BACHELOR SCIENCE HONOURS IN COMPUTING

SPRING 3 + HIBERNATE

GUÍA PASO A PASO

Documentación realizada para la asignatura ‘Programación Cliente Servidor‘ (2015)

Autores: Garikoitz Aguirre, Jon Ander Cermeño, Daniel Sánchez, Daniel Miñana.

Contenido

Spring, Hibernate y Maven (Conociendo al enemigo).....3

Ventajas y Desventajas (Spring e Hibernate).....5

Estado del arte.....6

Preparando el entorno.....6

Creando nuestro Proyecto.....7

Desarrollando el Proyecto.....14

Estructura de Paquetes y Clases.....14

JSP y XML.....16

Codificación del Proyecto.....16

/dao/AbstractDAO.java.....16

/model/Usuario.java.....17

/dao/Usuario.java.....24

/dao/impl/UsuarioDAOImpl.java.....25

/service/UsuarioService.....32

/service/impl/UsuarioServiceImpl.....33

/controller/UsuarioController.java.....35

/webapp/WEB-INF/views/usuario/ index.jsp.....43

/resources/config.properties.....47

/webapp/WEB-INF/applicationContext.xml.....48

Bibliografía/Webgrafía.....50

Conclusiones.....51

Conclusions.....52

Análisis.....53

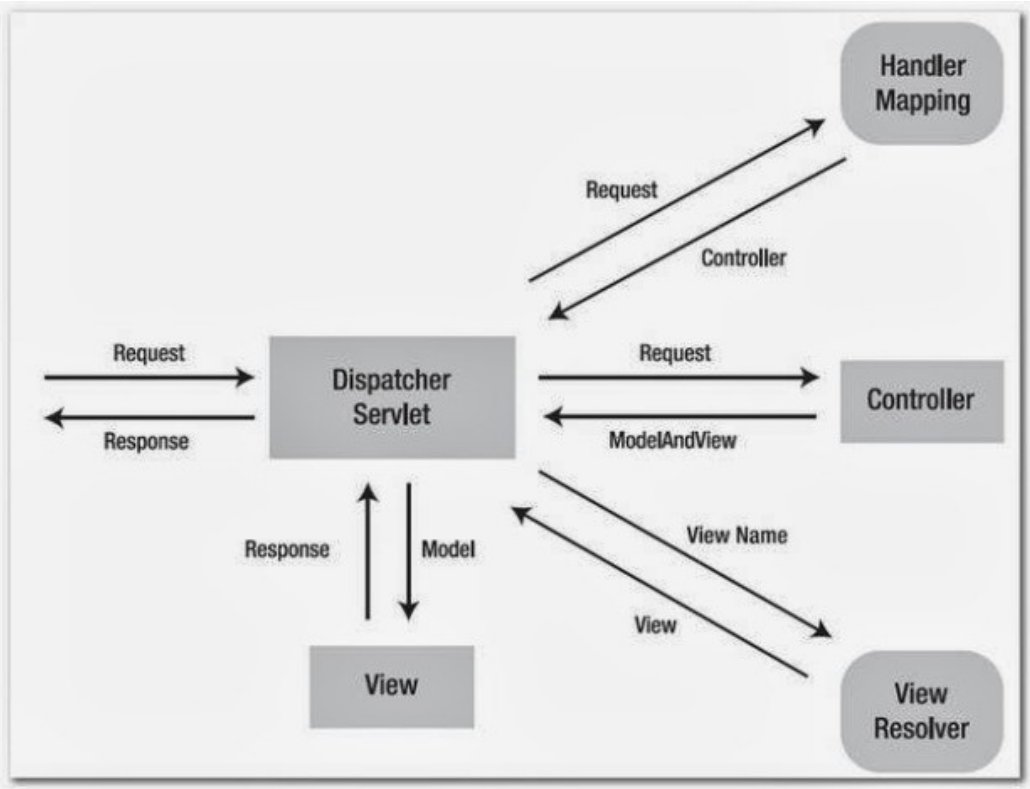
Diagrama entidad-relacion.....53

Spring, Hibernate y Maven (Conociendo al enemigo)

Se hace necesario, como en cualquier toma de contacto que se precie, un breve concepto de cuáles son las funciones y uso los elementos que utilizaremos para la consecución de nuestro objetivo.

Si bien no es objeto teorizar sobre orígenes, autores, versiones y demás parafernalia daremos un breve rodeo sobre las ventajas que nos aportan

Spring: Es un framework en Java que se utiliza para desarrollar proyectos en arquitectura MVC el cual funciona de la siguiente forma:



Se realiza el Request (Petición Http) al Dispatcher Servlet.

Este a través del Handler Mapping localizara el controlador respectivo a la petición.

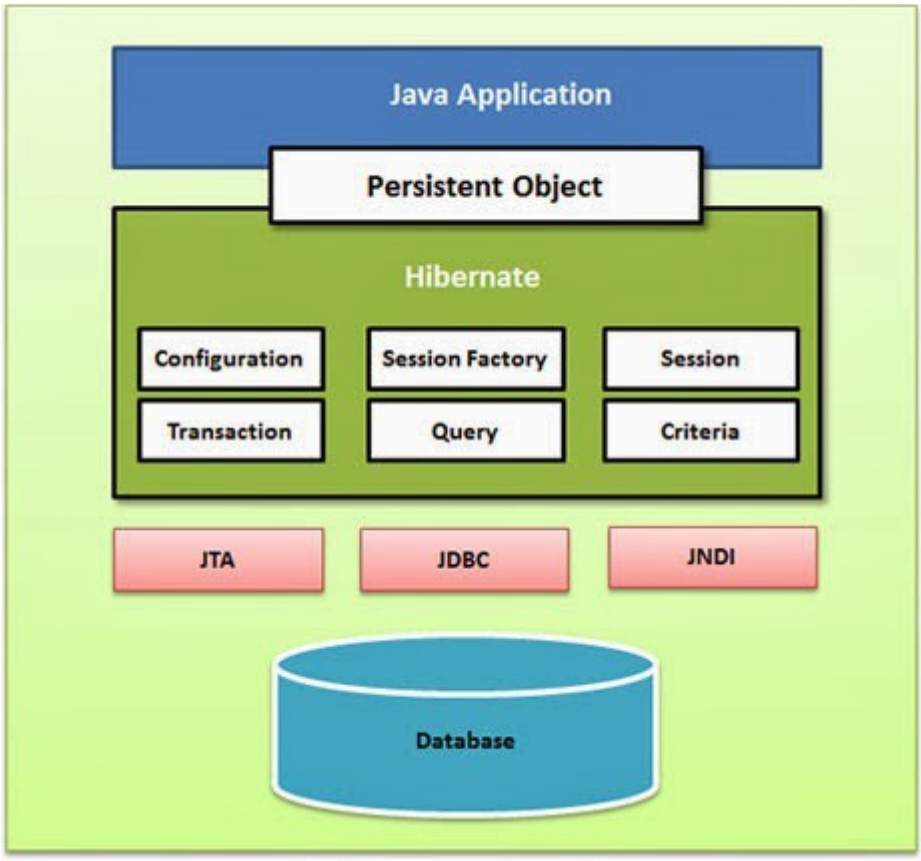
Después el Dispatcher Servlet retornara al request la información para su tratamiento y devolverá mediante el objeto ModelAndView los datos necesarios para la vista (nombre)

Tras ello se consultara al ViewResolver la correspondiente al nombre de la vista jsp obtenida en el paso anterior.

Ahora que se tiene el nombre del jsp y la información necesaria para la vista se obtiene el jsp a partir del View.

Finalmente el Dispatcher Servlet le da la respuesta HTTP también llamada Response a nuestro jsp para que se muestre con la información respectiva.

Hibernate: Es un framework ORM (Object Relational Mapping) que interactuara como pasarela entre la BD y los objetos de nuestra aplicación permitiendo transformar nuestras tablas de la BD en Objetos vivos para nuestra aplicación.



Session Interface: Interface básica de Hibernate.

SessionFactory Interface: Retorna un objeto de tipo Session a la aplicación (Es preciso un objeto para el correcto funcionamiento de la aplicación)

Configuration Interface: Interface es usada para configurar y arrancar Hibernate

Transaction Interface: Interface opcional encargada de abstraer el código para cualquier tipo de transacción.

Query y Criteria Interface: Nos permite crear y ejecutar la queries.

Maven: Se trata de una herramienta que se encarga por nosotros de gestionar y descargar así como de automatizar la integración de las dependencias (.jar) de las que va a hacer uso nuestro proyecto, según lo que definamos en el archivo de configuración *'pom.xml'* el cual estudiaremos más adelante.

Ventajas y Desventajas (Spring e Hibernate)

Spring es uno de los Frameworks más extendidos hoy día para desarrollo web en Java, son muchas las ventajas que nos pueden llevar a pensar en utilizar esta herramienta, citaremos las más relevantes:

1. Gracias a la filosofía MVC y al uso de anotaciones específicas para cada tipo de elemento del código nos ofrece una estructuración de código limpia y claramente entendible
2. Uso de .xml para su configuración, prácticamente todo es configurable desde este tipo de archivos; desde conexiones JDBC, LDAP o acciones específicas.
3. Ofrece una integración sencilla con otras herramientas; siendo un ejemplo Hibernate el cual nos ha ocupado el desarrollo de este proyecto
4. Permite el uso de inyección de dependencias desde un *'Contenedor DI'* y objetos simples según lo especificado en el fichero de configuración
5. Estandarizado; lo cual establece que las aplicaciones desarrolladas en Spring están estructuradas de la misma forma lo cual facilita su uso y entendimiento.

De entre las desventajas aparte de gustos y preferencias de cada programador, sea por citar una el tiempo que requiere cualquier nueva tecnología en ser dominada.

Hibernate es un ORM utilizado en Java para proporcionar persistencia a los objetos hoy día la programación esta exclusivamente orientada a objetos con lo cual no es francamente útil una herramienta que nos permita cumplir este objetivo de forma trasparente. Destacaremos algunas de las principales ventajas

1. Independencia del proveedor, como solución ORM se abstrae del SGBD y permite ser utilizado prácticamente con cualquier tipo de BD
2. Agiliza el código de la capa de persistencia, permitiendo que sea lo más claro y sencillo posible, al ser más claro es también más fácil de mantener
3. Configuración sencilla, la conexión con la BD se realiza de forma sencilla configurando el respectivo archivo .properties y referenciándolo sobre él .xml correspondiente
4. Interactúa con Java a través de anotaciones (Hibernate Annotations)
5. Utiliza su propio lenguaje SQL (HQL), además de su propia API (Criteria) para construir las consultas más fácilmente.
6. Proporciona una capa extra de seguridad ante el acceso a datos.

Entre las desventajas se podría citar quizás la “necesidad” de aprender el lenguaje SQL que Hibernate ofrece; algo depreciable en cuanto a la ventaja de abstracción nos ofrece sobre todo si queremos que nuestro programa sea adaptable a varios SGBD

Estado del arte

Spring 3 es una tecnología puntera muy empleada en la programación web. Spring se utiliza con la finalidad de añadir herramientas y personalización de aplicaciones y servicios web. Aunque en nuestra práctica hemos empleado la versión 3, la última versión estable es la 4.1.1. Spring se suele emplear con el framework Hibernate. Este framework empleado para el mapeo objeto-relacional (ORM) al igual que Hibernate es una tecnología muy puntera. Por último y aunque no lo hemos utilizado el framework Struts se usa junto con estos dos, cada uno centrándose en una parte del modelo MVC (Hibernate para el modelo, Struts para la vista y Spring para el controlador). De hecho el dominio de estos tres frameworks es importante a la hora de contratar programadores web.

