), cada forma de inyectar las dependencias tiene sus ventajas y sus desventajas, aunque la mayoría de los desarrolladores prefieren inyectar las dependencias mediante los métodos Set.

Mediante la anotación *@autowired* que hemos estado usando durante todo el proyecto, le indicamos a Spring que se tiene que encargar de buscar un Bean que cumpla los requisitos para ser inyectado, en caso de que hubiese mas de un Bean que cumpliese esos requisitos tendríamos que decirle a Spring cuál es el Bean correcto.

10.2 Configurando el Spring Container.

Como ya hemos comentado, el contenedor de Spring puede configurarse mediante XML o mediante anotaciones. En nuestro proyecto usaremos el primer método, añadiendo los cambios necesarios en el archivo *WEB-INF/spring/appServlet/servlet-context.xml.*

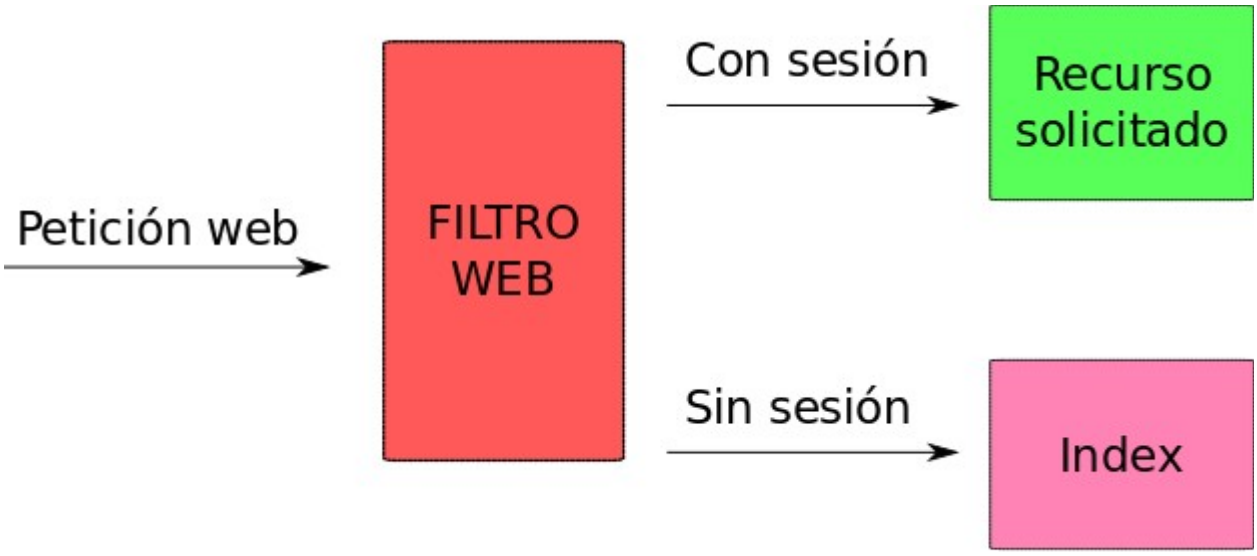
```
1  <bean id="user" class="model.User" scope="session" />
2  <bean id="cart" class="model.Bill" scope="session" />
3
4  <bean id="bookControlDao" class="control.BookControlDao" />
5  <bean id="userControlDao" class="control.UserControlDao" />
6
7
8  <bean id="billDao" class="model.dao.BillDao">
9      <property name="user" ref="user"/>
10 </bean>
11
12 <bean id="billControlDao" class="control.BillControlDao">
13     <property name="dao" ref="billDao" />
14 </bean>
15
16 <bean id="cartController" class="spring.CartController">
17     <property name="bill" ref="cart" />
18 </bean>
```

Declaramos los *beans* necesarios para nuestra aplicación, especificando las referencias que tienen entre ellos.

11. Seguridad básica.

11.1 Filtros de acceso.

En este punto del desarrollo tenemos una aplicación web totalmente funcional, pero que carece del mas mínimo sistema de seguridad.
En este capítulo vamos a implementar un básico sistema de acceso utilizando filtros web, que darán acceso al usuario a ciertas zonas de nuestra web dependiendo de si a iniciado sesión o no.



