UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE SISTEMAS INFORMÁTICOS

PROYECTO DE FIN DE GRADO

GRADO EN INGENIERÍA DEL SOFTWARE

DESARROLLO DE UNA APLICACIÓN WEB CON SPRING FRAMEWORK PARA UN GESTOR DE UN RECETARIO

Autor: FEDERICO JULIÁN GUTIÉRREZ FARAONI
Tutor: ADOLFO YELA RUIZ

CURSO 2014/2015

CONTENIDO

I. INTRODUCCIÓN

A. RESUMEN

Este proyecto tiene como intención llevar a cabo el desarrollo de una aplicación basada en tecnologías Web utilizando Spring Framework, una infraestructura de código abierto para la plataforma Java. Se realizará primero un estudio teórico sobre las características de Spring para luego poder implementar una aplicación utilizando dicha tecnología como ejemplo práctico.

La primera parte constará de un análisis sobre las características más significativas de Spring, recogiendo de esta forma información sobre todos los componentes del framework necesarios para desarrollar una aplicación genérica. El objetivo es descubrir y analizar cómo Spring facilita la implementación de un proyecto con arquitectura MVC y cómo permite integrar seguridad, internacionalización y otros conceptos de forma transparente.

La segunda parte, el desarrollo de la aplicación web, sirve como demostración práctica de cómo utilizar los conocimientos recogidos sobre Spring. Se desarrollará una aplicación que gestiona un recetario generado por una comunidad de usuarios. La aplicación contiene un registro de usuarios que deberán autenticarse para poder ver sus datos personales y modificarlos si lo desean. Dependiendo del tipo de usuarios, tendrán acceso a distintas zonas de la aplicación y tendrán un rango distinto de acciones disponibles. Las acciones principales son la visualización de recetas, la creación de recetas, la modificación o eliminación de recetas propias y la modificación o eliminación de recetas de los demás usuarios. Las recetas constarán de un nombre, una descripción, una fotografía del resultado, tiempos estimados, dificultad estimada, una lista de ingredientes y sus cantidades y finalmente una serie de pasos con fotografías demostrativas si se desea añadir. Los administradores, un tipo específico de usuarios, podrán acceder a una lista de usuarios para monitorizarlos, modificarlos o añadir y quitarles permisos.

B. SUMMARY

The purpose of this project is the development of an application based on Web technologies with the use of Spring Framework, an open-source application framework for the Java platform. A theoretical study on the characteristics of Spring will be performed first, followed by the implementation of an application using said technology to show as object lesson.

The first part consists of an analysis of the most significant features of Spring, thus collecting information on all components of the framework necessary to develop a generic app. The goal is to discover and analyze how Spring helps develop a project based on a MVC architecture and how it allows seamless integration of security, internationalization and other concepts.

The second part, the development of the web application, serves as a practical demonstration of how to use the knowledge gleaned about Spring. An application will be developed to manage a cookbook generated by a community of users. The application has a set of users who have to authenticate themselves to be able to see their personal data and modify it if they wish to do so. Depending on the user type, the user will be able to access different parts of the application and will have a different set of possible actions. The main possible actions are: creation recipes, modification or deletion of owned recipes and the modification and deletion of any recipe. The recipes consist its name, a description, a photograph, estimated times and difficulties, a list of ingredients along with their quantities and lastly a series of steps to follow along with demonstrative photographs if desired; and other information such as categories or difficulties. The administrators, a specific type of users, will have access to a list of users where they can monitor them, modify them or grant and remove privileges.

C. OBJETIVOS

El objetivo principal se puede definir como el estudio de Spring Framework y la implementación de una aplicación completa.

Este objetivo puede dividirse en dos partes distinguidas: el estudio teórico de Spring Framework y la implementación de una aplicación basado en los conocimientos adquiridos en el estudio teórico.

Los objetivos de la primera parte son:

- ✓ Estudio de la instalación y configuración de Spring: Se estudia cómo empezar a utilizar Spring, aprendiendo a añadir módulos necesarios. Se estudiará el uso de Spring Boot para la configuración.
- ✓ Estudio del uso de Inversión de Control de Spring: Se estudia la utilización de inyección de dependencias como inversión de control y se localizan las ventajas de esto frente a las aplicaciones tradicionales.
- ✓ Estudio del patrón MVC aplicado a Spring: Se estudia cómo aplica Spring el patrón Modelo-Vista-Controlador y se analizan los distintos componentes y cómo colaboran entre sí.
- ✓ Estudio de la aplicación de i18n con Spring: Se estudian las facilidades que ofrece Spring a la hora de implementar una aplicación que necesite localizar al usuario para ofrecer formatos, lenguajes y otras distinciones.
- ✓ Estudio de la seguridad en Spring: Se estudia el módulo Spring Security y cómo es aplicable al resto del programa para autenticar usuarios y restringir accesos.
- ✓ Estudio de otros aspectos en Spring: Se estudian otras características genéricas y muy utilizadas en aplicaciones web como la subida y descarga de archivos, montajes de APIs REST y validación de formularios.
- ✓ Estudio del despliegue de aplicaciones en Spring: Se estudia como subir una aplicación a un entorno de producción una vez se haya terminado.

El objetivo de la segunda parte es aplicar los objetivos de la primera a un aplicación completa. Se utilizará una aplicación web con registro y autenticación de usuarios que serán capaces de crear, visualizar y modificar recetas. La aplicación estará contenida en un servidor Web Apache Tomcat. Los datos de los usuarios, las recetas y el resto de modelos utilizados serán almacenados en una base de datos MySQL.

II. ESTUDIO TEÓRICO

1. SPRING

1.1. Introducción

Spring es un framework de desarrollo de código libre para la plataforma Java, por lo tanto cualquier sistema operativo con una máquina virtual de Java puede ejecutar aplicaciones desarrolladas con este framework. Su aspecto modular lo hace flexible y configurable para cualquier tipo de aplicación.

1.2. Versiones

Una primera versión del framework fue publicada en octubre del 2002 por Rod Johnson. En junio del 2003 publicó una versión bajo la licencia Apache 2.0 (código libre), a partir de esta todas las versiones fueron publicadas con esta licencia. En marzo del 2004 lanzó la versión 1.0, la cual fue más popularmente conocida. El hecho de ser un proyecto open-source bien documentado con la utilización de Javadocs y su adaptación a los estándares J2EE impulsó el uso de muchos desarrolladores.

Spring 2.0 se publicó en octubre del 2006, Spring 3.0 en diciembre del 2009 y Spring 4.0 en diciembre del 2013, cada una de las versiones con sus respectivas subversiones. Actualmente la versión estable es la 4.1.4, lanzada en diciembre de 2014. La versión 4 ofrece novedades como compatibilidades con Java 8, Groovy 2 Java EE7 y WebSocket.

1.3. Características

Como se ha mencionado anteriormente, Spring Framework se compone de múltiples módulos que abarcan una gran variedad de servicios que facilitan la implementación de elementos de una aplicación empresarial.

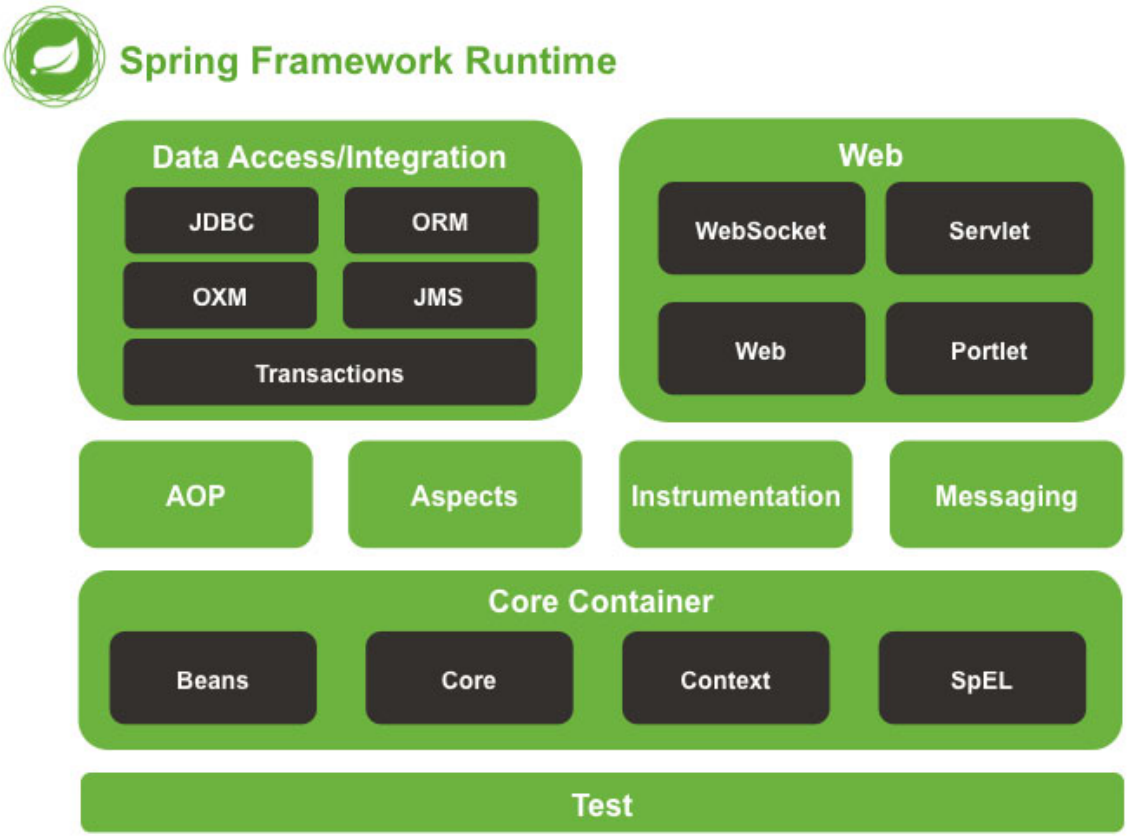
El módulo principal de Spring es el Spring Core Container, el proveedor de contenedores de Spring (BeanFactory y ApplicationContext). Algunos de los módulos estudiados en este proyecto son:

- Spring MVC: Basado en HTTP y servlets para la extensión y personalización de aplicaciones web y servicios REST.
- Contenedor de inversión de control (IoC): Configura los componentes de la aplicación mediante inyección de dependencias.

- Acceso a datos: Trabaja con bases de datos relacionales utilizando ORM (object-relational mapping) y JDBC (Java DataBase Connectivity).
- Autenticación y autorización: Utiliza estándares, protocolos, herramientas y prácticas para la configuración de seguridad y acceso.
- Gestión de transacciones: gestiona y coordina las transacciones para los objetos Java.
- Convención sobre Configuración: ayuda a generar rápidamente aplicaciones configurando Spring automáticamente.

Otros módulos que se pueden distinguir son: Mensajes, Acceso remoto, Administración Remota y Testing.

Actualmente, los módulos de Spring se pueden resumir en el siguiente diagrama:



(Imagen de: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/images/spring-overview.png)

1.4. Instalación y requisitos

Los siguientes aspectos son fundamentales para desarrollar una aplicación:

- Java: Spring 4 soporta desde Java SE 6 a Java 8
- Gestión y construcción: Maven o Gradle
- IDE: Spring Tool Suite (basado en Eclipse) es el IDE oficial aunque se pueden desarrollar aplicaciones en otros IDEs como IntelliJ IDEA con ayuda de plugins.
- Servidor: Versiones de Tomcat de 5 a 8
- BBDD: MySQL

Con una combinación del software mencionado anteriormente se puede empezar a desarrollar con Spring. Spring se integra al proyecto como dependencia a través de Maven o Gradle. En este proyecto se utiliza Maven y STS (Spring Tool Suite) debido a que hay mayor adaptación y documentación con estas tecnologías.