Preparando el entorno

Lo primero que necesitamos para comenzar a desarrollar nuestro proyecto en Spring es hacernos con un entorno de desarrollo adecuado para tal fin. En este caso, aunque no como única opción, optaremos por utilizar una modificación de Eclipse llamada Spring Tool Suite preparada para tal fin y que podemos descargar en su página oficial.

http://www.spring.io/tools/sts/all

A fin de mantener la persistencia de datos y las transacciones con los objetos de nuestra aplicación a través del ORM necesitaremos un SGBD. Para esta guía haremos uso de MySQL Server en su versión 5.5 que podremos descargar de la página oficial de MySQL

http://www.mysql.com/downloads/

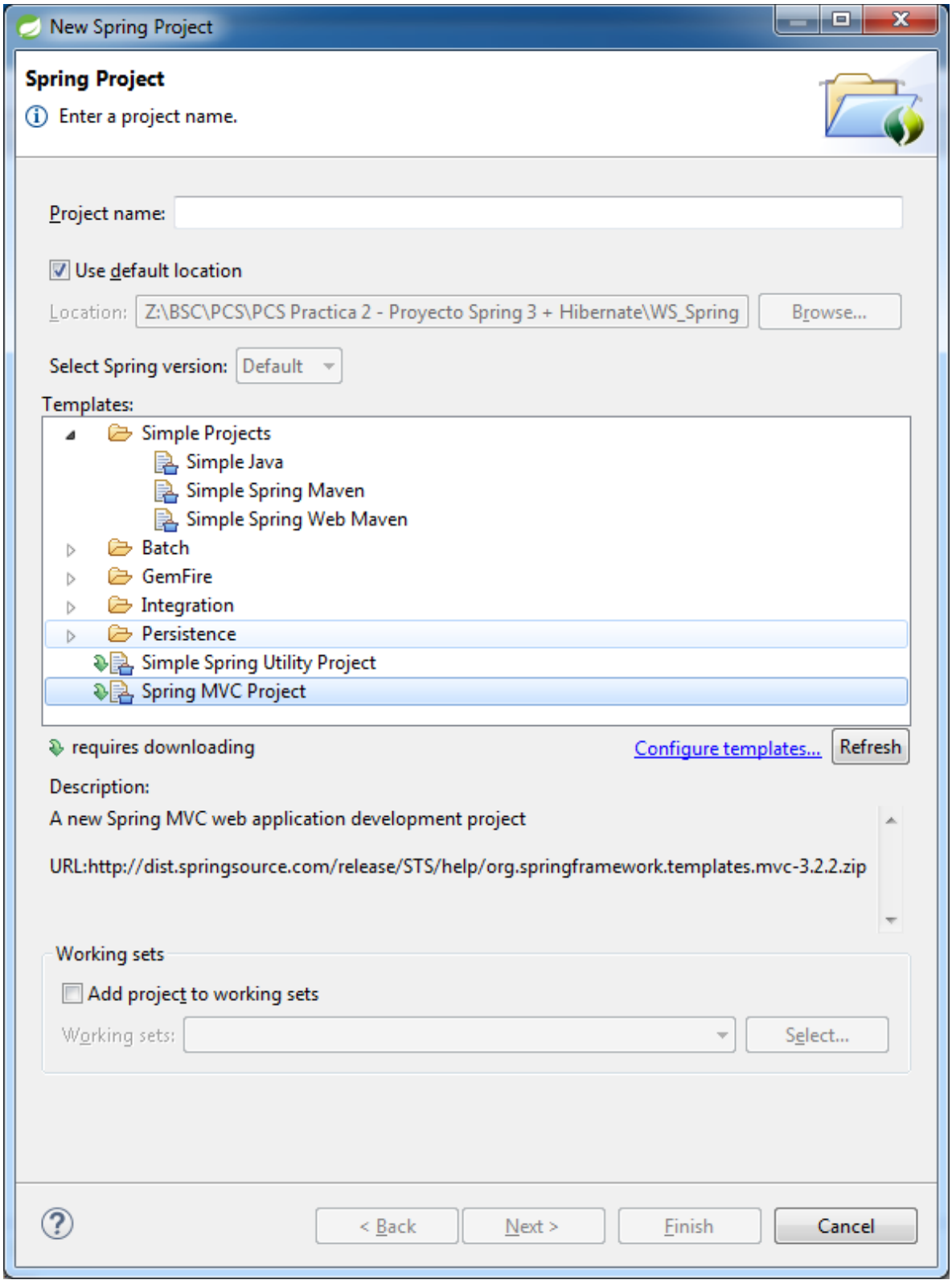
Por último, no necesitaremos tener un servidor Tomcat para ejecutar nuestra aplicación, ya que Spring Tool Suite viene con Pivotal Server incorporado, que es una alternativa a Tomcat.

NOTA: No se contempla en esta guía la configuración e instalación de los requisitos previamente citados.

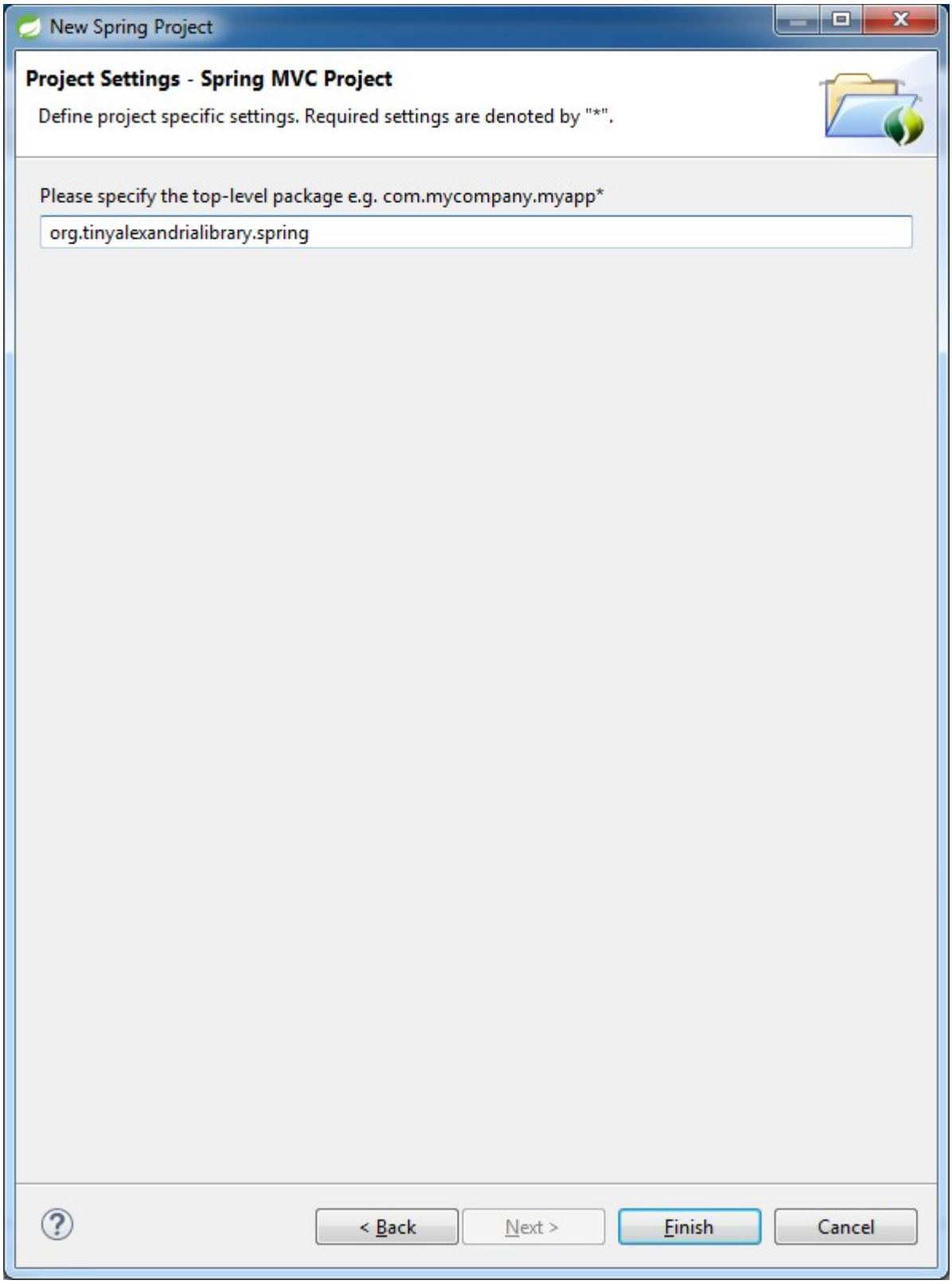
Creando nuestro Proyecto

Crearemos un nuevo proyecto Web a través del menú: File → *New* → *Spring Project* → *Spring MVC Project*

Spring MVC Project

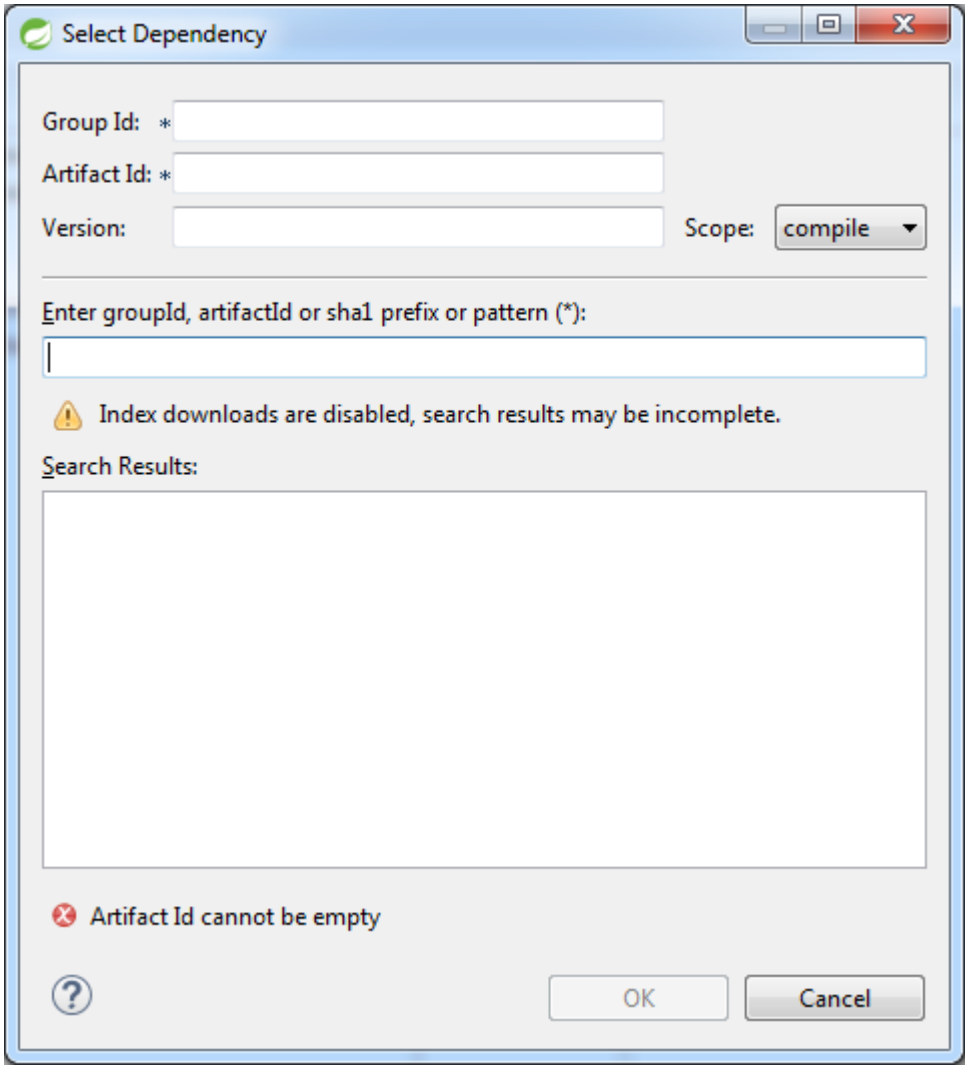


El siguiente paso es especificar el *'top-level package'* según la forma requerida que hará de prefijo para nuestra aplicación.