Empezaremos creando el filtro web que comprobará si el usuario ha iniciado sesión. Para ello creamos un nuevo paquete llamado *filter* y dentro creamos una clase llamada *LoggedFilter*:

```
1  @WebFilter( urlPatterns = {"/bill/*",
2                             "/book/create.form",
3                             "/book/edit.form",
4                             "/book/delete.form",
5                             "/cart/buy.form"}
6                           )
7  public class LoggedFilter implements Filter {
8
9      public void doFilter(ServletRequest request,
10                          ServletResponse response,
11                          FilterChain chain)
12                          throws IOException, ServletException
13     {
14         HttpSession session = ((HttpServletRequest) request).getSession();
```

```
15          Object oUser = session.getAttribute("actualUser");
16          if(oUser != null){
17              chain.doFilter(request, response);
18          }
19          else {
20              request.getRequestDispatcher("/index.form")
21                      .forward(request, response);
22          }
23      }
24  }
```

Líneas 1-6 : Mediante el uso de la anotación *@WebFilter* mapeamos este filtro a los patrones de url que queramos. En este caso, vamos a mapearlo a todas las peticiones que se hagan al controlador *Billcontroller* que respondía a las peticiones dentro de */bill/.* Utilizamos el asterisco para especificar que este filtro interceptará todas las peticiones de esa ruta.
También mapeamos las operaciones que modifican los datos de los libros (crear, editar y borrar).
Por último mapearemos a este filtro la petición de compra (*/cart/buy*) .

Línea 7 : Hacemos que nuestro flitro implemente la interfaz *Filter*.
Líneas 9-22 : Implementamos el método *doFilter* el cual es el encargado de permitir el acceso al recurso solicitado o de desviar el flujo hacia la página de inicio.
Para ello, intentamos extraer de la sesión el atributo *actualUser* (**línea 15**), el cúal, en caso de ser diferente de *null*, será indicativo de que el usuario ha iniciado sesión. En caso de ser así, continuaremos el flujo normal mediante la invocación del método *doFilter* de la cadena de filtros. Si un fese el caso, desviamos el flujo hacia la página de inicio (**lineas 20-21**).

12. Web asíncrona: AJAX.

12.1 ¿Qué es AJAX?

AJAX, acrónimo de Asynchronous JavaScript And XML (JavaScript asíncrono y XML), es una técnica de desarrollo web para crear aplicaciones interactivas o RIA (Rich Internet Applications). Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones.

Ajax es una tecnología asíncrona, en el sentido de que los datos adicionales se solicitan al servidor y se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página, aunque existe la posibilidad de configurar las peticiones como síncronas de tal forma que la interactividad de la página se detiene hasta la espera de la respuesta por parte del servidor.

JavaScript es un lenguaje de programación (scripting language) en el que normalmente se efectúan las funciones de llamada de Ajax mientras que el acceso a los datos se realiza mediante XMLHttpRequest, objeto disponible en los navegadores actuales. En cualquier caso, no es necesario que el contenido asíncrono esté formateado en XML (de hecho, nosotros usaremos formato JSON (JavaScript Object Notation) en nuestra aplicación).

Ajax es una técnica válida para múltiples plataformas y utilizable en muchos sistemas operativos y navegadores dado que está basado en estándares abiertos como JavaScript y Document Object Model (DOM).

12.2 Implementado AJAX: En el servidor.

En particular, vamos a realizar de forma asíncrona las peticiones de añadir y quitar libros del carrito de la compra.
Como es de esperar, debemos adaptar nuestro código del lado del servidor para aceptar, procesar y responder a esta peticiones AJAX.