Por lo tanto, los módulos de Spring que se deseen agregar en el proyecto se añadirán al pom.xml (Project Object Model) para que Maven lo integre. Las dependencias se añaden entre las etiquetas `<dependencies></dependencies>`. Un ejemplo de dependencia es el siguiente:

```
<dependencies>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-web</artifactId>
	</dependency>
</dependencies>
```

2. SPRING BOOT

2.1. ¿Qué es Spring Boot?

Como se puede apreciar, Spring tiene una amplia cantidad de módulos que implican multitud de configuraciones. Estas configuraciones pueden tomar mucho tiempo, pueden ser desconocidas para principiantes y suelen ser repetitivas. La solución de Spring es Spring Boot, que aplica el concepto de Convention over Configuration (CoC).

CoC es un paradigma de programación que minimiza las decisiones que tienen que tomar los desarrolladores, simplificando tareas. No obstante, la flexibilidad no se pierde, ya que a pesar de otorgar valores por defecto, siempre se puede configurar de forma extendida. De esta forma se evita la repetición de tareas básicas a la hora de construir un proyecto.

Para comenzar a utilizar Spring Boot, hace falta indicarle a Maven y heredar del proyecto:

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.2.2.RELEASE</version>
</parent>
<dependencies>
   <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
</dependencies>
```

Con las etiquetas `<parent></parent>`indicamos que nuestro POM hereda del de Spring Boot. La dependencia spring-boot-starter-web es la necesaria para poder empezar con un proyecto de este tipo, pero a medida que crece la aplicación hacen falta más dependencias.

Si queremos alterar la configuración dada por defecto normalmente se ofrecen campos que podemos añadir al archivo application.properties en la carpeta src/main/resources del proyecto.

Para configuración avanzada, se añaden componentes que reemplazarán los componentes de auto-configuración. Estos componentes de auto-configuración son los que integra Spring Boot siempre que se cumpla alguna condición, que generalmente es que no exista ya un componente que haga las funciones que puede hacer este.

3. INYECCIÓN DE DEPENDENCIAS

3.1. Inversión de control

La inyección de dependencias es una característica fundamental de Spring, ya que todo el framework se basa en esta. Dependency injection (DI) es una de las formas que toma la idea de Inversion of Control (IoC). En una aplicación tradicional, los elementos que realizan tareas hacen llamadas a librerías reusables para hacer ciertos procesos genéricos, con IoC se consigue que este código reusable sea el que influya en el código que va a realizar el proceso.

Para entender cómo se utiliza DI en Spring hace falta entender un par de conceptos específicos de Spring: ApplicationContext y Bean.

3.2. Bean

Un *bean* es un objeto gestionado por el contenedor IoC, es decir, es el propio contenedor de IoC quien lo instancia y controla su ciclo de vida. Estos *beans* se pueden configurar mediante XML, donde se especifican los metadatos de configuración:

- Cómo se crea
- Detalles del ciclo de vida
- Dependencias

A continuación se muestra un ejemplo de un XML de configuración de *beans*:

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

  <bean id="..." class="...">
      <!-- configuración del bean -->
  </bean>
  <!-- Bean con método de inicialización -->
  <bean id="..." class="..." init-method="...">
      <!-- configuración del bean -->
  </bean>
  <!-- Bean con destructor -->
  <bean id="..." class="..." destroy-method="...">
      <!-- configuración del bean -->
  </bean>

</beans>
```

Estos *beans* se pueden integrar mediante la interfaz BeanFactory, un contenedor simple con compatibilidad con DI. En este caso se podría utilizar XmlBeanFactory para inicializar los *beans* obtenidos del archivo indicado. Los módulos de Spring integran *beans* ya definidos y listos para utilizar en la aplicación, dejando atrás la utilización de BeanFactory. Esta forma de configurar *beans* se está dejando de utilizar, reemplazado por configuraciones a través de código Java, como se explicará en el siguiente punto.

3.3. ApplicationContext

ApplicationContext es un contenedor avanzado de Spring. Puede cargar definiciones de beans, unir (*wire*) beans y administrarlos siempre que se solicite. Este contenedor está definido en la interfaz org.springframework.context.ApplicationContext. Aparte de tener todas las funcionalidades de BeanFactory añade otras como la capacidad de obtener mensajes de texto de un fichero .properties y la capacidad de publicar eventos a los elementos interesados.

En esta aplicación, se utiliza SpringApplication de Spring Boot, que genera un ApplicationContext de tipo ConfigurableApplicationContext. Para configurar beans, en vez de utilizar XML, también se puede utilizando anotaciones de Java. Dada la siguiente clase:

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
public class Application {
   public static void main(String[] args) {
       SpringApplication.run(Application.class, args);
   }
}
```

`@Configuration` indica que es un bean de tipo configuración, `@ComponentScan` realiza la búsqueda de beans para inyectarlos en el ApplicationContext, mientras que `@EnableAutoConfiguration` de Spring Boot inyecta automáticamente beans que considera adecuadas para que la aplicación pueda ejecutarse, mediante clases de auto-configuración. Estos beans irán siendo reemplazados por los que introduzca el desarrollador si lo desea. Las clases de auto-configuración son componentes de tipo `@Configuration`. Estas tres anotaciones suelen ir juntas en la clase principal de la aplicación, por lo tanto se puede reemplazar por `@SpringBootApplication`.

Los componentes pueden ser, entre otros, de los siguientes tipos:

- `@Component`
- `@Bean`
- `@Service`
- `@Repository`

- `@Controller`
- `@RestController`

Estos se registran como beans en el ApplicationContext gracias a `@ComponentScan`. Para inyectar estos beans a otras clases, se pueden utilizar las siguientes anotaciones:

- `@Resource`: para variables de instancia y setters
- `@Autowired`: para variables de instancia setters, constructores o cualquier otro tipo de método

Debido a su mayor flexibilidad, la combinación de `@ComponentScan` con `@Autowired` resulta ser de las más cómodas. A continuación se muestra un ejemplo de un componente de tipo `@Controller` que inyecta la clase `ExampleComponent` en su constructor:

```
package com.welbits.spring.example;

import org.springframework.stereotype.Component;

@Component
public class ExampleComponent {

	public void method1() {
		...
	}

	public boolean method2() {
		...
	}

}
```

`ExampleComponent` se es introducido al contenedor de ApplicationContext debido a `@ComponentScan`, listo para ser inyectado mediante anotaciones. Como por ejemplo:

```
package com.welbits.spring.example;

import org.springframework.beans.factory.annotation.Autowired;
import com.welbits.spring.example.ExampleComponent;

@Controller
public class ExampleController {

   private ExampleComponent exampleConponent;

   @Autowired
   public RootController(ExampleComponent exampleConponent) {
      this.exampleConponent = exampleConponent;
   }

   ...
}
```

A los componentes se les puede añadir la anotación `@Qualifier` con un nombre como parámetro para luego indicarlo en el método en el que se inyecta. Esto se utiliza cuando varios componentes implementan el mismo tipo, pudiendo especificar cuál es el componente que quiere utilizarse. En el ejemplo anterior habría que añadir la anotación `@Qualifier` en una clase `UserRepository` y además en el método con `@Autowired`:

```
@Autowired
public UserServiceImpl(@Qualifier("userRepository1") UserRepository userRepository)
{
       this.userRepository = userRepository;
}
```

Se puede utilizar una clase de tipo @Configuration para añadir beans a modo de métodos que devuelven elementos de un cierto tipo de clase. Esto es útil cuando queremos inyectar objetos de una clase que no podemos modificar, como pueden ser clases de una librería externa. A continuación se muestra una clase de configuración de ejemplo:

```
package com.tutorialspoint;
import org.springframework.context.annotation.*;