Un vez creado nuestro proyecto necesitamos alimentarlo con las librerías necesarias para su desarrollo; es aquí donde entra el fichero '*pom.xml*' mencionado al comienzo de la guía y que encontramos en el directorio raíz de nuestro proyecto.

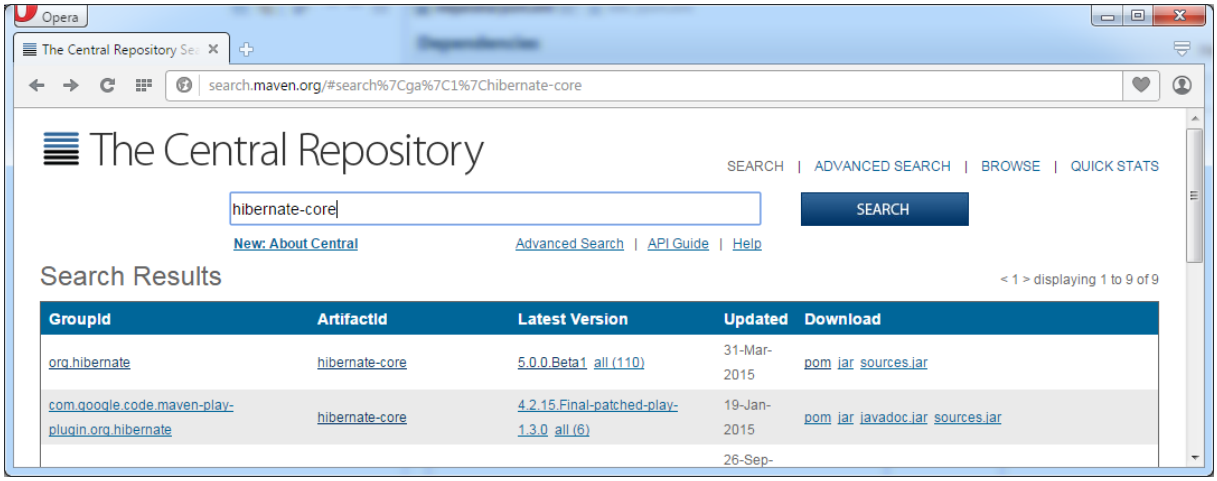
Podemos agregar las librerías de dos formas distintas, la primera de ellas es a través del interfaz ofrecido; mediante la pestaña de '*Dependencies*' y una vez allí mediante la opción Add.

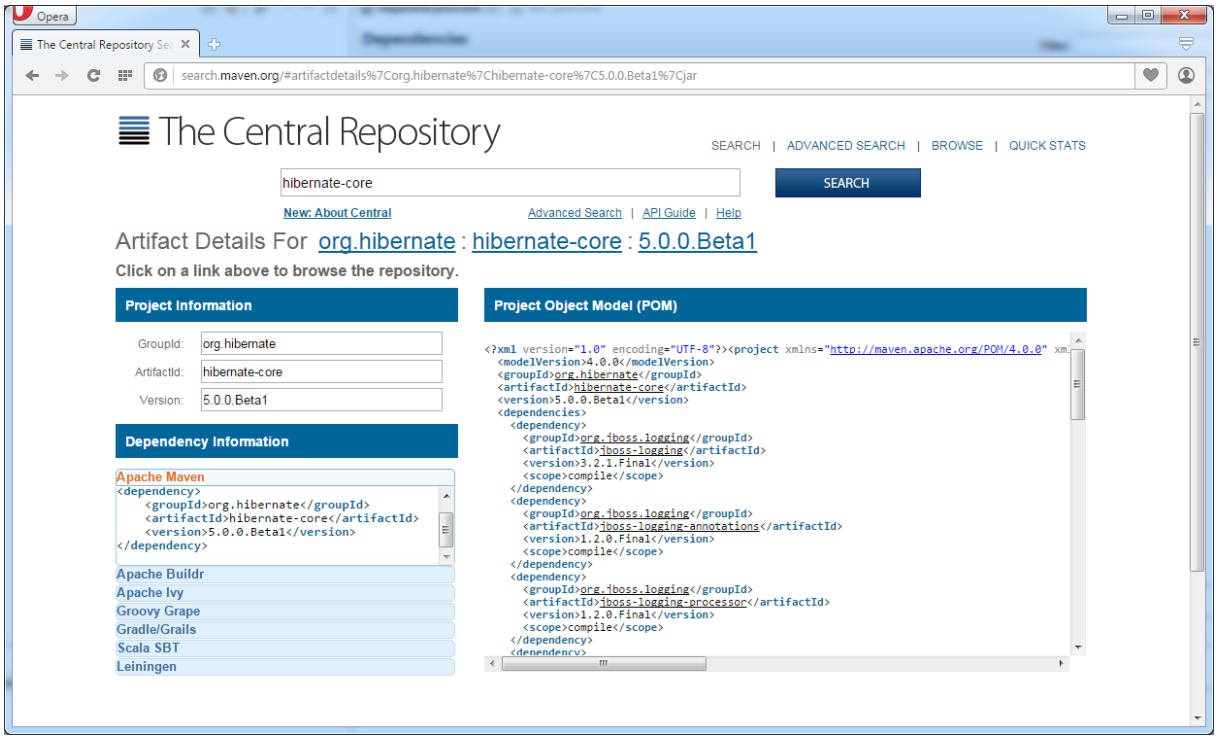


En el cuadro de contexto podremos realizar la búsqueda de dependencias, por ejemplo para el desarrollo del proyecto adjunto necesitamos entre otras agregar librerías de Hibernate para gestionar correctamente el ORM.

Para este caso concreto especificaremos en el campo *'Enter groupId, artifactId o sha prefix or pattern'* Hibernate para que nos localice las librerías disponibles para Hibernate y seleccionar la que más nos convenga.

Otra forma de agregar las librerías es modificando directamente el archivo *'pom.xml'* para lo cual necesitaremos buscar en alguna página de descarga de dependencias para Maven como por ejemplo http://search.maven.org y utilizar el buscador para localizar las dependencias deseadas.





Una vez localizada la dependencia deseada seleccionamos la versión y copiamos el texto que aparece debajo de 'Apache Maven' del tipo:

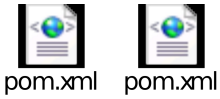
```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.0.0.Beta1</version>
</dependency>
```

De esta forma iremos agregando todas las dependencias a nuestro archivo *'pom.xml'* situándolas entre los *tags* de dependencias de la siguiente forma:

```
<dependencies>
//RESTO DE DEPENDENCIAS ANTES AÑADIDAS
        <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-core</artifactId>
                <version>4.0.1.Final</version>
        </dependency>
</dependencies>
```

De cualquiera de las dos formas iremos agregando las librerías que nuestro proyecto necesite para que una vez guardado Maven se encargue de su descarga e instalación; una vez descargadas las encontraremos en la carpeta '\.m2\repositorio' de nuestro Usuario.

De esta forma nuestro 'pom.xml' debiera quedar similar al adjunto en el link, teniendo en cuenta las diferentes dependencias añadidas, nombres de proyecto y referencias al 'top-level package'.



Comentaremos brevemente algunas de las entradas específicas del archivo 'pom.xml' utilizado para el proyecto en exposición haciendo referencia a sus entradas más significativas.

Como antes hemos comentado necesitaremos hacernos con la dependencia del ORM (Hibernate) que haga las veces de pasarela entre nuestra aplicación y la BD, de esta forma hemos añadido las dependencias correspondientes según la forma indicada previamente y que se reflejan en la entrada abajo expuesta.

```
<!-- Hibernate -->
<!-- <dependency> <groupId>org.hibernate</groupId> <artifactId>hibernate</artifactId>
        <version>3.2.6.ga</version> </dependency> -->
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>4.0.1.Final</version>
</dependency>
<dependency>
        <groupId>org.hibernate</groupId>
```

```
        <artifactId>hibernate-entitymanager</artifactId>
        <version>4.0.1.Final</version>
</dependency>
<!-- <dependency> <groupId>org.hibernate</groupId> <artifactId>hibernate-commons-annotations</artifactId>
<version>3.2.0.Final</version> </dependency> -->
```

Dado que vamos a trabajar contra una BD necesitaremos también informar sobre el tipo y versión a través de su dependencia para que Maven se encargue de la descarga y configuración precisa, podemos ver esta referencia a continuación

```
<!-- MySQL -->
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.9</version>
</dependency>
```

A estas alturas y aun si no se ha observado el archivo 'pom.xml' adjunto entramos en cuenta que no hemos añadido ninguna referencia a dependencias de Spring ni de Maven sobre el cual llevamos tratando un rato.

¿Y cómo es esto posible?, pues bien he ahí las ventajas de trabajar con una de las versiones adaptadas para Spring (Spring Tool Suite), en otro caso hubiésemos tenido que configurar agregar y configurar los repositorios de Maven a nuestro Eclipse (digamos genérico) y tras ello generar un proyecto de tipo 'Maven' en el que añadiríamos las dependencias y plugins necesarios para Spring.

En cambio de esta forma se nos ofrece una preconfiguración adaptada al tipo de proyecto en desarrollo:

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

```
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
</dependency>
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${org.springframework-version}</version>
</dependency>
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>${org.springframework-version}</version>
</dependency>
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>${org.springframework-version}</version>
</dependency>
```

Desarrollando el Proyecto

Llegados a este punto podemos empezar a codificar nuestro proyecto, si bien no hay una forma simple y rápida de mostrar el potencial de este conjunto de herramientas intentaremos explicar a través de referencias al proyecto adjunto su estructura y funcionamiento.