12.2.1 Dependencias.

Lo primero que haremos será instalar las dependencias necesarias para dar soporte al formato JSON desde java. Para ello instalaremos las siguientes dependencias tal como hicimos en el capítulo 2:

- com.fasterxml.jackson.core:jackson-core:2.4.3
- com.fasterxml.jackson.core:jackson-databind:2.4.3

12.2.2 El objeto de respuesta.

Una vez tenemos las dependencias instaladas, vamos a crear la clase que usaremos para intercambiar información con el cliente en las peticiones AJAX.
Para ello crearemos una clase nueva en el paquete *model* llamada *AjaxCartResponse*:

```
1  public class AjaxCartResponse {
2
3      public static final int OK_CODE = 1;
4      public static final int FAIL_CODE = 0;
5
6      private int code = FAIL_CODE;
7      private int cartAmount = 0;
8      private int lineAmount = 1;
9      private double subtotal = 0;
10     private double total = 0;
11
12     /* Getters & Setters */
13 }
```

En este objeto almacenaremos el código de estado de la operación (**lineas 3-6**), el total de libros en el carro (**línea 7**), el total de libros en la línea modificada (**línea 8**), el subtotal de la línea modificada (**línea 9**) y el total del carro (**línea 10**).

12.2.3 El controlador.

Ahora que hemos definido el tipo de datos que se va a utilizar en el proceso AJAX, modificaremos nuestro CartController para adaptarlo a esta nueva forma de actuación:
Veamos el método de añadir libro al carrito:

```
@ResponseBody
@RequestMapping(value = "/add",
                method = RequestMethod.POST,
                produces = MediaType.APPLICATION_JSON_VALUE,
                consumes = MediaType.APPLICATION_JSON_VALUE)
public AjaxCartResponse add(@RequestBody Book bookToAdd){
    AjaxCartResponse response = new AjaxCartResponse();
    try {
        Optional<Book> bookOptional = dao.search(bookToAdd.getId());
        if(bookOptional.isPresent()){
            Book book = bookOptional.get();
            Optional<BillDetail> detail = bill.getDetails().stream()
                 .filter(d -> d.getTitle().equalsIgnoreCase(book.getTitle()))
                 .findFirst();
            if(detail.isPresent()){
                detail.get().setAmount(detail.get().getAmount() + 1);
                response.setLineAmount(detail.get().getAmount());
            }
            else{
                BillDetail billDetail = new BillDetail();
                billDetail.setAmount(1);
                billDetail.setTitle(book.getTitle());
                billDetail.setPrice(book.getPrice());
                bill.getDetails().add(billDetail);
            }
            response.setCode(AjaxCartResponse.OK_CODE);
            response.setCartAmount(bill.count());
            response.setTotal(bill.total());
        }
    } catch (ModelException e) {
        e.printStackTrace();
    }
    return response;
}
```

Cambios realizados en este método:
Línea 1: Anotamos este método con *@ResponseBody* para indicar que no queremos que siga el flujo normal del modelo vista controlador, si no que vamos a devolver directamente el cuerpo de la repuesta, que será lo que llegue al cliente, en este caso el JSON correspondiente.

Líneas 4-5: a la anotación *@RequestMapping* le añadimos los tipos producidos y consumidos, que en este caso son ambos *application/json*.

Línea 6: El retorno del método ahora es del tipo *AjaxCartResponse* y los parámetros los vamos a recoger del cuerpo de la petición en vez de los parámetros de la peticición con *@RequestBody*.

Por lo demás, el comportamiento del método sigue siendo el mimos de antes, salvando el hecho que debemos crear y retornar una nueva instancia de *AjaxCartResponse.* Durante el proceso, especificaremos las propiedades de esta instancia en función de si todo sale bien o no (**lineas 26-28**).

En el método de borrar libro del carrito, debemos realizar los mismos cambios que hemos realizado en este método.

12.2.4 Las vistas.

Ya que las peticiones AJAX se hacen en segundo plano y la página no se refresca, tenemos que implementar algún mecanismo para notificar al usuario de los cambios.
Para ello empezamos por modificar nuestra plantilla base para añadir una notificación en el botón del carrito de la barra de navegación:

Para ello aprovecharemos la clase *popover* de bootstrap:

```
1  <form class="navbar-form navbar-right"
2      method="get" action="${pageContext.request.contextPath}/cart/view.form">
3          <button type="submit" class="btn btn-default">Carrito
4              <span class="badge" id="cartCountBadge"
5                  data-container="body"
6                  data-toggle="popover"
7                  data-placement="bottom"
8                  data-content="Carrito actualizado">${sessionScope.cart.count()}
9              </span>
10         </button>
11 </form>
```

Como podemos ver, al span le hemos añadido un id *cartCountBadge* para poder acceder facilmente desde javascript y poder actualizar el número de elementos del carro dinámicamente. Además hemos añadido los *data-attributes* necesarios (http://getbootstrap.com/javascript/#popovers-options) para configurar un *popover*.