@Configuration
public class ExternalClassConfig {

   @Bean
   public ExternalClass externalClass(){
      return new ExternalClass();
   }
}
```

La llamada al método `externalClass()` no genera un objeto cada vez, si no que devuelve el objeto generado y contenido en el contenedor Spring gracias a la anotación `@Bean`. El nombre utilizado para inyectar estos beans es por defecto el nombre de la clase, pero pueden indicarse nombres en la anotación: `@Bean(name="name1", "name2")`. Además, si solamente hay un bean de una determinada clase, esta será inyectada independientemente de los nombres dados en la configuración del bean y en su inyección.

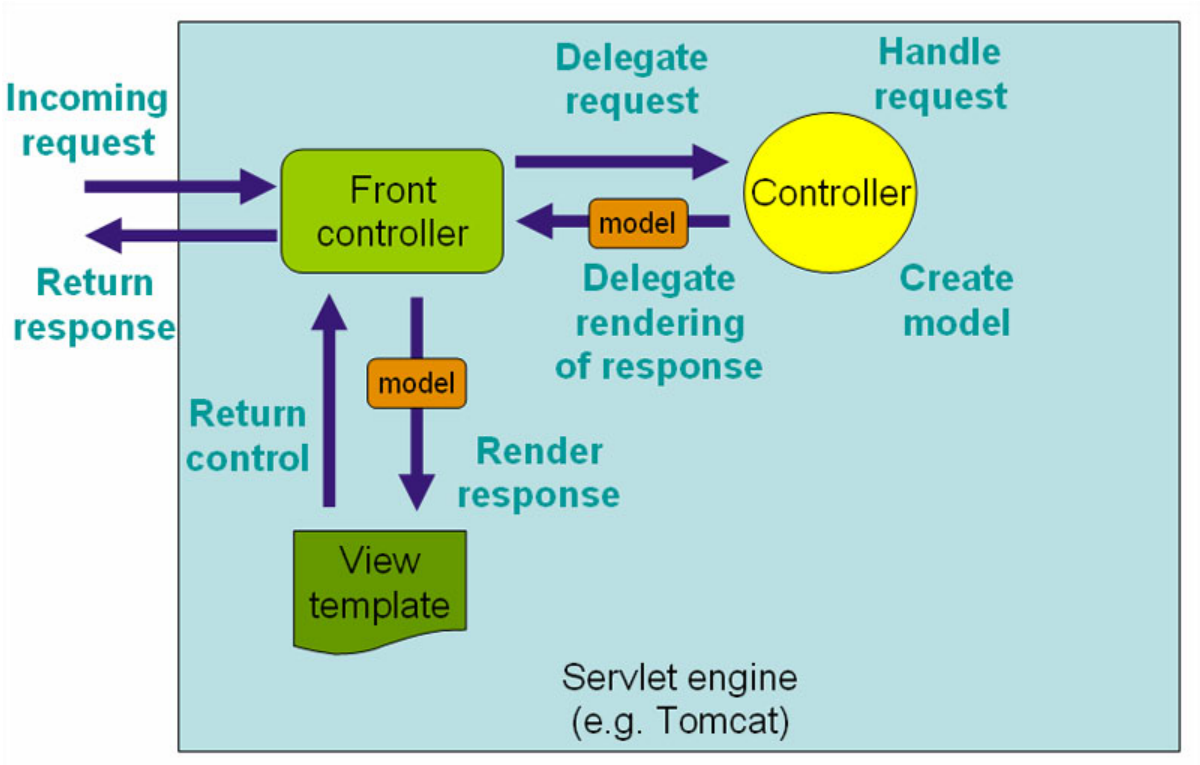
4. SPRING WEB MVC

4.1. Funcionamiento de MVC con Spring

El framework Web model-view-controller (MVC) está diseñado en base a un DispatcherServlet, que se encarga de distribuir (como su nombre indica) las peticiones a distintos controladores. Permite configurar el mapeo de los controladores y resolución de vistas, localización, zona horaria y temas. El controlador por defecto es `@Controller`, aunque hay otros que lo usan como base y añaden especificación. `@RequestMapping` son las anotaciones básicas utilizadas para configurar el mapeo de estos controladores.

Los controladores son los responsables de preparar un modelo y elegir el nombre de la vista, pero pueden responder directamente y completar la petición si se desea. El modelo es una interfaz que permite abstraerse del tipo de tecnología que se está utilizando para la vista, transformándose al formato adecuado cuando se solicita. Las vistas pueden generarse utilizando tecnologías como JSP, Velocity, Freemaker o directamente a través de estructuras como XML o JSON.

La comunicación básica de estos elementos es la que se muestra en la figura:

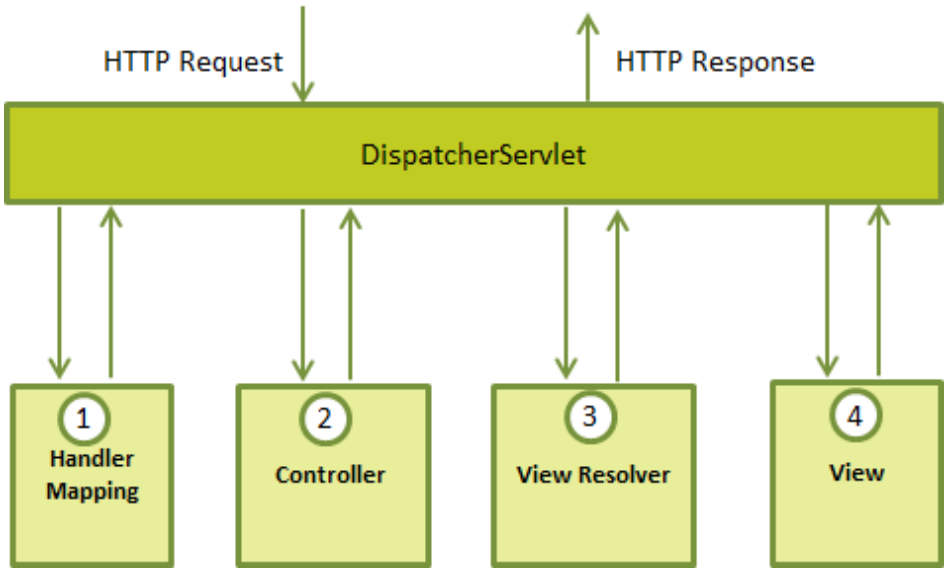


(Imagen de: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/images/mvc.png)

Para entender mejor cómo se decide qué controlador va a realizar las acciones y qué vistas se van a utilizar, es necesario entender el concepto de DispatcherServlet. El DispatcherServlet es el componente que recibe las peticiones HTTP y devuelve las respuestas, es "Front controller" de la previa imagen.

Tras una petición HTTP, el DispatcherServlet realiza una serie de acciones:

1. Consulta el mapeo de direcciones para llamar al controlador adecuado.
2. El controlador recibe la petición mediante el método adecuado. Este método preparará los datos basados en la lógica de negocio definida e indicará el nombre de la vista al DispatcherServlet.
3. Con ayuda del objeto ViewResolver, el DispatcherServlet escogerá la vista adecuada para la petición.
4. El DispatcherServlet pasa los datos propios de la lógica de negocio a la vista para que se muestre adecuadamente.



(Imagen de: http://www.tutorialspoint.com/spring/images/spring_dispatcherservlet.png)

Los siguientes puntos explican cómo se comporta cada uno de los componentes del MVC, una vez que se entiende en qué contexto se encuentran hace falta entender cómo se comporta cada uno de ellos internamente.

4.2. Controladores

Los elementos anotados con `@Controller` son una especialización de `@Component`, por lo tanto serán detectados e integrados en el contenedor de Spring automáticamente. Estos se usan en combinación con los métodos anotados con `@RestMapping` o especializaciones de esta anotación.

```
package com.welbits.spring.example;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class ExampleController {

    @RequestMapping("/")
    @ResponseBody
    public String home() {
        return "This is a body, not a file";
    }
}
```

En `@RequestMapping("/")` se pasa por parámetro el valor, que es la URI que debe mapearse con ese método (la petición a la que va a responder). Se permiten patrones de estilo Ant para este parámetro. `@RequestMapping` se puede aplicar a la clase, todos los métodos de esta clase heredan el valor del mapeo, añadiendo la especificación de cada método particular. Otros parámetros de `@RequestMapping` son:

- name: el nombre del mapeo
- method: el tipo de peticiones HTTP a los que responde (GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE)
- params: los parámetros de la petición
- headers: cabeceras a las que responde el método
- consumes: el tipo de soportes que consume el método
- produces: el tipo de soportes que produce el método

También se puede añadir anotaciones extra a `@RequestMapping`, en este ejemplo se utiliza `@ResponseBody` para especificar que es el método quien escriba directamente en el cuerpo de la respuesta, evitando utilizar una vista. Otra forma de conseguir este resultado es utilizando la anotación `@RestController`, que aplica `@ResponseBody` de forma que todos los métodos internos lo heredan. Este tipo de respuestas es útil para servicios REST o respuestas a peticiones tipo AJAX. Por defecto, si no se utiliza `@ResponseBody`, la vista debe ser resuelta por un `ViewResolver`, dependiendo del nombre de la vista devuelto.

Cuando a las peticiones generadas por el usuario tienen como resultado una vista en base a una plantilla y no hace falta intervenir previamente, se pueden mapear llamadas en un archivo de configuración de una forma mucho más compacta.

```
package com.welbits.spring.config;

import org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
        registry.addViewController("/login").setViewName("login");
    }
}
```

La clase de configuración hereda de `WebMvcConfigurerAdapter` que es una implementación de `WebMvcConfigurer` con todos los métodos vacíos para que sólo tengan que sobrecargarse los necesarios y no haga falta implementar métodos vacíos en la clase final. `WebMvcConfigurer` define métodos que permiten cambiar la configuración MVC de Spring. El método `addViewControllers` permite mapear controladores a las URLs dadas para que renderizen la vista dependiendo del nombre dado. El siguiente punto explica cómo se resuelven las vistas en base a estos nombres dados.

4.3. Vistas

4.3.1. View Resolvers

Los View Resolvers son objetos que implementan la interfaz ViewResolver. Estos objetos tienen que ser capaces de decidir qué vista debe enviarse dependiendo de un nombre. Pueden ofrecer características como la resolución de distintas vistas dependiendo de localización u otro tipo de distinciones (internacionalización). La interfaz tan sólo tiene el método: `View resolveViewName(String viewName, Locale locale) throws Exception;`

Utilizan objetos de la clase View, la cual ajusta los datos de una petición al tipo de tecnología de visualización se está utilizando para interactuar con el usuario.

En caso de utilizar Spring Boot, la anotación `@EnableAutoConfiguration` añade la clase de auto-configuración `WebMvcAutoConfiguration` que a su vez añade al contexto varios tipos de ViewResolver, dependiendo del resto de módulos integrados en el proyecto.

Si se desea añadir una petición que devuelva una vista resuelta por un ViewResolver, se obtendría como resultado:

```
@Controller
public class ExampleController {

   @RequestMapping("/")
   public String home() {
       return "/WEB-INF/jsp/home.jsp";
   }
}
```

En el directorio src/main/webapp/WEB-INF se encuentran archivos que no forman parte del directorio público de la aplicación, por lo tanto se considera buena práctica almacenar las plantillas que utilizarán los ViewResolver en este directorio. El directorio webapp se consideraría la raíz del proyecto, por lo tanto, suponiendo que el archivo se encuentra en la carpeta jsp/ dentro de WEB-INF/ sólo hay que devolver "/WEB-INF/jsp/home.jsp".

Se considera buena práctica que todas las plantillas se almacenen en el mismo directorio y sean del mismo tipo, para simplificar los controladores se puede especificar al ViewResolver el prefijo y el sufijo del nombre del archivo ofrecido. Con Spring Boot, se deben indicar estos dos valores a los ViewResolver inyectados automáticamente y se puede utilizar un archivo de configuración .properties para especificarlos:

```
spring.view.prefix: /WEB-INF/jsp/
spring.view.suffix: .jsp
```

Este es el motivo por el cual a un ViewResolver normalmente se le da como nombre de la vista tan sólo el nombre del fichero, sin especificar la ruta ni la extensión. Es decir, una vez configurados el prefijo y el sufijo que utilizarán los ViewResolver, el controlador de ejemplo sería de esta forma:

```
@Controller
public class ExampleController {

    @RequestMapping("/")
    public String home() {
        return "home";
    }
}
```

4.3.2. JSP y JSTL

JSP

JavaServer Pages (JSP) es una de muchas tecnologías que se pueden utilizar para renderizar una vista a partir de modelos de Spring. JSP permite generar páginas web dinámicas basadas en documentos estándar de páginas web como HTML o XML. Al ser basado en Java, tiene gran compatibilidad con Spring. Al ser código Java, se trata de código compilado, por lo tanto ofrece ventajas de estabilidad y rapidez frente a alternativas de código interpretado. Se ejecuta en el lado del servidor, por lo tanto ofrece facilidad y seguridad a la hora de acceder a bases de datos o hacer otros procesos complejos como procesar imágenes.

Para ejecutar las páginas JSP, se necesita un servidor Web con un contenedor Web que cumpla con las especificaciones de JSP y de Servlet. Tomcat cumple con dichas especificaciones desde la versión 5. Las páginas JSP se ejecutan dentro de contenedores *servlet*.

Las páginas JSP son archivos con la extensión '.jsp' y que combinan texto plano con código Java insertado entre la apertura '<%' y el cierre '%>'. La salida es generalmente un archivo interpretable por navegadores, combinando el texto plano con el resultado del código Java, que ofrece la parte dinámica de la web.

INTEGRACIÓN

Para utilizar JSP con JSTL es necesario añadir estas dependencias en caso de utilizar Maven:

```
<dependency>
	<groupId>org.apache.tomcat.embed</groupId>
	<artifactId>tomcat-embed-jasper</artifactId>
	<scope>provided</scope>
</dependency>
<dependency>
	<groupId>javax.servlet</groupId>
	<artifactId>jstl</artifactId>
	<scope>provided</scope>
</dependency>
```

Añadiendo así Jasper, capaz de ejecutar JSP, y JSTL para la utilización de la misma.

SINTAXIS

- Variables implícitas: Variables que se pueden utilizar sin necesidad de declarar o configurar. Estas son:
  - pageContext
  - request
  - response

- session
- config
- application
- out
- page
- exception

❖ Directivas: Generan información que puede ser utilizada por el motor de JSP, sirven como configuración previa a la ejecución de la página.

Se utiliza la siguiente sintaxis:

```
<%@ directive attribute="value" %>
```

Las directivas definidas son:

- **include**: incluye contenidos de otros ficheros.