Estructura de Paquetes y Clases

Lo primero de todo necesitamos crear los paquetes necesarios para nuestra aplicación la ruta src/main/java de nuestro proyecto:

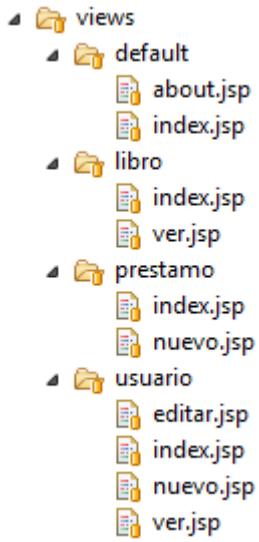
- org.alexandrialibrary.spring.controller
- org.alexandrialibrary.spring.dao
- org.alexandrialibrary.spring.editor
- org.alexandrialibrary.spring.form
- org.alexandrialibrary.spring.model
- org.alexandrialibrary.spring.service
- org.alexandrialibrary.spring.util

El siguiente paso es crear las clases necesarias dentro de nuestros paquetes:

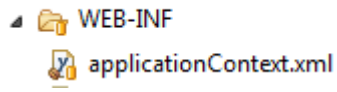
- org.alexandrialibrary.spring.controller
  - AbstractController.java
  - DefaultController.java
  - LibroController.java
  - PrestamoController.java
  - UsuarioController.java
- org.alexandrialibrary.spring.dao
  - AbstractDAO.java
  - EjemplarDAO.java
  - LibroDAO.java
  - PrestamoDAO.java
  - UsuarioDAO.java
- org.alexandrialibrary.spring.dao/impl
  - EjemplarDAOImpl.java
  - LibroDAOImpl.java
  - PrestamoDAOImpl.java
  - UsuarioDAOImpl.java
- org.alexandrialibrary.spring.editor
  - EjemplarEditor.java
  - UsuarioEditor.java
- org.alexandrialibrary.spring.form
  - PrestamoIsDevuelto.java
- org.alexandrialibrary.spring.model
  - Ejemplar.java
  - Libro.java
  - Prestamo.java
  - Usuario.java
- org.alexandrialibrary.spring.service
  - EjemplarService.java
  - LibroService.java
  - PrestamoService.java
  - UsuarioService.java
- org.alexandrialibrary.spring.util
  - PrestamoFormValidator.java
  - UsuarioFormValidator.java

JSP y XML

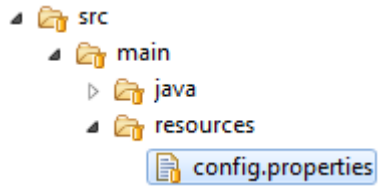
Una vez definidas las clases necesitamos crear las páginas .jsp y los archivos de configuración .xml. Creamos dentro de /src/main/webapp/WEB-INF/views los .jsp con las vistas de la aplicación



Necesitamos crear también el archivo applicationContext.xml en /src/main/webapp/WEB-INF que albergara la configuración del contexto de la aplicación.



Por último en /src/main/resources necesitamos crear el archivo config.properties que contendrá las referencias al JDBC y la configuración para Hibernate.



Codificación del Proyecto

Cumplidos los pasos previos anteriores podemos empezar a codificar nuestro proyecto; explicaremos a continuación las clases más significativas en base a su uso.

/dao/AbstractDAO.java

```
package org.alexandrialibrary.spring.dao;

import org.hibernate.Session;
```

```
import org.hibernate.SessionFactory;

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

/**

 * Clase DAO base.

 */

abstract public class AbstractDAO {

	/**

	 * Sesión que actuará con la capa de persistencia, en este caso el ORM Hibernate.

	 *

	 * No necesita ser instanciada gracias a la anotación @Autowired.

	 * La sesión será accesible antes de que el controlador mande los objetos a la vista.

	 * Después será destruida, así que todo lo que no se haya consultado entonces, no será accesible.

	 */

	@Autowired

	protected SessionFactory sessionFactory;


	protected Session getCurrentSession() {

		return sessionFactory.getCurrentSession();

	}

}
```

A través de esta clase instanciamos la SessionFactory que será la encargada de ejecutar las queries hacia la base de datos.

/model/Usuario.java

```
package org.alexandrialibrary.spring.model;


import java.io.Serializable;

import java.util.List;


import javax.persistence.CascadeType;

import javax.persistence.Column;

import javax.persistence.Entity;
```

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

/**
 * Clase Usuario del modelo. Corresponde a la tabla 'usuario'.
 *
 */
@Entity
@Table(name = "usuario")
public class Usuario implements Serializable {

	private static final long serialVersionUID = 1L;

	/*
	 * Atributos de Usuario, que a su vez serán campos de la tabla 'usuario'.
	 *
	 * Las anotaciones indican a JPA (Java Persistence API) qué papel representa cada uno.
	 */

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	@Column(name = "id")
	private long id;

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

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

```
    @Column(name = "dni", unique = true) // Indica que es una clave única
    private String dni;

    @Column(name = "email", unique = true) // Indica que es una clave única
    private String email;

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

    @OneToMany(targetEntity = Prestamo.class, mappedBy = "usuario", cascade={CascadeType.ALL}, orphanRemoval = true)
    private List<Prestamo> prestamos;

    /* -------- */
    public Usuario() {
        super();
    }

    public Usuario(String nombre) {
        this.nombre = nombre;
    }

    public Usuario(String nombre, String apellidos, String dni, String email, String direccion) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.dni = dni;
        this.email = email;
        this.direccion = direccion;
    }

    public String toString() {
```

```
        return this.nombre + " " + this.apellidos;
    }
    /* -------- */

    /*
     * Getters/Setters
     */

    public long getId() {
        return id;
    }

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

    public String getNombre() {
        return nombre;
    }

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

    public String getApellidos() {
        return apellidos;
    }

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

```
    public String getDni() {
        return dni;
    }

    public void setDni(String dni) {
        this.dni = dni;
    }

    public String getEmail() {
        return email;
    }

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

    public String getDireccion() {
        return direccion;
    }

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

    public List<Prestamo> getPrestamos() {
        return prestamos;
    }

    public void setPrestamos(List<Prestamo> prestamos) {
        this.prestamos = prestamos;
    }
```

```
        /**
         * Se recorre todos sus préstamos y devolverá true si alguno está pendiente.
         */
        public boolean getHasPrestamosPendientes() {
                for (Prestamo prestamo : prestamos) {
                        if (!prestamo.isDevuelto()) {
                                return true;
                        }
                }
                return false;
        }
}
```

NOTA: Para no intensificar el desarrollo de la guía se incluye de la clase lo preciso para la explicación; la clase completa se encontrara junto con el proyecto

Describiremos el funcionamiento de las clases que formaran el modelo de nuestra aplicación; primeramente observamos los import de la clase, el uso de estas clases es el que nos va a permitir asociar nuestra tabla, en este caso 'Usuario' de la BD con la clase Usuario del modelo.

En este caso haremos uso de los import de JPA (Java Persistence API) para dotar a las clases del modelo de la interacción necesaria con el ORM que implementemos.

Para este caso hemos elegido Hibernate, pero gracias al uso de estas clases de las que a su vez extiende Hibernate podemos hacer uso de otro ORM.

De esta forma permitimos un mapeo de las clases del modelo independiente del ORM que utilicemos

Ese proceso es una interacción permitida en este caso gracias a Hibernate y es conocido como: *'Hibernate con anotaciones'*

Describimos a continuación el tipo de anotaciones que podemos encontrar:

- @**Entity**: Referencia a nuestra clase como una entidad que se asociara con una tabla en la Base de Datos.
- @**Table** (name = " "): Indica la tabla a la que representara nuestra clase.
- @**Column** (name = ""): Establece el nombre de los atributos y las características para cada columna.

De la misma forma modificaremos el resto de clases del modelo (*/model*), haciendo uso de la notación de Hibernate para establecer la asociación con las tablas de la BD.

/dao/Usuario.java

```
package org.alexandrialibrary.spring.dao;

import java.util.List;

import org.alexandrialibrary.spring.model.Usuario;

/**
 * Interfaz DAO para Usuarios con los métodos a ser implementados.
 */
public interface UsuarioDAO {

	List<Usuario> getAllUsuarios();

	Usuario getUsuario(long id);

	List<Usuario> getUsuariosByNombreParcial(String nombre);

	boolean isDniAvailable(Usuario usuario);

	boolean isEmailAvailable(Usuario usuario);

	long save(Usuario usuario);

	void update(Usuario usuario);

	void delete(long id);


}
```

El próximo paso es modificar las clases interfaz del DAO, así como sus implementaciones.

La clase anteriormente descrita es una interfaz que permite especificar las acciones a realizar sobre la tabla asociada (Usuario) a partir de la entidad del modelo con la que la asociemos, que en este caso es también Usuario.

Nos queda implementar la interfaz sobre la respectiva clase creada anteriormente en /dao/impl

/dao/impl/UsuarioDAOImpl.java

```
package org.alexandrialibrary.spring.dao.impl;


import java.util.ArrayList;

import java.util.List;


import org.alexandrialibrary.spring.dao.AbstractDAO;

import org.alexandrialibrary.spring.dao.UsuarioDAO;

import org.alexandrialibrary.spring.model.Usuario;

import org.hibernate.Criteria;

import org.hibernate.Hibernate;

import org.hibernate.Query;

import org.hibernate.criterion.Restrictions;

import org.springframework.stereotype.Repository;


/**

 * Implementación de la clase DAO de Usuario.

 *

 * Se comunicará con la BBDD a través de Hibernate.

 */

@Repository

public class UsuarioDAOImpl extends AbstractDAO implements UsuarioDAO {


	/**

	 * Devuelve todos los usuarios.

	 */

	@Override

	@SuppressWarnings("unchecked")
```

```
	public List<Usuario> getAllUsuarios() {
		// Creamos un criterio llamando a la sesión para obtener todos los objetos de Usuario de la BBDD.
		Criteria criteria = this.getCurrentSession().createCriteria(Usuario.class);
		List<Usuario> usuarios = criteria.list();

		/* La colección Préstamos de Ejemplares es Lazy:
		 * no se carga a menos que lo indiquemos explícitamente.
		 */
		for (Usuario usuario : usuarios) {
			// Inicializamos la colección de préstamos
			Hibernate.initialize(usuario.getPrestamos());
		}
		return usuarios;
	}

	/**
	 * Devuelve un usuario concreto.
	 */
	@Override
	public Usuario getUsuario(long id) {
		// Creamos un criterio llamando a la sesión para obtener el objeto de Usuario de la BBDD que se corresponda con una ID concreta.
		Usuario usuario = (Usuario) this.getCurrentSession().get(Usuario.class, id);
		if (usuario != null) {
			/* La colección Préstamos de Ejemplares es Lazy:
			 * no se carga a menos que lo indiquemos explícitamente.
			 */
			Hibernate.initialize(usuario.getPrestamos());
		}

		return usuario;
	}
```

```
	/**
	 * Devuelve todos los objetos usuario cuyo título parcial coincida.
	 *
	 * Ejemplo: "pepe", recogería "Pepe Pérez".
	 *
	 * Utilizaremos la restricción ilike(), que es case insensitive.
	 */
	@Override
	@SuppressWarnings("unchecked")
	public List<Usuario> getUsuariosByNombreParcial(String nombre) {
		// Creamos un criterio llamando a la sesión para obtener todos los objetos de Usuario de la BBDD.
		Criteria criteria = this.getCurrentSession().createCriteria(Usuario.class);
		// Le añadimos la restricción LIKE (case insensitive) con %nombre% como parámetro
		criteria.add(Restrictions.ilike("nombre", "%"+nombre+"%"));
		List<Usuario> usuarios = criteria.list();

		/* La colección Préstamos de Ejemplares es Lazy:
		 * no se carga a menos que lo indiquemos explícitamente.
		 */
		for (Usuario usuario : usuarios) {
			// Inicializamos la colección de préstamos
			Hibernate.initialize(usuario.getPrestamos());
		}
		return usuarios;
	}
	/**
	 * Comprobamos si el DNI del usuario pasado como parámetro está libre.
	 *
	 * Se harán 2 distinciones:
```

```
	 *  - Si la ID de usuario es null, el usuario es nuevo y se buscará a ver si el dni está entre los usuarios existentes.
	 *  - Si la ID de usuario NO es null, el usuario existe y se buscará el dni está entre los usuarios existentes, excluyéndole a él.
	 */
	@Override
	@SuppressWarnings("unchecked")
	public boolean isDniAvailable(Usuario usuario) {
		List<Usuario> usuarios = new ArrayList<Usuario>();