También vamos a añadir una nueva zona a nuestra plantilla llamada *extraScripts*, para que las vistas que lo necesiten puedan vincular nuevos scripts (cosa que haremos a continuación):

```
1 <tiles:insertAttribute name="extraScripts" ignore="true" defaultValue=""/>
```

Añadimos esta nueva zona justo antes del cierre de la etiqueta *body*.

Para terminar, editaremos las vistas desde las que se puede añadir un libro al carrito (*book/list.jsp*) y desde las que se puede eliminar un libro del carrito (*cart/view.jsp*)

Veamos los cambios sobre *book/list.jsp* (los cambios en *cart/view.jsp* serán los mismos):

- Modificamos el formulario de añadir libro al carro y le añadadimos un nuevo atributo *name* para poder listar todos los formularios con ese nombre desde javascript.

```
1 <form action="${pageContext.request.contextPath}/cart/add.form"
2           method="post"
3           name="cartAjaxForm">
```

- Vinculamos un nuevo archivo de script (que crearemos después), sobreescribiendo la nueva zona dinámica:

```
1 <tiles:putAttribute name="extraScripts">
2     <script
3         src="${pageContext.request.contextPath}/js/cartController.js">
4     </script>
5 </tiles:putAttribute>
```

12.3 Implementado AJAX: En el cliente.

No solo debemos modificar el comportamiento del servidor para consumir la peticiones AJAX, también debemos modificar la parte del cliente para que este las produzca y consuma sus resultados.
Para ello crearemos el archivo de javascript *web/js/cartController.js* que será el encargado de sobrescribir el comportamiento habitual de los formularios marcados con el nombre de *cartAjaxForm* haciendo que realicen peticiones en segundo plano y procesen sus resultados llegado el momento.

Aprovechando la dependencia de *JQuery* por parte de *Bootstrap*, por lo cual ya tenemos *JQuery* cargado en nuestro sitio web, usaremos sus funcionalidades de acceso al DOM y de realizar peticiones AJAX,

Lo primero que necesitamos es poder sacar la información de un formulario en formato JSON, para ello creamos la siguiente función:

```
/**
 * Get data from a form in JSON notation
 * @param $form to get data from
 * @returns an object with the data
 */
function getFormData($form){
    var unindexed_array = $form.serializeArray();
    var indexed_array = {};

    $.map(unindexed_array, function(n){
        indexed_array[n['name']] = n['value'];
    });

    return indexed_array;
}
```

Básicamente mapeamos las propiedades *name* y *value* del *array* que devuelve *serializeArray()* en un único objeto nuevo.

También necesitamos inicializar los *popovers* de la página y añadiremos una funcionalidad para mostrarlos y ocultarlos programáticamente:

```
var pops = $('[data-toggle="popover"]');
var pop = $(pops[0]);
var timer = undefined;
```

```
function showAndHidePopover(){
    pop.popover('show');
    if(timer){
        clearTimeout(timer);
    }
    timer = setTimeout(function(){
        pop.popover('hide');
    }, 2000);
}
pops.popover();
```