```
<%@ include file="example.html" %>
```

- **taglib:** importa Tag Libraries.

```
<%@ taglib uri="/tags/example-html" prefix="html" %>
```

- **page:** especifica atributos propios de la página:
  - **import:** importa clases y paquetes java.
  - **sesión:** indica si utiliza datos de sesión del cliente.
  - **contentType:** indica el tipo MIME de la respuesta.
  - **buffer:** buffer del objeto writer 'out'.
  - **errorPage:** dirección de la página a la que se tiene que redirigir la actual en caso de error.
  - **isErrorPage:** indica si es una página de error para manejar excepciones. Da acceso a la variable implícita 'exception' en caso de tener el valor "true".

❖ Declaraciones: se declaran funciones y variables añadiendo '!' a la apertura '<%'.

```
<%! int numVisits = 30; %>
```

❖ Scriplets: partes de código java.

```
<% … Java code … %>
```

❖ Expresiones: expresiones que se ejecutan dentro del servlet JSP.

```
<%= numVisits + 1 %>
```

❖ Etiquetas: se especifican prefijos para etiquetas que ayudan a simplificar el código.

```
<%@ taglib uri="/taglib/example" prefix="exmpl" %>
...

<exmpl:hello/>
...
```

JSTL

JSP Standard Tag Library (JSTL) es una agrupación de etiquetas JSP estándares y comúnmente utilizadas en aplicaciones web. JSTL abarca tareas estructurales como iteraciones y condicionales y tareas de manipulación de información como documentos XML, i18n y extracción de datos a través de SQL. La colección de etiquetas se puede dividir en una serie de grupos, cada uno de los grupos compone una librería. Para utilizar cualquiera de estas librerías, se debe añadir mediante una directiva 'taglib'. Estos son los grupos:

- Etiquetas básicas: Las etiquetas usadas más frecuentemente se encuentran en este grupo. Uri: "http://java.sun.com/jsp/jstl/core"
- Etiquetas de formato: Son aquellas que se utilizan para dar formato a texto, fechas, horas o numeraciones. Uri: "http://java.sun.com/jsp/jstl/fmt"
- Etiquetas SQL: Se utilizan para la comunicación con bases de datos relacionales. Uri: "http://java.sun.com/jsp/jstl/sql"
- Etiquetas XML: Son utilizadas para crear y manipular información en formato XML. Uri: "http://java.sun.com/jsp/jstl/xml"
- Funciones JSTL: Esta librería contiene una serie de funciones estándar, principalmente basadas en la manipulación de cadenas de caracteres. Uri: "http://java.sun.com/jsp/jstl/functions"

4.4. ORM en Spring

4.4.1. ORM

Object-relational mapping (ORM) es una técnica utilizada en lenguajes orientados a objetos que permite el acceso a datos de una base de datos relacional, empleando un motor de persistencia para convertir datos entre el sistema de tipos del lenguaje de programación y el de la BBDD relacional. Con ORM se obtienen las entidades de una base de datos relacional en forma de objetos, permitiendo utilizar las características propias del paradigma de orientación a objetos.

El ORM permite abstraerse de la base de datos que agiliza el desarrollo reduciendo tiempos de implementación. El código obtenido es reutilizable y por lo tanto más mantenible, ya que ORM es una capa que encapsula la lógica entre la base de datos y el resto de la aplicación.

Además, los ORM implementados ofrecen sistemas de seguridad para evitar ataques como inyecciones SQL.

Con ORM se generan y obtienen los modelos de una aplicación, completando así las tres partes de la arquitectura MVC. Este punto se centra en la aplicación de ORM con el Framework de Spring.

4.4.2. Spring Data

Spring Data es un módulo de Spring que a su vez contiene una colección de submódulos asociados a tecnologías de acceso a datos. Sus módulos dan acceso a tecnologías nuevas como bases de datos no relacionales, servicios cloud y frameworks map-reduce, pero algunos submódulos también ofrecen un soporte robusto sobre las tradicionales bases de datos relacionales.

En concreto, Spring Data JPA es un submódulo que permite implementar repositorios JPA de forma simple. Java Persistence API (JPA) es una API de persistencia desarrollada para Java que maneja datos relacionales siguiendo el patrón ORM. Spring Data JPA permite implementar la capa ORM de una forma simple y reduce las líneas de código necesarias implementando automáticamente la capa de acceso a los datos en base a interfaces de repositorio escritas por el desarrollador.

Estas interfaces de repositorio son clases parametrizadas que capturan el tipo del objeto que se va a gestionar junto al tipo del id del objeto. Las implementaciones de dichos repositorios pueden incluir métodos de tipo CRUD (Create, Read, Update, Delete), como lo hace la interfaz CrudRepository<T, ID>. La interfaz de la que deben extender los repositorios de Spring Data JPA es JpaRepository<T, ID>. El siguiente apartado profundiza sobre cómo se utiliza esta tecnología y cómo funciona internamente.

4.4.3. Utilización

Para empezar utilizar Spring Data JPA es necesario añadir su módulo y un módulo para una base de datos relacional, que en este caso será MySQL:

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
</dependency>
```

Para definir una entidad se debe crear una clase Java con la estructura de dicha entidad, suponiendo que la entidad es un usuario:

```
package com.welbits.spring.entities;

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

@Entity
@Table(name="usr")
public class User {

	public static final int EMAIL_MAX = 100;
	public static final int NAME_MAX = 50;

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private long id;

	@Column(nullable = false, length = EMAIL_MAX)
	private String email;

	@Column(nullable = false, length = NAME_MAX)
	private String name;

	@Column(nullable = false)
	private String password;

	protected User(){}

	public User(String email, String name, String password){
		this.email = email;
		this.name = name;
		this.password = password;
	}

	// Getters, setters
	...
}
```

Se puede observar que el usuario contiene los campos: "id", "email", "name" y "password". Cada uno de estos campos tiene su getter y setter.

Todas las entidades se definen con la anotación `@Entity` para indicar que serán utilizadas por Spring Data. Por defecto, el nombre de la tabla en la que serán almacenadas las entidades de un tipo es el mismo nombre que el de la clase, en este caso se ha utilizado la anotación `@Table` para indicar un nombre específico.

El ID del objeto también debe ser indicado a JPA tras anotaciones para que lo utilice como clave primaria en la base de datos. Se utiliza `@Id` para especificar qué campo es y `@GeneratedValue` si se desea que JPA auto-genere estos IDs. Aunque por defecto es AUTO, las estrategias de auto-generado posibles son: TABLE, SEQUENCE, IDENTITY y AUTO.

Las columnas son por defecto el resto de atributos y también utilizan por defecto el nombre del campo a no ser que se especifique lo contrario. Estas columnas pueden ser indicadas con la anotación `@Column` y su nombre puede ser modificado a través del argumento "name". Otros posibles parámetros utilizados en la base de datos son:

- unique: Especifica si es una clave única. Por defecto 'false'.
- nullable: Especifica si se puede insertar un registro sin este campo. Por defecto 'true'.
- insertable: Especifica si se incluye en operaciones INSERT. Por defecto 'true'.
- updatable: Especifica si se incluye en operaciones UPDATE. Por defecto 'true'.
- columnDefinition: Especifica el tipo. Por defecto inferido del campo.
- table: Especifica la tabla en la que se encuentra. Por defecto en la primaria.
- length: Especifica la longitud de la columna. Por defecto 255.
- precision: Especifica la precisión decimal. Por defecto 0.
- scale: La escala de un numérico. Por defecto 0.

Utilizando estos parámetros se define la estructura de la base de datos relacional sin necesidad de crearla y asegurar de que hay trazabilidad entre los datos. La capa ORM generada por Spring JPA se encarga de estas operaciones, que son completamente esquemáticas.

También se observa un constructor con argumentos, utilizado como una de las formas de generar entidades de este tipo. El otro constructor está vacío y con visibilidad *protected*, éste es para el uso de JPA. Se pueden añadir otros métodos si se desea, es buena práctica implementar el método toString para *testing* y *logging*.

El siguiente paso es codificar un repositorio que enlace a esta entidad. Como se ha explicado previamente, el repositorio será tan sólo una interfaz específica para cada entidad que JPA es capaz de implementar en tiempo de ejecución de forma automática.

El siguiente repositorio funciona con entidades de tipo User:

```
package com.welbits.spring.repositories;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import com.welbits.spring.entities.User;
public interface UserRepository extends JpaRepository<User, Long> {
      User findByEmail(String email);
      List<User> findByName(String name);
}
```

`UserRepository` extiende de la interfaz `JpaRepository`, una extensión de `Repository` específica para JPA. Los parámetros son `User` y `Long`, que indican el tipo de la identidad y del ID respectivamente. Debido a la herencia, `UserRepository` ya tiene los métodos básicos tipo CRUD para encontrar, guardar y borrar entidades de tipo `User`.

Para definir peticiones más específicas a la entidad, se pueden declarar métodos en la interfaz. Dado que JPA es quien implementa la interfaz, es importante que se siga una estructura de nombres que luego utilizará JPA para implementar las peticiones necesarias.

En el caso de `User findByEmail(String email);` la petición creada es:

```
select u from usr u where u.email = ?1
```

En el caso `de List<User> findByName(String name);` la petición creada es:

```
select u from usr u where  u.name = ?1
```

Todos los métodos comienzan por `findBy` y una propiedad del objeto buscado, seguido de más opciones que se especifican mediante palabras clave. Las palabras clave son: `And`, `Or`, `Between`, `LessThan`, `GreaterThan`, `After`, `Before`, `IsNull`, `(Is)NotNull`, `Like`, `NotLike`, `StartingWith`, `EndingWith`, `Containing`, `OrderBy`, `Not`, `In`, `NotIn`, `True`, `False`, `IgnoreCase`.

Ejemplos de uso:

`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`
`findByAgeOrderByLastnameDesc`	`… where x.age = ?1 order by x.lastname desc`
`findByActiveTrue`	`… where x.active = true`

Esta forma de crear peticiones suele ser la más cómoda tanto de codificar como de visualizar. No obstante, hay alternativas para casos en los que el nombre del método no sea buena opción:

- Mediante XML: Editando el fichero de configuración orm.xml en META-INF. Se le asocia un nombre que puede ser resuelto en tiempo de ejecución a un *tag* <named-query>.

```
<named-query name="User.findByName">
 <query>select u from usr u where u.name = ?1</query>
</named-query>
```

- Mediante anotaciones en la clase: Evita ficheros de configuración permitiendo añadir anotaciones en la entidad que se resuelven en tiempo de compilación.

```
@Entity
@NamedQuery(name = "User.findByEmail",
            query = "select u from usr u where u.email = ?1")
```

```
public class User {
  ...
}
```

- Mediante anotaciones en el método: Tienen preferencia sobre las anteriores. Se definen en los métodos del repositorio. Se pueden utilizar peticiones nativas con el parámetro "nativeQuery".

```
public interface UserRepository extends JpaRepository<User, Long> {
      @Query("select u from User u where u.email = ?1")
      User findByEmail(String email);
}
```

Una vez definida una entidad y su respectivo repositorio, es necesario configurar la conexión con una base de datos. Con la utilización de MySQL, tras generar una base de datos llamada "spring_demo", se configuran los siguientes campos en application.properties:

```
spring.datasource.url: jdbc: mysql://localhost:3306/spring_demo
spring.datasource.username: root
spring.datasource.password:
spring.datasource.driverClassName: com.mysql.jdbc.Driver

spring.jpa.hibernate.ddl-auto: update
spring.jpa.database: MYSQL
```

Las propiedades de `spring.datasource` indican al módulo genérico de Spring Data sobre cuál es la localización de la base de datos a utilizar, junto a sus credenciales.