		Query query = null;
		Long id = usuario.getId();
		if (id == null || id == 0) {
			// Es usuario nuevo
			query = (Query) this.getCurrentSession().
					createQuery("from Usuario u where u.dni = :dni");
		} else {
			// Es usuario registrado
			query = (Query) this.getCurrentSession().
					createQuery("from Usuario u where u.dni = :dni and u.id != :id");
		query.setParameter("id", id);
		}

	query.setParameter("dni", usuario.getDni());
	query.setMaxResults(1); // Límite 1
	usuarios = query.list();

	// Si hay algún resultado es que el DNI no está disponible
	return (usuarios.size() == 0);
	}

	/**
```

```
        * Comprobamos si el email del usuario pasado como parámetro está libre.
        *
        * Se harán 2 distinciones:
        *  - Si la ID de usuario es null, el usuario es nuevo y se buscará a ver si el email está entre los usuarios existentes.
        *  - Si la ID de usuario NO es null, el usuario existe y se buscará el email está entre los usuarios existentes, excluyéndole a él.
        */
        @Override
        @SuppressWarnings("unchecked")
        public boolean isEmailAvailable(Usuario usuario) {
                List<Usuario> usuarios = new ArrayList<Usuario>();

                Query query = null;
                Long id = usuario.getId();
                if (id == null || id == 0) {
                        // Es usuario nuevo
                        query = (Query) this.getCurrentSession().
                                        createQuery("from Usuario u where u.email = :email");
                } else {
                        // Es usuario registrado
                        query = (Query) this.getCurrentSession().
                                        createQuery("from Usuario u where u.email = :email and u.id != :id");
                query.setParameter("id", id);
                }

        query.setParameter("email", usuario.getEmail());
        query.setMaxResults(1); // Límite 1
        usuarios = query.list();

        // Si hay algún resultado es que el Email no está disponible
        return (usuarios.size() == 0);
```

```
	}

	/**
	 * Guarda un usuario nuevo.
	 */
	@Override
	public long save(Usuario usuario) {
		return (Long) this.getCurrentSession().save(usuario);
	}

	/**
	 * Actualiza un usuario existente.
	 */
	@Override
	public void update(Usuario usuario) {
		/* Con la ID que viene en el usuario a guardar obtenemos una copia completa
		 * (con sus colecciones cargadas) de lo que existe actualmente en la BBDD.
		 */
		Usuario existingUsuario = getUsuario(usuario.getId());

		/* Sobre el usuario existente, volcamos aquellos atributos que hayan podido cambiar:
		 *  - Nombre
		 *  - Apellidos
		 *  - Dni (se habrá hecho la comprobación anteriormente, ya que debe ser único)
		 *  - Email (se habrá hecho la comprobación anteriormente, ya que debe ser único)
		 *  - Dirección
		 */
		existingUsuario.setNombre(usuario.getNombre());
		existingUsuario.setApellidos(usuario.getApellidos());
		existingUsuario.setDni(usuario.getDni());
```

```
                existingUsuario.setEmail(usuario.getEmail());
                existingUsuario.setDireccion(usuario.getDireccion());

                // Una vez cambiados los atributos pertinentes, guardamos el usuario existente en la BBDD.
                this.getCurrentSession().merge(existingUsuario);
        }

        /**
         * Elimina un usuario concreto.
         */
        @Override
        public void delete(long id) {
                Usuario usuario = getUsuario(id);
                this.getCurrentSession().delete(usuario);
        }
}
```

Las clases en las que implementamos la interfaz han de extender de AbstractDAO e implementar cada una su respectiva interfaz.

De esta forma obtenemos el acceso a la sesión a través del método getCurrentSession(), presente en AbstractDAO, el cual nos asegurará la existencia de la conexión a la BD abierta cuando estemos dentro una clase DAO para hacer las consultas necesarias a la tabla correspondiente.

Otra cosa a tener en cuenta es como se manejan las '*query's*' de forma que estas en vez de realizarse contra la tablas de la base de datos, sino contra la entidad que las representa y contra los atributos de la misma en vez de contra las filas de la tabla.

```
query = (Query) this.getCurrentSession().
        createQuery("from Usuario u where u.email = :email");
```

En el caso mostrado se representa a la entidad (Clase del modelo) identificada como 'Usuario' y a su atributo 'email', no a la tabla.

Las acciones de actualizar, guardar, añadir y eliminar se gestionan también a través del objeto obtenido de getCurrentSession con sus respectivos métodos; getUsuario(), getAllUsuarios(), .save , .update , delete. Estos métodos suelen tener este tipo de nombres, ya que son suficientemente descriptivos y comunes a la mayoría de aplicaciones. Sin

embargo, no hay una norma escrita que obligue a llamarlos así, aunque sí una convención o recomendación de buenas prácticas para desarrolladores de Spring e Hibernate.

```
Usuario usuario = getUsuario(id);

        this.getCurrentSession().delete(usuario);
```

De esta forma y siguiendo las mismas prácticas editaríamos las diferentes interfaces para las clases de nuestro modelo así como sus respectivas implementaciones.

/service/UsuarioService

```
package org.alexandrialibrary.spring.service;


import java.util.List;


import org.alexandrialibrary.spring.model.Usuario;


/**
 * Interfaz Service para Usuarios con los métodos a ser implementados.
 *
 * En las implementaciones se deberá llamar a los métodos pertinentes de las
 * clases DAO.
 */
public interface UsuarioService {


        List<Usuario> getAllUsuarios();


        Usuario getUsuario(long id);


        List<Usuario> getUsuariosByNombreParcial(String nombre);


        boolean isDniAvailable(Usuario usuario);


        boolean isEmailAvailable(Usuario usuario);
```

```
        long save(Usuario usuario);

        void update(Usuario usuario);

        void delete(long id);


}
```

Que actuara como interfaz para la clase UsuarioServiceImpl

/service/impl/UsuarioServiceImpl

```
package org.alexandrialibrary.spring.service.impl;

import java.util.List;

import org.alexandrialibrary.spring.dao.UsuarioDAO;
import org.alexandrialibrary.spring.model.Usuario;
import org.alexandrialibrary.spring.service.UsuarioService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
/**
 * Implementación del servicio de Usuarios.
 *
 * Se anota con @Service y @Transactional para que la sesión que se comunica
 * con Hibernate esté iniciada en este punto.
 *
 * Por norma general la clase del Servicio llama a la clase DAO correspondiente.
 */
@Service
@Transactional
public class UsuarioServiceImpl implements UsuarioService {
        @Autowired
   private UsuarioDAO usuarioDAO;
```

```
    @Override
    public List<Usuario> getAllUsuarios() {
        return usuarioDAO.getAllUsuarios();
    }
    @Override
    public Usuario getUsuario(long id) {
        return usuarioDAO.getUsuario(id);
    }
    @Override
    public List<Usuario> getUsuariosByNombreParcial(String nombre) {
        return usuarioDAO.getUsuariosByNombreParcial(nombre);
    }
    @Override
    public boolean isDniAvailable(Usuario usuario) {
        return usuarioDAO.isDniAvailable(usuario);
    }
    @Override
    public boolean isEmailAvailable(Usuario usuario) {
        return usuarioDAO.isEmailAvailable(usuario);
    }
    @Override
    public long save(Usuario usuario) {
        return usuarioDAO.save(usuario);
    }
    @Override
    public void update(Usuario usuario) {
        usuarioDAO.update(usuario);
    }
    @Override
    public void delete(long id) {
        usuarioDAO.delete(id);
    }
```

```
}
```

Este tipo de clases tendrán el cometido de ser instanciadas por el controlador específico (UsuarioController) e invocar a los métodos del DAO (UsuarioDAO) que necesitemos.

/controller/UsuarioController.java

```
package org.alexandrialibrary.spring.controller;

import java.util.List;

import org.alexandrialibrary.spring.model.Usuario;

import org.alexandrialibrary.spring.service.UsuarioService;

import org.alexandrialibrary.spring.util.UsuarioFormValidator;

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

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.validation.BindingResult;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.servlet.mvc.support.RedirectAttributes;


/**

 * Usuario controller.

 *

 * Contendrá las rutas para todo lo relacionado con usuarios: - Listado de

 * usuarios - Crear un nuevo usuario - Ver un usuario - Editar un usuario -

 * Eliminar un usuario

 *

 */

@Controller

@RequestMapping("/usuario")

// Prefijo para todas las rutas del controlador.

public class UsuarioController extends AbstractController {
```

```
        /**
         * Servicio de usuarios, que llamará al DAO de usuarios, que interactuará
         * con Hibernate para las consultas.
         * No necesita ser instanciado gracias a la anotación @Autowired.
         */
        @Autowired
        private UsuarioService usuarioService;
        /**
         * Validador de usuarios que será utilizado al recibir por post un
         * formulario enviado.
         */
        @Autowired
        private UsuarioFormValidator validator;
        /**
         * Listado de usuarios
         * También permite buscar por nombre parcial de usuarios.
         * Ejemplo: /?nombre=Pepe
         * @param nombre
         * @param model
         * @return
         */
        @RequestMapping(value = "/", method = RequestMethod.GET)
        public String index(
                        @RequestParam(value = "nombre", required = false) String nombre,
                        Model model) {
                logger.info("Iniciamos usuario/index [GET]");
                Usuario usuario = null;
                List<Usuario> usuarios = null;
                if (nombre != null && !nombre.equals("")) {
                        // Si estamos buscando por nombre parcial:
                        // - Guardamos el nombre en el formulario "usuario"
                        // - Obtenemos la lista de usuarios coincidentes
```

```
                usuario = new Usuario(nombre);
                usuarios = usuarioService.getUsuariosByNombreParcial(nombre);
            } else {
                // Si no estamos buscando:
                // - Creamos un formulario "usuario" en blanco
                // - Obtenemos la lista de usuarios por defecto
                usuario = new Usuario();
                usuarios = usuarioService.getAllUsuarios();
            }
            logger.info("Pasamos el formulario y el listado de usuarios a la vista.");
            // Los atributos pasados al modelo serán recibidos por la vista.
            model.addAttribute("usuario", usuario);
            model.addAttribute("usuarios", usuarios);
            return "usuario/index";
        }
        /**
         * Formulario de nuevo usuario [GET]
         * @param model
         * @return
         */
        @RequestMapping(value = "/nuevo", method = RequestMethod.GET)
        public String nuevo(Model model) {
            logger.info("Iniciamos usuario/nuevo [GET]");
            // Creamos un nuevo usuario que actuará como formulario en la vista.
            model.addAttribute("usuario", new Usuario());
            logger.info("Pasamos un usuario nuevo a la vista, para vincularlo al
form.");
            return "usuario/nuevo";
        }
        /**
         * Formulario de nuevo usuario [POST]
         * @param usuario
         * @param result
```