Por último sobrescribiremos el funcionamiento de los formularios:

```
function sendToCart( event ) {
    var formData = getFormData($(this));
    var that = this;
    $.ajax({
        type: "POST",
        url: $(this).attr( 'action' ),
        data: JSON.stringify(formData),
        contentType : "application/json",
        dataType : 'json',
        success: function(data)
        {
            if(data.code && data.code === 1){
                $('#cartCountBadge').html(data.cartAmount);
                if(formData['title']){
                    var tr = that.parentNode.parentNode;
                    if(data.lineAmount > 0){
                        $(tr.children[3]).html(data.lineAmount);
                        $(tr.children[4]).html(data.subtotal.toFixed(2));
                    }
                    else{
                        tr.remove();
                    }
                    $('#tdTotal').html(data.total.toFixed(2));
                }
                showAndHidePopover();
            }
        }
    });
```

```
29
30      event.preventDefault();
31  }
32
33  $('form[name="cartAjaxForm"]').each(function(){
34      $(this).submit(sendToCart);
35  });
```

La función *sendToCart* es la encargada de, mediante el uso de *$.ajax*, realizar la petición AJAX con los datos extraídos del formulario correspondiente (de hecho el valor de *this* dentro de esta función es el propio formulario en sí mismo). La función *$.ajax* espera recibir un objeto con las propiedades *type*, *data*, *contentType*, *dataType* y *success*. Debemos prestar especial atención a esta última, que debe ser una función que *$.ajax* utilizara a modo de *callback* y ejecutará cuando la petición se ejecute con éxito.
En este *callback* comprobaremos que el *code* devuelto con el servidor es 1 (éxito) y, si es el caso, actualizamos el valor del badge de la barra de navegación, y la tabla del carrito si fuese necesario. Por último mostramos los *popovers*.

Fuera de esta función, buscamos todos los formularios con nombre *cartAjaxForm* y les sobreescribimos su método *submit* para redirigirlos a nuesta función *sendToCart.*

Todas esta funcionalidades estaran dentro del evento de carga de la página, para asegurarnos que el HTML se ha cargado correctamente antes de intentar acceder a él. Para eelo envolvemos todo dentro de:

```
1  $(function(){
2          /* Code here */
3  }
```

13. Log de errores: Log4j.

Durante el desarrollo de la aplicación, hemos ido revisando lo errores mediante el uso de la consola en el mismo momento del error, pero una vez que la aplicación esté desplegada en un servidor real, ni vamos a tener acceso a la consola ni mucho menos estaremos delaten de ella en el momento se produzca el error.
Es por esto que debemos configurar nuestra aplicación para que deje constancia de estos errores en forma de archivo de log. Para ello, usaremos Log4j, una biblioteca open source desarrollada por la Apache Software Foundation que permite a los desarrolladores escribir mensajes de registro. Log4j permite filtrar los mensajes en función de su importancia. La configuración de salida y granularidad de los mensajes es realizada en tiempo de ejecución mediante el uso de archivos de configuración externos.

13.1 Configuración

Lo primero que debemos hacer, es configurar log4j para que escriba los errores donde y cuando nos interese. Para esto crearemos el archivo *log4j.properties* dentro de la carpeta *src*:

```
1  # Root logger option
2  log4j.rootLogger=ERROR, stdout, file
3
4  # Redirect log messages to console
5  log4j.appender.stdout=org.apache.log4j.ConsoleAppender
6  log4j.appender.stdout.Target=System.out
7  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
8  log4j.appender.stdout.layout.ConversionPattern=%-4r [%t] %-5p %c - %m%n
9
10 # Redirect log messages to a log file
11 log4j.appender.file=org.apache.log4j.DailyRollingFileAppender
12 #outputs to Tomcat home
13 log4j.appender.file.File=${catalina.home}/logs/LibSpringTilesHibernate.log
14 log4j.appender.LOGFILE.append=true
15 log4j.appender.LOGFILE.DatePattern='.'yyyy-MM-dd
16 log4j.appender.file.layout=org.apache.log4j.PatternLayout
17 log4j.appender.file.layout.ConversionPattern=%-4r [%t] %-5p %c - %m%n
```

13.1.1 Appenders.

En Log4J los mensajes son enviados a una (o varias) salida de destino, lo que se denomina un *appender*.

Existen varios *appenders* disponibles y configurados, aunque también podemos crear y configurar nuestros propios *appenders*.

Típicamente la salida de los mensajes es redirigida a un fichero de texto .log (*FileAppender, RollingFileAppender*), a un servidor remoto donde almacenar registros (*SocketAppender*), a una dirección de correo electrónico (*SMTPAppender*), e incluso en una base de datos (*JDBCAppender*).