Las propiedades de `spring.jpa` indican a este módulo específico cómo tratar con la base de datos conectada.

4.4.4. Herencia

En el caso de herencia, se puede utilizar una serie de anotaciones para indicar cómo se quiere tratar. Se utilizará como ejemplo una serie de clases: Step y sus clases hijas SimpleStep y MultipleStep.

Cuando una serie de objetos comparten herencia, se puede apreciar que, normalmente, no contienen los mismos datos. Es por esto que no se pueden almacenar todas las clases en una misma tabla. Sabiendo esto, hay tres distintas opciones que se pueden utilizar:

- Utilizar una tabla única para todas las clases, dejando vacíos los campos que no son propios de la clase de la que se trata. Esto puede producir una gran cantidad de espacio reservado innecesario en muchos casos, pero es viable cuando los atributos de las clases prácticamente no varían. Debe utilizar un discriminador para distinguir la clase de la que

se trata en cada registro. El discriminador almacena el nombre de la clase para poder generar un objeto de la clase adecuada cuando se requiera.

- Utilizar una tabla para cada clase. En este caso, el espacio reservado siempre será adecuado y se puede distinguir la clase sin necesidad de un discriminador. Por otra parte, puede complicar peticiones como búsquedas conjuntas.

- Utilizar una tabla única para datos comunes y una tabla específica de cada clase para el resto. Es el caso en el que los atributos de la clase padre se comparten y los datos específicos se almacenan en distintas tablas. También requiere un discriminador.

Con estos tres casos se cubren todas las posibilidades, la decisión depende de la naturaleza de los datos de cada clase. Una vez tomada la decisión, la implementación es simple ya que sólo debe añadirse la anotación `@Inheritance` indicando la estrategia (por defecto SINGLE_TABLE) y en caso de que sea necesario indicar un nombre para la columna del discriminador mediante la anotación `@DiscriminatorColumn`.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name="disc")
@Table(name="step")
public abstract class Step {
    …
}
```

Para las clases hijas no es necesario realizar ningún cambio, basta con que estén anotadas como entidades también y que hereden de la clase padre como es debido.

5. INTERNACIONALIZACIÓN

Los conceptos de internacionalización (i18n) y localización (L10n) están presentes en cualquier proyecto cuyo objetivo es alcanzar a la mayor cantidad de usuarios posible. i18n se encarga de ofrecer un método para adaptar una aplicación a un determinado idioma y región sin necesidad de operar sobre el código de dicha aplicación. L10n es el proceso que adapta la interfaz para que muestre el lenguaje y los formatos regionales adecuados.

El framework de Spring ofrece la opción de utilizar esto y Spring Boot ofrece los componentes mínimos necesarios para comenzar a utilizarlo. Uno de estos componentes es ResourceBundleMessageSource, ofrecido por el componente de auto-configuración MessageSourceAutoConfiguration, que permite utilizar un sistema de ficheros para asociar textos dependiendo de la localización del cliente. A continuación se muestran los campos que se pueden configurar en application.properties, con sus valores por defecto:

```
spring.messages.basename=messages
spring.messages.cache-seconds=-1
spring.messages.encoding=UTF-8
```

"basename" es la parte inicial de los ficheros .properties que se utilizarán para i18n, "cache-seconds" son los segundos de cache definidos (-1 es una cache permanente) y "encoding" es la codificación de los textos utilizados en los ficheros. Para crear los distintos ficheros basados en localización, se han de crear los archivos con el formato {basename}_{locale}.properties, siendo {locale} es el código del lenguaje utilizado. Por lo tanto, el fichero para los usuarios que están en zonas de habla española es "messages_es.properties", mientras que el fichero para zonas en las que se habla inglés es "messages_en.properties". El fichero "messages.properties" sirve como fichero por defecto en caso de no obtener el mensaje de una clave en el fichero especifico de la localidad deseada.

Ejemplo de messages_en.properties:

```
helloWorld = Hello, World!
message = Generic message
paramMessage = Hello, {0}
```

Ejemplo de messages_es.properties:

```
helloWorld = ¡Hola, Mundo!
message = Mensaje genérico
paramMessage = Hola, {0}
```

Como se puede ver, los ficheros, al igual que el resto de ficheros .properties, se componen de una lista de pares "clave = valor" o "clave : valor". El valor en este caso es el mensaje que se quiere mostrar. En este caso, utilizar la clave message no devolverá el mismo valor en España que en Inglaterra, ya que el código de localización no es el mismo. Los mensajes pueden tener parámetros para valores que dependen de ciertos valores que cambian durante la ejecución de la aplicación. Estos parámetros se indican con "{n}" donde n indica la posición en la que se encuentra el parámetro a la hora de solicitar el mensaje. Las claves de estos textos se pueden utilizar tanto en el código Java como en las plantillas de la vista de la aplicación.

En caso de utilizar las claves dentro del código Java, se utiliza la clase `MessageSource`:

```
private static MessageSource messageSource;

@Autowired
public Utils(MessageSource messageSource) {
     Utils.messageSource = messageSource;
}

private static String getMessage(String messageKey, Object... args) {
     return messageSource.getMessage(messageKey, args, Locale.getDefault());
}
```

El método `getMessage` del objeto `MessageSource` tiene como parámetros la clave que identifica al mensaje, los argumentos que puedan ser utilizados por el mensaje y la localidad para la cual se quiere resolver el mensaje. El ejemplo muestra un método que facilita el mensaje para la localidad registrada para el usuario en cuestión.

En caso de utilizarlo en las plantillas de visualización, depende de la tecnología utilizada. Este es un ejemplo utilizando JSP, donde se añade la librería 'tags' de Spring para poder acceder a los mensajes:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<p><spring:message code="helloWorld" /></p>
```

A veces se necesita saber dentro de la aplicación la localidad del usuario, para eso hay que configurar un componente de tipo `LocaleResolver`:

```
@Bean
public LocaleResolver localeResolver() {
   SessionLocaleResolver slr = new SessionLocaleResolver();
   slr.setDefaultLocale(Locale.US);
   return slr;
}
```

En este ejemplo se utiliza `SessionLocaleResolver`, que obtiene la localidad configurada en la sesión del usuario en caso de que la haya cambiado, al contrario de la implementación principal de LocaleResolver, AcceptHeaderResolver. En caso de que no se haya configurado ninguna localidad en la sesión, se utilizará la localidad de la cabecera accept-request o si no la introducida como 'default locale'.

Otro componente importante es un LocaleChangeInterceptor, que funciona como intermediario en caso de que se quiera cambiar la localidad manualmente. Esto puede ocurrir en escenarios de testing o en caso de ofrecer una interfaz dentro de la aplicación que ofrezca un cambio de idioma. Para configurar un componente de este tipo se puede utilizar el siguiente código:

```
@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
    lci.setParamName("lang");
    return lci;
}
```

Este interceptor busca el parámetro 'lang' para modificar la localidad utilizada en la ejecución. Por ejemplo, 'www.example.com/?lang=en' obligaría a la aplicación a utilizar el inglés independientemente de la localidad del usuario.

Para que este interceptor sea utilizado por Spring Boot, hace falta añadirlo en su registro. Se puede utilizar la clase de configuración que extiende de WebMvcConfigurerAdapter para sobrecargar el método addInterceptors e introducir el interceptor que se ha creado. Un ejemplo de lo utilizado en este punto aplicado a la clase de configuración es:

```
@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter {
    …
    @Bean
    public LocaleResolver localeResolver() {
        SessionLocaleResolver slr = new SessionLocaleResolver();
        slr.setDefaultLocale(Locale.US);
        return slr;
    }
    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
        lci.setParamName("lang");
        return lci;
    }
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }
}
```

6. FORMULARIOS

6.1. Introducción

Los formularios forman una parte importante de una página web que requiera interactuar con el usuario. Las entradas de texto, numéricas, opciones en listas, casillas e incluso selección de archivos componen estos formularios, pudiendo crear desde pantallas de inicio de sesión y registro a pantallas mucho más complejas.

Mientras que la manera de presentar estos formularios forma parte de la vista y es una tarea bastante simple, el código necesario para utilizar los datos que el usuario introduce puede ser costoso. Cuando se crea una vista que recibe datos del usuario, luego hay que transformar esta información a objetos Java, posiblemente ajustándose a algún modelo o modelos que luego se almacene en nuestra base de datos.

Además, es necesario controlar si el usuario ha cometido un error en alguna de sus entradas. Para evitar esto, se debe limitar los valores posibles que se pueden introducir al mínimo posible y, aun así, hay que comprobar finalmente si todos los datos tienen coherencia respecto al resto. Para ofrecer al usuario una aplicación de mayor calidad en cuanto a usabilidad, también hay que implementar el envío de mensajes de error indicando los fallos que debe solventar y en qué zonas, y luego un mensaje de aviso cuando todo está correcto.

Con Spring es posible simplificar estas tareas, mediante enlaces directos entre las vistas y objetos java, integración con validadores de modelos, la clase SpringValidatorAdapter para validaciones más específicas y los atributos de redirección para indicar el estado del proceso. En este apartado se explican estas características.

6.2. Data Binding

Cuando se accede a una URL, el *dispatcher* asignará al controlador responsable de esta petición. El controlador, antes de resolver una vista, puede modificar un objeto Model de forma que ofrece información a esta vista.

Model es un objeto del paquete UI de Spring Framework. Model permite añadir nuevos atributos que mediante librerías de Spring en la vista pueden mostrarse:

```
@RequestMapping(value = "/signup", method = RequestMethod.GET)
public String signup(Model model) {
      model.addAttribute(new SignupForm()); // atributo por defecto:
                                              signupForm
    return "signup"; }
```

En este ejemplo se ve el objeto model, que es inyectado automáticamente por Spring, en uso mediante el método addAttribute. Este método permite introducir cualquier objeto o colección de objetos que se convertirán en objetos accesibles desde la vista. El método tiene la posibilidad de introducir un String que indique el nombre del atributo con el que queramos referirnos en la vista. Como ocurre con la mayoría de nombres opcionales, si no se introduce se utilizará el nombre de la clase en *lowerCamelCase*. En este caso, el objeto se llamará "signupForm".

Para simplificar la explicación, se supone de momento que el objeto SignupForm es el siguiente:

```
package com.welbits.spring.dto;

public class SignupForm {

	private String email;
	private String name;
	private String password;

	public String getEmail() {
		return email;
	}

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

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getPassword() {
		return password;
	}

	public void setPassword(String password) {
		this.password = password;
	}

	@Override
	public String toString() {
		return "SignupForm [email=" + email + ",
				name=" + name + ",
				password=" + password + "]";
	}

}
```

Es importante que todos los atributos del modelo a los que se quiere acceder tengan sus respectivos *getters* y *setters* accesibles públicamente. Si esto se cumple, acceder al atributo de un objeto enviado a la vista es simple:

```
${signupForm.name }
```

En este caso se estaría accediendo al atributo name del formulario, que al estar recién creado es una cadena vacía. Su funcionalidad cobra sentido cuando se quiere generar una página dinámica que utiliza valores de la base de datos. Sin embargo cuando se quiere generar un formulario de edición de un objeto, no es posible asociar un nuevo valor al atributo del modelo enviado por el controlador de forma trivial.