```
         * @param model
         * @param redirectAttributes
         * @return
         */
        @RequestMapping(value = "/nuevo", method = RequestMethod.POST)
        public String nuevo(@ModelAttribute("usuario") Usuario usuario,
                        BindingResult result, Model model,
                        final RedirectAttributes redirectAttributes) {
                logger.info("Iniciamos usuario/nuevo [POST]");
                // Vinculamos el formulario a su validador para, obviamente, validarlo.
                validator.validate(usuario, result);
                if (result.hasErrors()) {
                        // Volvemos a mostrar el formulario, ya que contiene errores
                        logger.info("Formulario con errores, mostramos de nuevo el
formulario.");
                        return "usuario/nuevo";
                }
                logger.info("Formulario correcto! Guardamos el usuario.");
                // El formulario es correcto, así que llamamos al servicio para guardar
                // el usuario
                usuarioService.save(usuario);
                // Generamos un mensaje "flash" que aparecerá en la siguiente página a
                // la que se redirija.
                // El mensaje flash nos informará de que el usuario se ha creado con
                // éxito.
                redirectAttributes.addFlashAttribute(
                                "success",
                                String.format("Usuario '%s %s' creado con éxito.",
                                                usuario.getNombre(), usuario.getApellidos()));
                logger.info("Redireccionamos a usuario/listado [GET]");
                // En vez de mostrar una vista, redireccionamos a la ruta del listado de
                // usuarios.
                return "redirect:/usuario/";
```

```
	}
	/**
	 * Ver un usuario concreto [GET]
	 * @param id
	 * @param model
	 * @return
	 */
	@RequestMapping(value = "/ver/{id}", method = RequestMethod.GET)
	public String ver(@PathVariable Long id, Model model) {
		logger.info("Iniciamos usuario/ver/{id} [GET]");
		// Obtenemos el usuario correspondiente a la ID y lo pasamos a la vista
		Usuario usuario = usuarioService.getUsuario(id);
		model.addAttribute("usuario", usuario);
		return "usuario/ver";
	}
	/**
	 * Eliminar un usuario concreto
	 * @param id
	 * @param model
	 * @return
	 */
	@RequestMapping(value = "/eliminar/{id}", method = RequestMethod.GET)
	public String eliminar(@PathVariable Long id, Model model,
			final RedirectAttributes redirectAttributes) {
		logger.info("Iniciamos usuario/eliminar/{id} [GET]");
		// Obtenemos el préstamo correspondiente a la ID
		Usuario usuario = usuarioService.getUsuario(id);
		if (usuario.getHasPrestamosPendientes()) {
			// Si el usuario tiene préstamos pendientes no permitiremos que sea
			// eliminado
			// Generamos un mensaje "flash" que aparecerá en la siguiente página
			// a la que se redirija.
```

```
                    // El mensaje flash nos informará de que el usuario no se ha podido
                    // eliminar por tener préstamos pendientes.
                    redirectAttributes
                                    .addFlashAttribute(
                                                    "danger",
                                                    String.format(
                                                                    "El usuario <strong>%s %s</strong> tiene préstamos pendientes y no se puede eliminar.",
                                                                    usuario.getNombre(), usuario.getApellidos()));
                    logger.info("Tiene préstamos pendientes, no se puede eliminar.");
                    logger.info("Redireccionamos a usuario/ver/{id} [GET]");
                    // En vez de mostrar una vista, redireccionamos a la ruta de la
                    // ficha del usuario a eliminar.
                    return String.format("redirect:/usuario/ver/%d", id);
            }
            logger.info("No tiene préstamos pendientes, eliminamos el usuario.");
            // En caso de no tener préstamos pendientes se llama al servicio para
            // eliminar el usuario.
            usuarioService.delete(id);
            // Generamos un mensaje "flash" que aparecerá en la siguiente página a
            // la que se redirija.
            // El mensaje flash nos informará de que el usuario se ha eliminado con
            // éxito.
            redirectAttributes.addFlashAttribute("success", String.format(
                            "Usuario <strong>%s %s</strong> eliminado con éxito.",
                            usuario.getNombre(), usuario.getApellidos()));
            logger.info("Redireccionamos a usuario/listado [GET]");
            // En vez de mostrar una vista, redireccionamos a la ruta del listado de
            // usuarios.
            return "redirect:/usuario/";
    }
    /**
```

```
	 * Formulario para editar un usuario concreto [GET]
	 * @param id
	 * @param model
	 * @return
	 */
	@RequestMapping(value = "/editar/{id}", method = RequestMethod.GET)
	public String editar(@PathVariable Long id, Model model) {
		logger.info("Iniciamos usuario/editar/{id} [GET]");
		// Obtenemos el préstamo correspondiente a la ID y lo pasamos a la
		// vista, donde actuará como formulario
		Usuario usuario = usuarioService.getUsuario(id);
		model.addAttribute("usuario", usuario);
		return "usuario/editar";
	}
	/**
	 * Formulario para editar un usuario concreto [POST]
	 * @param usuario
	 * @param result
	 * @param model
	 * @return
	 */
	@RequestMapping(value = "/editar/{id}", method = RequestMethod.POST)
	public String editar(@ModelAttribute("usuario") Usuario usuario,
			BindingResult result, Model model) {
		logger.info("Iniciamos usuario/editar/{id} [POST]");
		// Vinculamos el formulario a su validador para, obviamente, validarlo.
		validator.validate(usuario, result);
		if (result.hasErrors()) {
			// Si tiene errores, lo devolvemos de nuevo al formulario.
			logger.info("Formulario con errores, mostramos de nuevo el
formulario.");
			return "usuario/editar";
		}
```

```
            logger.info("Formulario correcto! Guardamos el usuario.");
            // El formulario es correcto, llamamos al servicio para que actualice el
            // objeto usuario.
            usuarioService.update(usuario);
            logger.info("Redireccionamos a usuario/listado [GET]");
            // En vez de mostrar una vista, redireccionamos a la ruta del listado de
            // usuarios.
            return "redirect:/usuario/";
        }
}
```

Explicaremos ahora las características típicas de una clase de tipo 'controller', haciendo referencia a alguno de sus elementos característicos y al final común entre ellas.

- Observamos en primer lugar todas las referencias a Spring en los import que son vitales para el desarrollo del controlador
- **@Controller:** Identifica la clase como un controlador y la define para que se tratada como tal
- **@RequestMapping:** De esta forma identificamos el recurso que inicializara la ejecución del método que referencia, en el ejemplo mostrado una referencia a *'/nuevo'* desde la página que lo invoca iniciaría la ejecución del método.

```
@RequestMapping(value = "/nuevo", method = RequestMethod.GET)
public String nuevo(Model model) {
        logger.info("Iniciamos usuario/nuevo [GET]");
        model.addAttribute("usuario", new Usuario());
logger.info("Pasamos un usuario nuevo a la vista, para vincularlo al form.");
        return "usuario/nuevo";
        }
```

Siguiendo el mismo patrón definimos todos los métodos necesarios para nuestras clases controlador y mapeándolas con sus *'@RequestMapping'* que las llamaran en el momento que nosotros hayamos definido tal y como se ha explicado.

Por otro lado, varias veces hemos nombrado a las clases Service/DAO alegando que serían instanciadas en el controlador correspondiente o en una clase que necesitara hacer referencia a ellas (por ejemplo, un validador de formularios), pero en ningún momento invocamos a un constructor.

Esto es gracias a la notación '*@Autowired*'. Al ejecutar el proyecto de Spring, éste busca aquellas anotaciones '*@Autowired*' y se encarga de tener una instancia lista cuando sean requeridas. Por lo tanto, nos evitamos tener que instanciar nada. Además, se suele hacer referencia con '*@Autowired*' a la interfaz (por ejemplo UsuarioDAO) y no a su implementación (UsuarioDAOImpl). Sin embargo, se nos abrirá paso a la instancia de la implementación para poder ejecutar los métodos correspondientes y recuperar su resultado.

El siguiente paso es modificar las páginas y relacionarlas con los controladores específicos.

/webapp/WEB-INF/views/usuario/ index.jsp

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib tagdir="/WEB-INF/tags" prefix="t"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<t:base>
        <jsp:attribute name="title">Listado de usuarios</jsp:attribute>
        <jsp:body>
<ul class="breadcrumb">
  <li><a href="<spring:url value="/" />">Inicio</a></li>
  <li class="active">Listado de usuarios</li>
</ul>
<spring:url var="urlFiltrarUsuarioForm" value="/usuario/" />
<form:form action="${urlFiltrarUsuarioForm}" method="get"
                        commandName="usuario" class="form-horizontal">
<div class="no-padding overflow">
        <div class="col-lg-2">
                <a href="<spring:url value="/usuario/nuevo" />"
                                                class="btn                btn-success"><span
class="glyphicon glyphicon-plus"></span> Nuevo usuario</a>
        </div>
        <div class="col-lg-2 col-lg-offset-3">
                <label for="nombre" class="control-label">Buscar por nombre: </label>
        </div>
        <div class="col-lg-3">
```

```
                <form:input type="text" class="form-control" path="nombre" id="nombre"
                                                placeholder="Nombre parcial" />
        </div>
        <div class="col-lg-2">
                <button type="submit" class="btn btn-primary">
                                                <span class="glyphicon glyphicon-search"></span> Buscar usuarios</button>
        </div>
</div>
</form:form>
<div class="panel panel-primary">
  <div class="panel-heading">
    <h3 class="panel-title">Listado de usuarios</h3>
  </div>
  <div class="panel-body">
    <table class="table table-striped table-hover ">
          <thead>
            <tr>
              <th>ID</th>
              <th>Nombre</th>
              <th>Apellidos</th>
              <th>DNI</th>
              <th>Email</th>
              <th>Dirección</th>
              <th>Acciones</th>
            </tr>
          </thead>
          <tbody>
          <c:forEach items="${usuarios}" var="usuario">
                <tr>
                  <td>${usuario.id}</td>
                  <td>${usuario.nombre}</td>
```

```
                    <td>${usuario.apellidos}</td>
                    <td>${usuario.dni}</td>
                    <td>${usuario.email}</td>
                    <td>${usuario.direccion}</td>
                    <td>
                        <a href="<spring:url value="/usuario/ver/${usuario.id}" />"
                                                                    class="btn btn-info btn-
xs">Ver</a>
                        <a href="<spring:url value="/usuario/editar/${usuario.id}" />"
                                                                    class="btn btn-primary
btn-xs">Editar</a>
                        <a href="<spring:url value="/usuario/eliminar/${usuario.id}" />"
                                                                    class="btn btn-${
usuario.hasPrestamosPendientes ? 'default disabled': 'danger'} btn-xs">Eliminar</a>
                    </td>
                </tr>
            </c:forEach>
        </tbody>
    </table>
  </div>
</div>
</jsp:body>
</t:base>
```

En las páginas de vistas diseñaremos el entorno visual de la aplicación así como las acciones entre los reenvíos de formulario

Por otra parte a través de las definiciones de url del tipo le proporcionamos el contextpath correcto a nuestras llamadas a través de la llamada Post-Get correspondiente.

```
"<spring:url value="/usuario/ver/${usuario.id}"
```

Que tendrá que tener su @Request Mapping sobre el método al que llamara en el controlador especifico (UsuarioController.java)

```
/**
     * Ver un usuario concreto [GET]
     *
```

```
     * @param id
     * @param model
     * @return
     */
    @RequestMapping(value = "/ver/{id}", method = RequestMethod.GET)
    public String ver(@PathVariable Long id, Model model) {
        logger.info("Iniciamos usuario/ver/{id} [GET]");

        Usuario usuario = usuarioService.getUsuario(id);
        model.addAttribute("usuario", usuario);

        return "usuario/ver";
    }
```

Por último ya solo nos queda editar los archivos de configuración de Hibernate y Spring.