Casi nunca es utilizado en un entorno de producción la salida a la consola (*ConsoleAppender*) ya que perdería gran parte de la utilidad de Log4J.

13.1.1 Layouts.

Permite presentar el mensaje con el formato necesario para almacenarlo simplemente en un archivo de texto .log (*SimpleLayout* y *PatternLayout*), en una tabla HTML (*HTMLLayout*), o en un archivo XML (*XMLLayout*).
Además podemos añadir información extra al mensaje, como la fecha en que se generó, la clase que lo generó, el nivel que posee…
En nuestro caso añadirá (append) los mensajes a un fichero (LibreriaSpringTilesHibernate.log), reservando los primeros 4 caracteres para los milisegundos en que se generó el mensaje (%-4r), entre corchetes quién generó el mensaje ([%t]), cinco espacios para la prioridad del mensaje (%-5p), la categoría del mensaje (%c) y finalmente el propio mensaje junto con un retorno de carro (%m%n).

13.2 Uso

Ahora que tenemos log4j configurado, es momento de empezar a usarlo. El procedimiento será el mismo en todas las clases donde queramos logear errores:

- Crearemos un cmpo *static final* de tipo *Logger.*
- Lo usaremos para logear las excepciones dentro de los bloques *catch.*

En nuestro caso, lo haremos dentro de todos los controladores, ya que todas las excepciones suben hasta estas clases. Veamos por ejemplo el caso del método *list* de *BillController*:

```
1 private static final Logger logger = Logger.getLogger(BillController.class);
2
3 @RequestMapping(value = {"/", "/list"} , method = RequestMethod.GET)
4 public ModelAndView list(){
5     ModelAndView model = new ModelAndView("bill/list");
6     try {
7         List<Bill> bills = dao.query(bill -> bill.getDni()
```

```
 8                                          .equalsIgnoreCase(user.getDni()));
 9          model.addObject("bills", bills);
10      } catch (ModelException ex) {
11          logger.error(ex.getMessage(), ex);
12          model.addObject("error", ex);
13      }
14      return model;
15  }
```

Línea 1: Declaramos el campo *logger*.
Línea 11: usamos el campo *logger* para logear la excepción.

14. Bibliografía.

14.1 Blibliografía.

Libros de Spring:

- Learning Spring Application Development [978-1-78398-736-8]
- Pro Spring MVC with Web Flow [978-1-4302-4155-3]
- Spring MVC Cookbook [978-1-78439-641-1]
- Introducing Spring Framework [978-1-4302-6532-0]

Libros de Hibernate:

- Beginning Hibernate, 3rd Edition [978-1-4302-6517-7]
- Hibernate Search by Example [978-1-84951-920-5]
- Java Persistence with Hibernate, 2nd Edition [978-1-6172-9045-9]
- Just Hibernate [978-1-4493-3437-6]

Libros de integración:

- Spring Persistence with Hibernate [978-1-4302-2632-1]

14.2 Webgrafía.

Webs de documentación:

- http://docs.spring.io/spring/docs/current/spring-framework-reference/html/spring-web.html
- https://tiles.apache.org/framework/tiles-jsp/tagreference.html
- http://hibernate.org/orm/documentation/4.2/

Tutoriales:

- https://spring.io/guides/gs/serving-web-content/
- http://frameworkonly.com/spring-mvc-4-tiles-3-integration/
- http://tuhrig.de/making-a-spring-bean-session-scoped/
- http://websystique.com/spring-4-mvc-tutorial/
- http://crunchify.com/simplest-spring-mvc-hello-world-example-tutorial-spring-model-view-controller-tips/
- http://o7planning.org/web/fe/default/en/document/8108/spring-mvc-tutorial-for-beginners

Web de consultas generales:

- http://lmgtfy.com/
- http://stackoverflow.com/

14.3 Código del proyecto.

El código fuente del proyecto puede descargarse de GitHub aquí:

- https://github.com/marcofernandezheras/SpringHibernateTiles