Para eso, Spring ofrece la librería forms. En el caso de utilizar JSP sería:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
```

Esta taglib reemplaza el uso de los formularios de HTML. La librería permite generar los mismos elementos que los de HTML pero con atributos adicionales que ayudan a enlazar los campos del formulario con modelos de la vista. Su uso es intuitivo si se conoce el uso común de formularios en web, y reduce el tiempo necesario para enlazar objetos de forma notable.

Cada tag renderiza una etiqueta del mismo nombre en HTML. Estos son los tags que ofrece la librería:

- **form**: Su atributo modelAttribute sirve como indicador al objeto con el que todos los elementos del formulario deben trabajar.
- **input**: Se indica el valor del atributo del modelo que se está editando mediante el atributo "path".
- **checkbox**: Tiene distintos usos. Puede cambiar el valor de un objeto booleano dependiendo de su estado, o añadir en una lista el valor seleccionado.
- **checkboxes**: Genera multiples eitquetas de tipo *checkbox* en HTML. Añadirá en una lista los valores seleccionados.
- **radiobutton**: Se utiliza generalmente para aplicar distintos valores al mismo atributo.
- **radiobuttons:** Genera automáticamente un elemento para cada valor de una lista.
- **password:** Genera un elemento <input> con tipo "password". Principalmente porque antes de Spring 3 no era posible generar elementos <input> de tipos distinto de texto.

Por defecto no muestra el password pero se puede cambiar el comportamiento con el atributo "showPassword"

- **select:** Renderiza el elemento <select> y sus etiquetas internas <option> y <options>. Al enlazarse con el modelo, indica el elemento de la lista que el modelo tiene en ese momento como seleccionado.
- **option:** Ofrece la posibilidad de generar la lista de opciones en la vista para que luego se asocien al atributo enlazado a la etiqueta "select".
- **options:** Genera etiquetas <option> a partir de una lista
- **textarea:** Asocia un atributo de texto a una etiqueta HTML <textarea>
- **hidden:** Genera una etiqueta <input> de tipo "hidden"
- **errors:** Renderiza etiquetas para indicar errores.

Un ejemplo con el objeto del registro:

```
<form:form modelAttribute="signupForm" role="form"
      autocomplete="off">
      <div class="form-group">
            <spring:message code="emailAddressPlaceholder"
                  var="emailAddressPlaceholder" />
            <form:label path="email">
                  <spring:message code="emailAddress" />
            </form:label>
            <form:input path="email" type="email" class="form-control"
                  placeholder="${emailAddressPlaceholder}" />
            <p class="help-block">
                  <spring:message code="emailAddressHelp" />
            </p>
      </div>

      <div class="form-group">
            <spring:message code="namePlaceholder" var="namePlaceholder" />
            <form:label path="name">
                  <spring:message code="name" />
            </form:label>
            <form:input path="name" class="form-control" type="fullname"
                  autocomplete="off" placeholder="${namePlaceholder}" />
            <p class="help-block">
                  <spring:message code="nameHelp" />
            </p>
      </div>

      <div class="form-group">
            <spring:message code="passwordPlaceholder"
                  var="passwordPlaceholder" />
            <form:label path="password">
```

```
                    <spring:message code="password" />
                </form:label>
                <form:password path="password" class="form-control"
                    placeholder="${passwordPlaceholder }" />
        </div>

        <button type="submit" class="btn btn-default">
                <spring:message code="submitSignUp" />
        </button>
</form:form>
```

Se puede observas cómo todas las etiquetas se encuentran dentro de la etiqueta `<form:form>`. Los atributos path contienen nombres de atributos de un objeto SignupForm. El botón de tipo "submit" enviará el formulario en una petición. En este caso la petición se dirige a la misma URL pero con el método POST.

Para recibir esta petición hay que implementar un método de controlador que pueda gestionarlo:

```
@RequestMapping(value = "/signup", method = RequestMethod.POST)
public String signup(@ModelAttribute("signupForm") SignupForm signupForm) {
        userService.signup(signupForm);
        return "redirect:/";

}
```

El controlador, mediante la anotación `@ModelAttribute` inyecta el objeto del formulario indicado. En este caso se pasa este objeto al servicio de usuarios, que utilizará sus atributos para crear un nuevo usuario. En cuanto el objeto es utilizado por el servicio, el traspaso a la base de datos es trivial al usar los repositorios de Spring.

Sin embargo, en ningún momento se está comprobando si los datos están introducidos correctamente. De esta forma se podrían enviar campos vacíos o en este ejemplo incluso podrían haber intentos de registro de varios usuarios con el mismo correo electrónico.

6.3. Validación en modelos

Spring permite la posibilidad de utilizar anotaciones de javax.validation.constraints para comprobar las limitaciones sobre campos de un modelo.

- `@NotNull`
- `@Size:` Indica la longitud mínima y máxima.
- `@Pattern`: Permite la validación mediante expresiones regulares.
- `@Min:` Indica el mínimo de un campo numérico.

- `@Max:` Indica el máximo de un campo numérico.

Estas anotaciones añaden un mensaje por defecto indicando cual es el error que está ocurriendo. Se puede añadir un mensaje mediante el atributo "message". Para poder integrar el mensaje con i18n, el validador por defecto utiliza los archivos "validationMessages.properties". En este caso, el mensaje introducido es la clave del mensaje localizable entre llaves.

La clase SignupForm utilizada en el ejemplo quedaría modificada como se muestra a continuación:

```
package com.welbits.spring.dto;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

public class SignupForm {
	@NotNull
	@Size(min=1, max=100, message="{emailLengthError}")
	@Pattern(regexp="[A-Za-z0-9._%-+]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}",
message="{emailPatternError}")
	private String email;

	@NotNull
	@Size(min=1, max=100, message="{nameLengthError}")
	private String name;

	@NotNull
	@Size(min=6, max=30, message="{passwordLengthError}")
	private String password;

	…
}
```

Una vez introducidas las anotaciones necesarias, hay que configurar el método del controlador que recibe este objeto para que lo valide. El primer paso es añadir la anotación `@Valid` delante del modelo que se quiere validar. Una vez hecho esto se puede acceder al objeto BindingResult en este método. Este objeto es inyectado debido al módulo de Spring Validation. Podemos consultarle a este objeto si el validador ha encontrado algún error.

```
@RequestMapping(value = "/signup", method = RequestMethod.POST)
public String signup(@ModelAttribute("signupForm") @Valid SignupForm
signupForm, BindingResult result) {
	if (result.hasErrors()) {
		return "signup";
	}
	userService.signup(signupForm);
	return "redirect:/";

}
```

En este caso, si se detecta el error se devuelve la misma vista del formulario. Esta vez el formulario tendrá los datos del último envío ya que se propaga el modelo hasta este punto. Además, el modelo tendrá los campos de error relacionados con cada uno de los atributos. Es necesario actualizar la vista para que se muestren estos errores si ocurren. Así se ofrece retroalimentación al usuario, ofreciendo una calidad de usabilidad adecuada.

En JSP, con la librería forms de Spring, se utiliza la etiqueta error. Con el atributo "path" se indica el campo de los errores que se quiere indicar. Con "path" a "*", se especifica que se quiere mostrar todos los errores, mientras que no especificar nada significa que sólo mostrará errores de objeto. Existe también un atributo "cssClass" que sirve para indicar la clase que se quiere asociar a la etiqueta generada por el error.

```
<div class="form-group">
      <spring:message code="emailAddressPlaceholder"
            var="emailAddressPlaceholder" />
      <form:label path="email">
            <spring:message code="emailAddress" />
      </form:label>
      <form:input path="email" type="email" class="form-control"
            placeholder="${emailAddressPlaceholder}" />
      <form:errors cssClass="text-danger" path="email" />
      <p class="help-block">
            <spring:message code="emailAddressHelp" />
      </p>

</div>
```

En el ejemplo sólo se mostrarán en pantalla los errores sobre el campo email, si los hay.

Como se puede ver, Spring ofrece objetos pre-configurados que se inyectan automáticamente, permitiendo empezar a desarrollar una aplicación en poco tiempo sin necesidad de montar en este caso un sistema de validación. Como ocurre con todos los objetos pre-configurados, las opciones son simples y rápidas pero en aplicaciones complejas esto no es suficiente. Una vez más, es posible reemplazar el objeto de validación por un objeto que herede de SpringValidatorAdapter.

6.4. Validador personalizado

La forma más común de implementar un validador personalizado es heredando de la clase LocalValidatorFactoryBean. Esta clase unifica las interfaces Validator de JavaX, Spring y la interfaz ValidatorFactory de JavaX también. Es necesario marcar la nueva clase con la anotación `@Component` para que Spring la añada en el contenedor de IoC. Los métodos necesarios para utilizar el validador son los de la interfaz de Spring: supports y validate.

El método supports sirve para indicar la clase del objeto que se quiere validar y se compara con la clase asignada. La documentación de Spring menciona que típicamente su implementación es:

```
return Foo.class.isAssignableFrom(clazz);
```

El método validate se llama cuando el objeto ya se ha validado si tiene anotaciones de JavaX. Aquí llega el objeto, los errores ya encontrados y las pistas que se van a ofrecer al usuario. Este es un ejemplo de un validador personalizado para la clase SignupForm:

```
package com.welbits.spring.validators;
…
@Component
public class SignupFormValidator extends Utf8Validator {

	private UserRepository userRepository;

	@Resource
	public void setUserRepository(UserRepository userRepository) {
		this.userRepository = userRepository;
	}

	@Override
	public boolean supports(Class<?> clazz) {
		return clazz.isAssignableFrom(SignupForm.class);
	}

	@Override
	public void validate(Object obj, Errors errors,
				final Object... validationHints) {
		super.validate(obj, errors, validationHints);

		if (!errors.hasErrors()) {
			SignupForm signupForm = (SignupForm) obj;
			User user = userRepository
					.findByEmail(signupForm.getEmail());
			if (user != null)
				errors.rejectValue("email", "emailNotUnique");

		}
	}

}
```

En este caso se comprueba primero si el formulario cumple con las condiciones mínimas para iniciar un registro, ya que si no es así la página mostrará esos errores antes de intentar registrar al usuario. Si el formulario es correcto, se comprueba si la dirección de correo electrónico ya existe en la base de datos. Si el usuario ya existe, se añade un error específico para el campo "email" del formulario y la clave del mensaje.

Spring Framework ofrece la clase ValidationUtils con métodos estáticos que están pensados para ser utilizados en los validadores personalizados. Los métodos que incluye son comprobantes de si un campo está vacío o si está vacío o sólo tiene espacios en blanco. Principalmente rechazan el campo indicado si el valor devuelto es "true" e indican un mensaje.

Una desventaja importante a tener en cuenta es que el validador utilizado por una clase que hereda de LocalValidatorFactoryBean no utiliza la misma clase MessageSource que el resto de la aplicación. Este objeto tiene su propio archivo de mensajes localizados "validationMessages.properties" y no utiliza UTF-8. La solución más rápida es utilizar el método setValidationMessageSource para indicar el mismo objeto MessageSource que se utiliza para el resto de la aplicación. Esto centraliza los ficheros de localización y además incluirá codificación en UTF-8 si se ha configurado como se explica en el apartado 5.