En nuestro fichero /resources/config.properties se especificarán parámetros convenientes:

/resources/config.properties

```
###################### JDBC Configuration
############################

jdbc.driverClassName=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/alejandria

jdbc.username=root

jdbc.password=


###################### Hibernate Configuration
############################

hibernate.dialect=org.hibernate.dialect.MySQLDialect

hibernate.show_sql=true

hibernate.hbm2ddl.auto=update

#hibernate.hbm2ddl.auto=create-drop

hibernate.generate_statistics=true

hibernate.connection.encoding=utf8

hibernate.connection.useUnicode=true
```

En este archivo definimos los parámetros de configuración para la conexión JDBC con la base de Datos y los parámetros de configuración de Hibernate

- El driver: jdbc.driverClassName=com.mysql.jdbc.Driver
- La BD: jdbc:mysql://localhost:3306/alejandria
- Usuario de conexion: jdbc.username=root
- Password: jdbc.password=

Dentro de la configuración de Hibernate:

- Dialecto de BD al que habrá de adaptarse Hibernate a la hora de traducir las sentencias HQL: hibernate.dialect=org.hibernate.dialect.MySQLDialect

Por último, en nuestro fichero applicationContext.xml, haremos uso de los parámetros especificados en config.properties, para configurar el Bean de la sesión de Hibernate con los parámetros y credenciales necesarios.

/webapp/WEB-INF/applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"

        xmlns:p="http://www.springframework.org/schema/p"
xmlns:tx="http://www.springframework.org/schema/tx"

        xmlns:mvc="http://www.springframework.org/schema/mvc"

        xsi:schemaLocation="http://www.springframework.org/schema/beans

    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

    http://www.springframework.org/schema/context

    http://www.springframework.org/schema/context/spring-context-3.0.xsd

    http://www.springframework.org/schema/tx

  http://www.springframework.org/schema/tx/spring-tx-3.0.xsd

  http://www.springframework.org/schema/mvc

  http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

        <context:annotation-config />

        <context:component-scan base-package="org.alexandrialibrary.spring" />

        <mvc:annotation-driven />

        <context:property-placeholder location="classpath:config.properties" />

        <bean id="messageSource"

        class="org.springframework.context.support.ResourceBundleMessageSource"

                p:basename="Messages" />

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

        <bean id="transactionManager"
        class="org.springframework.orm.hibernate4.HibernateTransactionManager"

                p:sessionFactory-ref="sessionFactory" />

        <bean id="sessionFactory"