```
package com.welbits.spring.validators;
…
public class Utf8Validator extends LocalValidatorFactoryBean {
	@Autowired
	@Override
	public void setValidationMessageSource(MessageSource messageSource) {
		super.setValidationMessageSource(messageSource);
	}

}
```

La clase `Utf8Validator` se encarga de implementar este método, de forma que si el resto de validadores de la aplicación heredan de este validador, todos utilizarán el mismo objeto `MessageSource`.

El último paso es asociar el validador a los atributos de modelo utilizados en las vistas. Para esto se utiliza la anotación `@InitBinder`:

```
@Controller
public class RootController {
	…
	@Autowired
	private SignupFormValidator signupFormValidator;

	@InitBinder("signupForm")
	protected void initSignupBinder(WebDataBinder binder) {
		binder.setValidator(signupFormValidator);
	}
	…

}
```

6.5. Atributos de redirección

Cuando se completa una acción iniciada por el usuario pero acabada por el programa, es buena práctica ofrecerle información sobre el estado final de dicha acción. Al igual que se le enviaban mensajes de error, se le pueden enviar mensajes de éxito o avisos neutros para completar el ciclo de interacción.

Uno de los sitios con mayor necesidad de enviar estos mensajes es tras procesar los formularios ofrecidos al usuario. Es común que, tras procesar los datos correctamente, la aplicación web se dirija a una nueva URL. Para esto, Spring permite que los controladores envíen atributos de redirección mediante la clase RedirectAttributes. Esta clase utiliza su método addFlashAttribute para enviar un valor y un nombre con el cual se puede acceder al atributo en las vistas.

Se pueden utilizar estos atributos para enviar mensajes y estados para que la siguiente vista sea capaz de visualizarlo independientemente de la dirección de la que procedan.

```
@RequestMapping(value = "/signup", method = RequestMethod.POST)
public String signup(HttpServletRequest request,
            @ModelAttribute("signupForm") @Valid SignupForm signupForm,
            BindingResult result, RedirectAttributes redirectAttributes) {
      if (result.hasErrors()) {
            return "signup";
      }

      userService.signup(signupForm);
      Utils.flash(redirectAttributes, "success", "signupSuccess",
                  localeResolver.resolveLocale(request));
      return "redirect:/";

}
```

Este ejemplo utiliza el método flash, que encapsula las siguientes acciones:

```
public static void flash(RedirectAttributes redirectAttributes,
            String flashClass, String messageKey, Locale locale) {
      redirectAttributes.addFlashAttribute("flashClass", flashClass);
      redirectAttributes.addFlashAttribute("flashMessage",
                  Utils.getMessage(messageKey, locale));

}
```

Añade dos atributos, uno especifica el nombre de la clase del elemento en el que se va a mostrar el mensaje y el otro es el mensaje localizado. Enviar una clase permite cambiar ciertos aspectos de como se muestra el mensaje.

Para terminar de implementar esta funcionalidad, se puede crear un archivo .jsp que contenga un contenedor de mensajes genérico de manera que se pueda integrar en cualquier vista con la etiqueta "include" de JSP.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<c:if test="${not empty flashMessage}">
      <div class="alert alert-${flashClass} alert-dismissable">
            <button type="button" class="close" data-dismiss="alert"
                  aria-hidden="true">&times;</button>
            ${flashMessage}
      </div>

</c:if>
```

Debido a que la implementación envía una pareja de atributos, se puede comprobar si existe uno de ellos para renderizar o no el contenedor del mensaje.

7. SEGURIDAD

7.1. Introducción

La mayoría de páginas web tienen sistema de usuarios, dando a cada usuario un perfil. Cada perfil tiene su nivel de acceso, no todos los usuarios pueden acceder a todas las secciones de una aplicación web. Además, cada usuario debe verificar su identidad. Spring Security ofrece medidas de autenticación y control de acceso para las aplicaciones Java.

Como ocurre con el resto de módulos de Spring, Spring Security se adapta a los estándares de seguridad para que su configuración se base principalmente en especificar requisitos propios de la aplicación. Spring Boot ofrece una configuración inicial que permite utilizar Spring Security al instante.

7.2. Uso básico

Para integrar Spring Security con Maven se debe introducir la siguiente dependencia:

```
<dependency>
	<groupId>org.springframework.security</groupId>
	<artifactId>spring-security-web</artifactId>
	<version>3.2.6.RELEASE</version>
</dependency>
```

Sin embargo, en un proyecto con Spring Boot, podemos integrar su módulo preconfigurado:

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Con el módulo cargado, se puede codificar una clase de configuración que de las especificaciones de acceso para la página web en cuestión:

```
@Configuration
@EnableWebMvcSecurity
public class WebSecurityConfig  extends WebSecurityConfigurerAdapter {

   @Resource
      private UserDetailsService userService;

   @Override
   protected void configure(HttpSecurity http) throws Exception {
       http
           .authorizeRequests()
               .antMatchers("/",
```

```
                        "/home",
                        "/error",
                        "/signup",
                        "/forgot-password",
                        "/reset-password/*",
                        "/public/**").permitAll()
                .anyRequest().authenticated();
        http
             .formLogin()
                 .loginPage("/login")
                 .permitAll().and()
             .logout()
                 .permitAll();
    }

    @Autowired
    @Override
    protected void configure(AuthenticationManagerBuilder authManagerBuilder)
              throws Exception {
        authManagerBuilder.userDetailsService(userService);
    }
}
```

`@EnableWebMvcSecurity` integra esta clase de configuración de seguridad con Spring MVC. La clase hereda de la clase abstracta `WebSecurityConfigurerAdapter`, que sirve de base y ofrece métodos que se pueden sobrecargar.

Spring genera un objeto de la clase `HttpSecurity` y lo envía como parámetro en el método `configure(HttpSecurity http)` donde se especifica qué acciones de seguridad deben aplicarse en esta página web. Se indican las peticiones que están autorizadas mediante el método `authorizeRequests()` del objeto de la clase `HttpSecurity`. Se especifican URLs básicas mediante antMatchers, que mapea patrones de URL de tipo Ant. La llamada al método `permitAll()` indica que cualquiera está autorizado a ver dichas URL. Se utiliza `anyRequest().authenticated()` para obligar a que el resto de peticiones sólo sean accesibles mediante autenticación.

El siguiente bloque de código especifica en qué URL se encuentra el formulario de inicio de sesión, y da acceso a cualquier usuario. La aplicación se dirigirá automáticamente a esta página siempre que se necesite autenticación. Si no se especifica una página de inicio de sesión, Spring proporciona una página por defecto. También se permite el método de cierre de sesión estándar de Spring a cualquier usuario.

En la documentación de Spring se especifica que la página de login debe contener un formulario (<form>) que genere una petición con ciertos requisitos:

- Debe ser una petición POST
- Debe incluir el parámetro HTTP correspondiente al nombre de usuario especificado a través del método `usernameParameter(String)`, "username" por defecto
- Debe incluir el parámetro HTTP correspondiente a la clave especificada a través del método `passwordParameter(String)`, "password" por defecto.
- Debe enviarse a la dirección especificada en `loginProcessingUrl(String)`, el propio login por defecto.

También proporciona mensajes de confirmación y error que se pueden utilizar para avisar al usuario. El método `logout()` tiene por defecto que una petición POST a "/logout" ejecutará un cierre de sesión y rediccionará a "/login" con un mensaje de aviso.

El otro método `configure(AuthenticationManagerBuilder authManagerBuilder)` tiene como parámetro un objeto de la clase `AuthenticationManagerBuilder`. Este builder genera el objeto utilizado para comprobar si las credenciales ofrecidas por el usuario son correctas. Se está inyectando un objeto de la clase `UserDetailsService` para aplicar al builder.

A continuación se muestra un ejemplo de un objeto que hereda de `UserDetailsService` e implementa el método `loadUserByUserName`, que devuelve los detalles del usuario mediante un objeto `UserDetails`:

```
@Service
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class UserServiceImpl implements UserService, UserDetailsService {

	...

	@Override
	public UserDetails loadUserByUsername(String username)
			throws UsernameNotFoundException {
		User user = userRepository.findByEmail(username);
		if (user == null)
			throw new UsernameNotFoundException(username);

		return new UserDetailsImpl(user);
	}
}
```

El método intenta encontrar al usuario especificado mediante nombre de usuario (en este ejemplo el nombre es el correo electrónico), lanzando una excepción si no se encuentra en el repositorio. Si lo encuentra, se devuelve un objeto que implemente la interfaz `UserDetails`, creado a partir del usuario encontrado.

Los objetos `UserDetail` son simplemente contenedores con la información del usuario que se encapsulan en objetos del tipo `Authentication`, manteniendo la información no relacionada con la seguridad en un sitio adecuado. Por lo tanto, estos objetos no son utilizados para la seguridad si no para utilizar la información de un usuario autenticado.

```
public class UserDetailsImpl implements UserDetails {

	private static final long serialVersionUID = -4581375328874871489L;
	private User user;

	public UserDetailsImpl(User user) {
		this.user = user;
	}
	@Override
	public Collection<? extends GrantedAuthority> getAuthorities() {
		Collection<GrantedAuthority> authorities =
		new HashSet<GrantedAuthority>(1);
		authorities.add(new SimpleGrantedAuthority("ROLE_USER"));
		return authorities;
	}
	@Override
	public String getPassword() {
		return user.getPassword();
	}
	@Override
	public String getUsername() {
		return user.getEmail();
	}
	@Override
	public boolean isAccountNonExpired() {
		return true;
	}
	@Override
	public boolean isAccountNonLocked() {
		return true;
	}
	@Override
	public boolean isCredentialsNonExpired() {
		return true;
	}
	@Override
	public boolean isEnabled() {
		return true;
	}
}
```

Este ejemplo de implementación es básico. Se define una cuenta que siempre está habilitada, nunca se bloquea y no expira. La contraseña y el correo provienen del objeto User pasado como parámetro en el constructor. En el método `getAuthorities()` se le añade un perfil con la forma más básica de la interfaz `GrantedAuthority`, `SimpleGrantedAuthority`. Dado que el objeto es serializable, se le asocia un ID autogenerado.