                class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">

                <property name="dataSource" ref="dataSource" />

                <property name="hibernateProperties">

                        <props>

                                <prop key="hibernate.dialect">${hibernate.dialect}</prop>
```

```
                <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
                <prop key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
                <prop key="hibernate.connection.CharSet">${hibernate.connection.encoding}</prop>
                <prop key="hibernate.connection.characterEncoding">${hibernate.connection.encoding}</prop>
                <prop key="hibernate.connection.useUnicode">${hibernate.connection.useUnicode}</prop>
            </props>
        </property>
        <property name="packagesToScan">
            <list>
                <value>org.alexandrialibrary.spring.model</value>
            </list>
        </property>
    </bean>
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource"
        p:driverClassName="${jdbc.driverClassName}" p:url="${jdbc.url}"
        p:username="${jdbc.username}" p:password="${jdbc.password}" />
</beans>
```

En este archivo de configuración queda definida la integración del archivo properties que contiene los parámetros de conexión y configuración citados en el punto anterior.

Además, y tal como se puede apreciar en el inicio del fichero XML, se pueden especificar otras cualidades de nuestra aplicación. Por ejemplo <mvc:annotation-driven /> el cual hace referencia al uso de anotaciones para las clases del modelo, rutas del controlador, etc... Si no se hiciera por anotaciones, existe la alternativa de configurar exactamente los mismos parámetros, pero mediante ficheros XML.

Bibliografía/Webgrafía

1. Spring Framework Reference Documentation. Spring IO. Disponible en: http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/

2. Spring CRUD Example using Many to One Mapping. D. Rajput. Disponible en: http://www.dineshonjava.com/2013/08/spring-crud-example-using-many-to-one.html

3. Spring 3.2.5.RELEASE and Hibernate 4 Integration Example Tutorial. L. Gupta. Disponible en: http://howtodoinjava.com/2013/12/08/spring-3-2-5-release-and-hibernate-4-integration-example-tutorial/

4. Spring MVC Flash Attribute Tutorial With Example. V. Patel. Disponible en: http://viralpatel.net/blogs/spring-mvc-flash-attribute-example/

5. Hibernate Error – Initial SessionFactory creation failed.java.lang.NoClassDefFoundError: org/apache/commons/collections/SequencedHashMap. Mkyong. Disponible en: http://www.mkyong.com/hibernate/hibernate-error-initial-sessionfactory-creation-failed-java-lang-noclassdeffounderror-orgapachecommonscollectionssequencedhashmap/

6. Hibernate Questions. Stackoverflow (múltiples consultas). Disponible en: http://stackoverflow.com/questions/tagged/hibernate

7. Spring MVC Questions. Stackoverflow (múltiples consultas). Disponible en: http://stackoverflow.com/questions/tagged/spring-mvc

8. JSP Questions. Stackoverflow (múltiples consultas). Disponible en: http://stackoverflow.com/questions/tagged/jsp

Conclusiones

A día de hoy es muy común trabajar con frameworks, tanto por libre como en empresas, ya que su cometido es facilitar y agilizar el trabajo, además de sentar una base de buenas prácticas para desarrolladores y conseguir un tipo de escritura de código homogénea y reconocible por otros desarrolladores.

Esto suele ser de vital importancia para las empresas a la hora de contratar nuevos trabajadores, ya que les asegura que el código de los trabajadores actuales será mantenible por el de futuras incorporaciones.

Sin embargo, Spring tiene una característica (más por ser Java que por el framework en sí) que no se ajusta precisamente al ideal de framework web: Escribir código se vuelve muy tedioso y repetitivo. La cantidad de líneas (tanto de código como de configuraciones en XML) que hay que escribir incluso para una funcionalidad pequeña es enorme. Otros frameworks disponen de sencillas herramientas para generar código común a diferentes funcionalidades o código que siempre va a estar presente, liberando al programador de dicha tarea.

Por otro lado, Hibernate parece suficientemente potente como para suplir nuestras necesidades, y la gestión de errores en la traducción de sentencias HQL → SQL es bastante intuitiva.

Finalmente, la gestión de dependencias con Maven resulta en un aprendizaje lleno de conflictos y de gran pérdida de horas, al menos en el caso que nos ocupa. El compañero que se dedicó a investigar invirtió fácilmente 20 horas en conseguir una configuración de dependencias en la que no existiera conflicto interno. La parte buena es que una vez que se consigue configurar el fichero `pom.xml` correctamente, el problema desaparece y se puede reutilizar en otros futuros proyectos.

Conclusions

Nowadays working with frameworks is very common, as a freelance and also developing in a company. That's because the frameworks goal is to make it easier and provide a bunch of useful “good practices” to the developer. Therefore, the developers would achieve a well-performed, readable code writing.

This feature is very important for the companies bussiness logic, in order to hire new people, because it ensures them to reach a maintanable code for the future developers of the company.

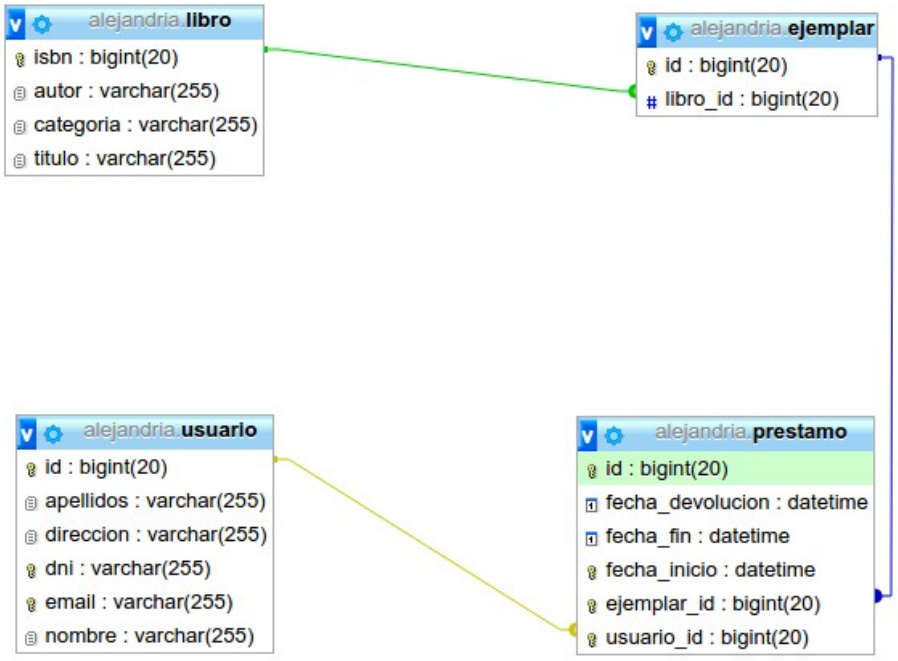
However, Spring has peculiarity (maybe because it's Java, rather than other reason) which doesn't really fit in the ideal framework logic: It's hard and repetitive to write code with it. There are dozens or hundreds of code lines (and Xml config lines) to be writen in order to make a small, reliable funcionality. Other frameworks offer tools for repetitive code generation, making it easier to the developer and freeing him from the hard and boring code writing.

On the other hand, Hibernate is powerful enough to fill the developer needs. And the HQL → SQL traduction exception handler is verbose enough to help (maybe we cannot tell the same if we talk about the Spring Xml configuration exception handlers)

Finally, the dependendy management with Maven turned out to be a learning curve full of conflicts and obstacles, at least the case we met during this groupal exercise. The student in charge of resolving the maven dependencies spent nearly 20 hours trying to make an stable bunch of non-conflictive dependencies. The good thing is that, once you make it and write a proper `pom.xml` file, you can reuse it in other Spring 3 + Hibernate projects.

Análisis

Diagrama entidad-relacion



Atributos:

Libro:

- **ISBN**: Primary Key – varchar
- **Autor**: varchar
- **Categoría**: varchar
- **Título**: varchar

Ejemplar:

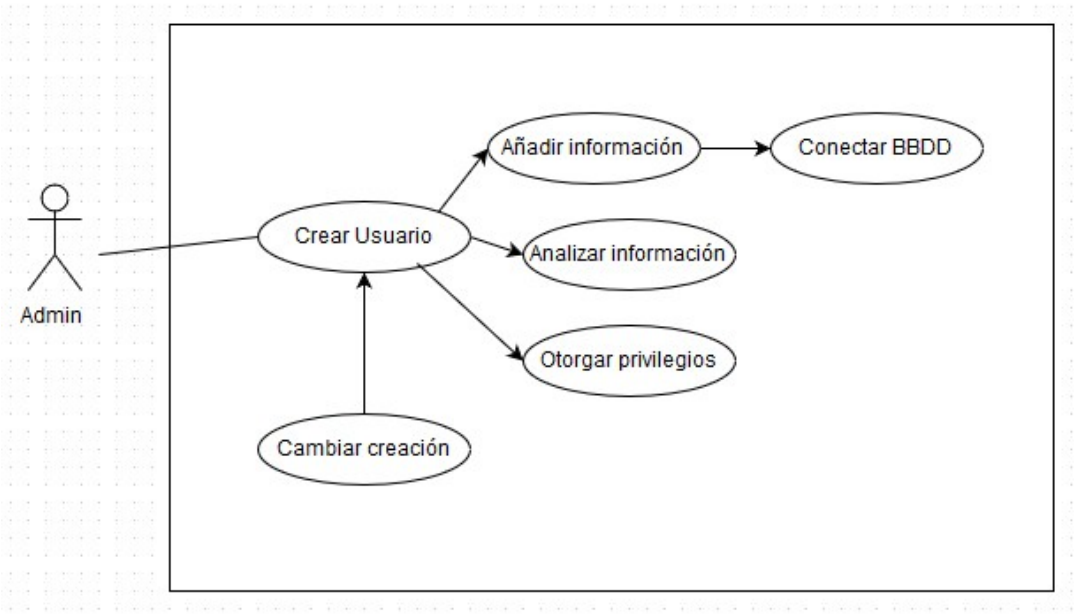
- **ID**: Primary Key – bigint
- **Libro_id**: Foreign Key [ISBN de Libro] – bigint

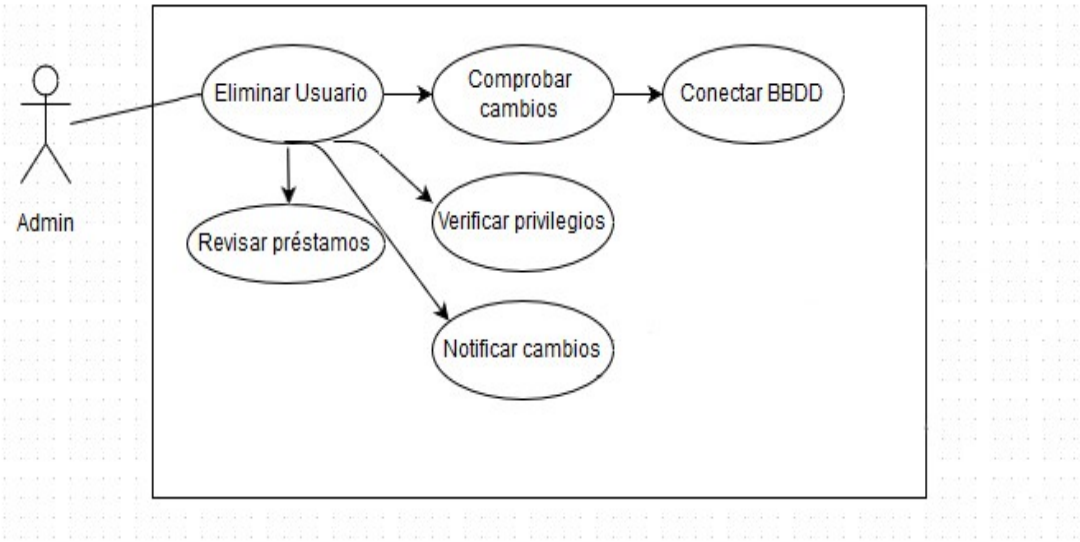
Préstamo:

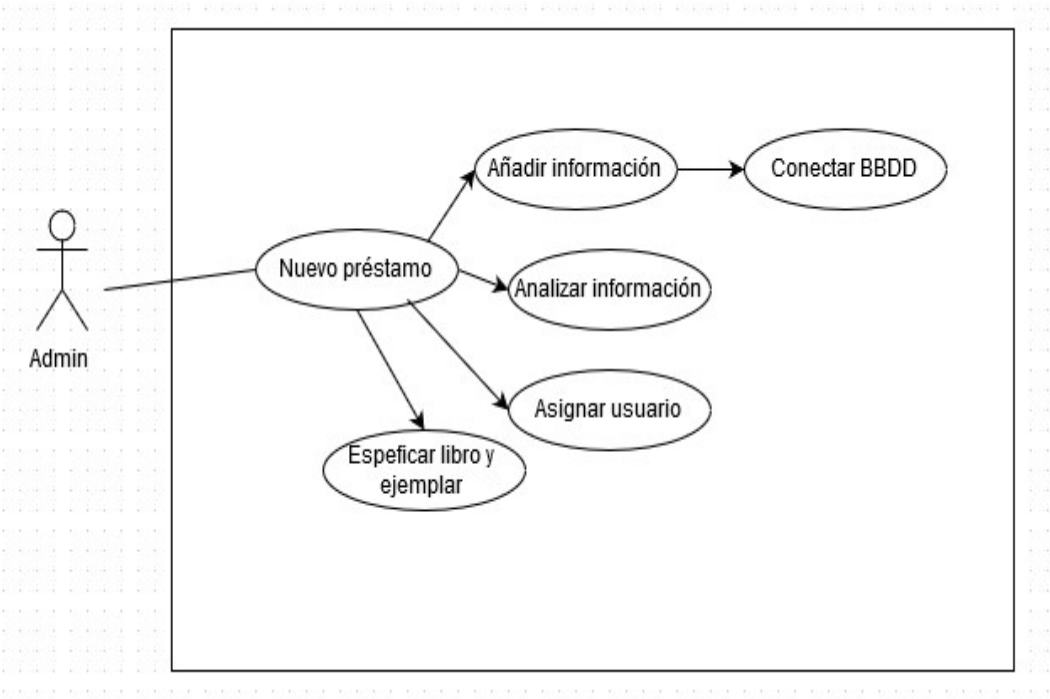
- **ID**: Primary Key – bigint
- **Fecha_devolucion:** Date
- **Fecha_fin**: Date
- **Fecha_inicio**: Date
- **Ejemplar_id**: Foreign Key [ID de Ejemplar]
- **Usuario_id**: Foreign Key [ID de Usuario]

Usuario:

- **ID**: Primary Key – bigint
- **Apellidos**: varchar
- **Dirección**: varchar
- **DNI**: Unique - varchar
- **Email**: Unique - varchar
- **Nombre**: varchar

Caso de uso: Crear usuario
Breve descripción: Creación de usuarios para nuestra Biblioteca, los cuales tendrán derecho a pedir prestados libros. Para ingresar nuevos usuarios al sistema el encargado tendrá que ser un administrador del sistema. Una vez creado se quedará registrado el usuario en nuestra Base de Datos en la cual se podrá ver información relacionada a ellos
Diagrama: 
Actores principales: Administrador
Actores secundarios: Usuario: cuando quiera solicitar libros en la Biblioteca, como saber si hay ejemplares disponibles del libro que quiera y poder hacer préstamos.
Eventos de activación: Cuando se desea añadir nuevo usuario al sistema para que dicha persona pueda pedir prestados libros de nuestra Biblioteca, como saber más información acerca de ellos.
Condiciones de terminación: El administrador ingresa los datos y se añaden al sistema, dando a entender que el ingreso al sistema del nuevo usuario ha sido satisfactorio.
Escenarios: En caso de haber un usuario con mismos identificativos habrá que cambiarlos por otros para poder llegar a inscribirse en el sistema.

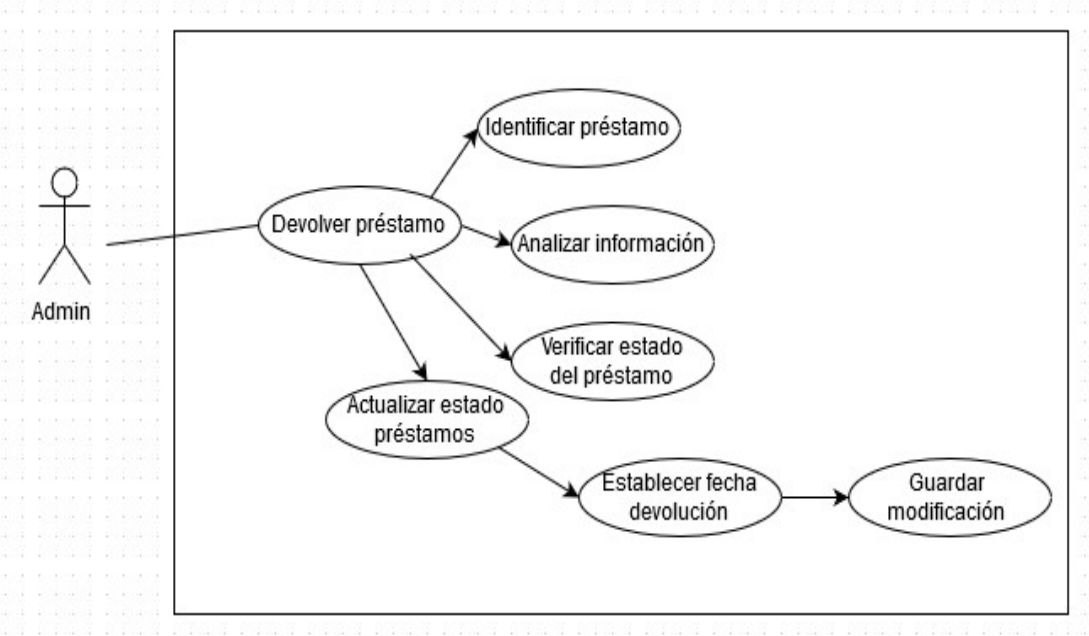
Caso de uso: Eliminar usuario
Breve descripción: Eliminación de un usuario que ya no sirva en el sistema, o al menos en el momento en el que se llega a este punto. El administrador será el encargado de llevar a cabo el cambio. Además habrá que ver si hay préstamos pendientes, si no es así no se podrá hacer dicho cambio.
Diagrama: 
Actores principales: Administrador
Actores secundarios: Usuario: cuando se llega a la conclusión de que ya no es de utilidad en el sistema.
Eventos de activación: El administrador decide deshacerse del algún usuario que ya no es necesario por el motivo que haya visto conveniente en el sistema
Condiciones de terminación: • El administrador realiza la eliminación satisfactoriamente el usuario del rol que sea del sistema. • Si el usuario tiene préstamos pendientes avisando de que no se podrá llevar a cabo hasta que no se solucione.

Caso de uso: Crear préstamo
Breve descripción: Como parte de la gestión de los préstamos podremos crear nuevos, para así poder entregar ejemplares de libros a los usuarios que los soliciten. De dicha gestión se podrá hacer cargo el administrador que lleve a cabo la gestión y se otorgará el préstamo dependiendo el usuario, el libro que se solicite, como su ejemplar. En caso no haber ningún error se guardará dicho préstamo.
Diagrama: 
Actores principales: Administrador
Actores secundarios: Usuario: cuando decida hacer la solicitud de nuevo préstamo en la Biblioteca para pedir un ejemplar de libro solicitado.
Eventos de activación: Cuando el administrador debe realizar el préstamo, seleccionando el usuario, se especifica el libro, como también el ejemplar del préstamo solicitado.
Condiciones de terminación: Cuando todos los ejemplares del libro están ya prestados con anterioridad al préstamo actual.
Escenarios: Que haya errores introducidos en el formulario de creación de préstamo.

Caso de uso: Devolver préstamos

Breve descripción: Se decide devolver el préstamo, por tanto la fecha de devolución será el momento en el que se haga dicho cambio en el sistema. De ello se podrá hacer cargo el administrador y obtendrá el préstamo del ejemplar de libro e indicará que se ha devuelto para que en un futuro se pueda volver a prestar el ejemplar por otro usuario que lo quiera solicitar.

Diagrama:



Actores principales: Administrador

Actores secundarios: Usuario: cuando se ha hecho efectiva la devolución y se puedan generar nuevos préstamos.

Eventos de activación: El administrador dará por devuelto el préstamo por el motivo que se haya decidido, de forma que quedará como devuelto el préstamo y será actualizado en el sistema.

Condiciones de terminación:

- Cuando se completa la devolución del préstamo y el sistema sigue funcionando correctamente, de forma que en el futuro se pueda volver a prestar el ejemplar del libro correspondiente.
- Que la ID del préstamo nos indique que el préstamo que se ha obtenido ya está devuelto.

DIAGRAMA DE CLASES