7.3. Spring Security Taglib

Spring security ofrece su propio taglib que permite a los archivos JSP acceder a información de seguridad de la app y permite aplicar restricciones de seguridad. Para integrar el taglib, hay que declararlo en el archivo JSP:

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
```

Se compone de tres *tags* o etiquetas:

- **authorize**: Se utiliza para determinar si los contenidos de la etiqueta deben ser evaluados o no. Por ejemplo:

```
<sec:authorize access="isAuthenticated()">
   Mostrar sólo a los usuarios autenticados.
</sec:authorize>
```

- **authentication**: Da acceso al objeto Authentication que contiene el objeto UserDetails. Por ejemplo, con el modelo anterior se podría acceder al nombre de usuario:

```
<sec:authentication property="principal.user.name" />
```

- **accesscontrollist**: Utilizado con el módulo Access Control List (ACL)

7.4. Cifrado de claves

Uno de los procesos básicos de seguridad contra robo de identidad es el cifrado de las claves de usuario. Spring Security ofrece decodificadores que se pueden aplicar de forma rápida al resto de la aplicación. Para esto hay que utilizar una clase que implemente la interfaz PasswordEncoder, que se utilizará para cifrar la clave introducida a la hora de crear el usuario. Además, hay que pasarselo al AuthenticationManagerBuilder cuando se configura:

```
@Configuration
@EnableWebMvcSecurity
public class WebSecurityConfig  extends WebSecurityConfigurerAdapter {
  ...
  @Bean
  public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
  }
  ...
  @Autowired
  @Override
  protected void configure(AuthenticationManagerBuilder authManagerBuilder)
                                                              throws Exception {
      authManagerBuilder.userDetailsService(userService)
                        .passwordEncoder(passwordEncoder());
  }
}
```

Se utiliza el *bean* creado en la clase anterior para inyectarlo en la implementación de `UserService` y cifrar la clave en el método `signup`:

```
private UserRepository userRepository;
private PasswordEncoder passwordEncoder;
@Autowired
public UserServiceImpl(UserRepository userRepository, PasswordEncoder
                       passwordEncoder) {
      this.userRepository = userRepository;
      this.passwordEncoder = passwordEncoder;
}

@Override
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void signup(SignupForm signupForm) {
      User user = new User();
      user.setEmail(signupForm.getEmail());
      user.setName(signupForm.getName());
      String encodedPass = passwordEncoder.encode(signupForm.getPassword());
      user.setPassword(encodedPass);
      userRepository.save(user);
}
```

7.5. Recordar usuario

Otro concepto básico en la seguridad es la opción de recordar el usuario. Si se recuerda el usuario, la sesión se iniciará automáticamente en el navegador hasta que se cierre sesión voluntariamente (o se borren los datos de sesión asociados). Para conseguir esto es necesario enviar el parámetro HTTP "_spring_security_remember_me" (por defecto) en la petición de inicio de sesión. Se realizan las siguientes modificaciones en la clase de configuración de seguridad:

```
@Configuration
@EnableWebMvcSecurity
public class WebSecurityConfig  extends WebSecurityConfigurerAdapter {
   @Value("${rememberMe.privateKey}")
   private String rememberMeKey;
   ...

   @Bean
   public RememberMeServices rememberMeServices() {
       TokenBasedRememberMeServices rememberMeServices =
       new TokenBasedRememberMeServices(rememberMeKey, userService);
       return rememberMeServices;
   }

   ...

   @Override
   protected void configure(HttpSecurity http) throws Exception {

   ...
```

```
        http
            .formLogin()
                .loginPage("/login")
                .permitAll().and()
            .rememberMe().key(rememberMeKey)
            .rememberMeServices(rememberMeServices()).and()
            .logout()
                .permitAll();
    }
    ...
}
```

Se configura un componente que implemente la clase RememberMeServices. En este caso se utiliza TokenBasedRememberMeServices, que guarda una cookie codificada en base64 indicando el usuario recordado. Para crear este objeto hace falta una clave privada que se almacena en application.properties para configurar con comodidad. El nuevo componente se aplica en el objeto de la clase HttpSecurity mediante el método rememberMe().

8. CORREO INTEGRADO

8.1. Introducción

Este es un breve apartado que, principalmente, muestra cómo integrar un servicio de correo electrónico en Spring. A su vez, es una breve introducción a Spring Context Support, un módulo de Spring que ofrece soporte a librerías externas comúnmente utilizadas. Estas librerías pueden ser para caché (EhCache, Guava, JCache), correo (JavaMail), librerías para tareas realizadas periódicamente (Quartz) y otras más.

Este caso se centra en JavaMail. Se realiza una breve implementación de un servicio SMTP (Simple Mail Transfer Protocol) para mostrar la sencillez del desarrollo gracias al módulo Spring Context Support.

8.2. Implementación

Para empezar, se implementa una clase java con la capacidad de enviar un correo con la ayuda de JavaMailSender:

```
@Component
public class SmtpMailSender {

	@Autowired
	private JavaMailSender javaMailSender;

	@Override
	public void send(String to, String subject, String body)
                                                throws MessagingException {
		MimeMessage message = javaMailSender.createMimeMessage();
		MimeMessageHelper helper;

		helper = new MimeMessageHelper(message, true);
		helper.setSubject(subject);
		helper.setTo(to);
		helper.setText(body, true);

		javaMailSender.send(message);
	}

}
```

Se inyecta `JavaMailSender`, implementado por Spring. El método `send` es muy simple y envía un correo a un solo usuario, añadiendo título y un cuerpo que acepta HTML. Si se quisiera añadir extras, como adjuntos, es posible gracias al `MimeMessageHelper`.

No todo está inicializado correctamente, hay ciertos aspectos tienen que ser configurados en application.properties:

```
spring.mail.host = smtp.gmail.com
spring.mail.username = example@gmail.com
spring.mail.password = *************

spring.mail.properties.mail.smtp.auth = true
spring.mail.properties.mail.smtp.socketFactory.port = 465
spring.mail.properties.mail.smtp.socketFactory.class =
javax.net.ssl.SSLSocketFactory

spring.mail.properties.mail.smtp.socketFactory.fallback = false
```

Las primeras líneas indican la identificación del correo que se va a utilizar en la aplicación. Estos datos pasan por un proceso de autorización antes de ser utilizados. El resto son ajustes a las propiedades de correos SMTP, la configuración utilizada es básica.

9. PERFILES DE SPRING

9.1. ¿Qué son los perfiles?

Es probable que la versión de una aplicación en producción no esté bajo las mismas condiciones que la versión en desarrollo. Cada una se encuentra en un entorno diferente y, aunque es buena práctica asemejar lo máximo posible, hay aspectos que no son iguales. También está el caso en el que se desea que la aplicación se comporte de distinta manera, sea para *testing*, para evitar acciones innecesarias a la hora de desarrollar o para simplificarlas.

Spring permite indicar a la aplicación el entorno en el que se está trabajando mediante perfiles. Los perfiles se configuran en las propiedades de la aplicación, permitiendo activar y desactivar partes de la aplicación mediante anotaciones.

9.2. ¿Cómo se utilizan?

Los perfiles que se quieran utilizar se definen por el propio desarrollador. No hay límite en cuanto al número de perfiles que se puedan crear ni tampoco hay límite de perfiles activos a la vez. Esto último permite bastante flexibilidad respecto a componentes que no siempre actúen acorde al resto.

No es necesario indicar los perfiles que se desea tener en total, tan sólo es necesario indicar los perfiles que están activos en las propiedades:

```
spring.profiles.active: prod
```

Por otro lado, los componentes que quieran separarse según el perfil deben marcarse con la anotación `@Profile`:

```
@Configuration
@Profile("prod")
public class ProductionConfiguration {

    // ...

}
```

Al igual que en las configuraciones, se pueden indicar múltiples perfiles posibles separados por comas. Además se puede utilizar el símbolo "!" delante de un perfil para indicar que ese componente estará activo siempre que ese perfil no esté activo.

9.3. Ejemplo

Para profundizar en su utilidad, se propone el caso en el que el servicio de correo electrónico del apartado anterior se quiere desactivar al estar en un perfil de desarrollo. No se desea enviar correos auténticos durante el desarrollo pero sí se desea ver cuál sería el resultado de enviar dicho correo. El perfil activado en dicho caso es:

```
spring.profiles.active: dev
```

La solución se encuentra en implementar una interfaz para el servicio de correo electrónico, que la implementarán el servicio de producción y el de desarrollo. Esto permitirá inyectar el componente independientemente de su implementación actual.

```
public interface MailSender {

	public abstract void send(String to, String subject, String body)
                                                    throws MessagingException;


}
```

Ahora, si se implementa una clase de configuración para el correo, se puede activar un componente u otro, dependiendo del perfil.

```
@Configuration
public class MailConfig {

    @Bean
      @Profile("dev")
      public MailSender mockMailSender() {
            return new MockMailSender();
      }

      @Bean
      @Profile("!dev")
      public MailSender smtpMailSender() {
            return new SmtpMailSender();
      }

}
```

Esto no es del todo necesario, ya que podría indicarse directamente en cada clase. Sin embargo, es buena práctica tener la configuración de los perfiles para estos dos componentes de forma conjunta, ya que la inclusión de uno implica la exclusión del otro. En caso de cambiar el tipo de perfil de alguno, estará la otra configuración a la vista.

`SmtpMailSender` es la implementación del apartado anterior, sólo que implementa MailSender y ya no se utiliza la anotación de componente, ya que se indica en las configuraciones:

```
public class SmtpMailSender implements MailSender {
	@Autowired
	private JavaMailSender javaMailSender;

	@Override
	public void send(String to, String subject, String body)
                                        throws MessagingException {
		MimeMessage message = javaMailSender.createMimeMessage();
		MimeMessageHelper helper;
		helper = new MimeMessageHelper(message, true);
		helper.setSubject(subject);
		helper.setTo(to);
		helper.setText(body, true);
		javaMailSender.send(message);
	}
}
```

La implementación para el entorno de desarrollo mostraría el correo, por ejemplo, por consola.

```
public class MockMailSender implements MailSender {
    private static final Log log = LogFactory.getLog(MockMailSender.class);

	@Override
	public void send(String to, String subject, String body) {
		log.info("Sending mail to " + to);
		log.info("Subject: " + subject);
		log.info("Body: " + body);
	}
}
```

La utilización, independientemente del perfil escogido, es la misma:

```
@Service
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class UserServiceImpl implements UserService, UserDetailsService {

	@Autowired
	private MailSender mailSender;
	…
	@Override
	@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
	public void signup(SignupForm signupForm) {
		…
		mailSender.send(user.getEmail(),
		    Utils.getMessage("verifySubject", Locale.getDefault()),
		    Utils.getMessage("verifyEmail", Locale.getDefault(),
                                verifyLink));
		…
	}
}
```

Se puede apreciar que no es necesario aplicar ningún cambio, ya que se utiliza un método de la interfaz.

10. CARGA Y DESCARGA DE ARCHIVOS

10.1. Introducción

El envío de archivos entre cliente y servidor no suele ser una operación crítica de una aplicación web, pero sí que forman parte del producto final de forma parcial. Mientras que hay servicios que se basan plenamente en el almacenamiento de archivos o de la distribución de los mismos, otras aplicaciones utilizan estas posibilidades para almacenar información del usuario, como puede ser una foto de perfil u otros datos secundarios.

Procesar estos archivos puede parecer complicado, pero los controladores de Spring permiten manipularlos de la misma forma que se manipula cualquier otro dato de entrada o salida. Como siempre, el tiempo necesario para desarrollar una implementación básica es mínima, permitiendo configurar ciertos aspectos desde las propiedades. Este apartado intenta demostrar la facilidad de conexión entre servicios de carga y descarga de archivos con el resto de una aplicación hecha con Spring Boot.

10.2. Subida de archivos

Para subir archivos, es necesario tener un objeto de la clase `MultipartConfigElement` correctamente configurado. En caso de configurarlo en "web.xml", se utilizaría la etiqueta "<multipart-config>". No obstante, Spring Boot prepara este componente automáticamente y lo registra en el contenedor de Spring. `MultipartConfigElement` es un elemento estándar preparado para Servlet 3.0 que define los límites de los archivos que se suben. Los contenedores más utilizados, como Tomcat y Jetty, son compatibles con este elemento.

Al estar utilizando la arquitectura Spring MVC, la subida de archivos se considera una petición que debe manejar un controlador.

```
@Controller
public class FileUploadController {
      @Value("${wb.fileLocation}")
      private String fileLocation;
      @Value("${wb.fileDirection}")
      private String fileDirection;

    @RequestMapping(value="/upload", method=RequestMethod.POST)
    public @ResponseBody String handleFileUpload(
            @RequestParam("file") MultipartFile file,
            HttpServletRequest request, HttpServletResponse response){
        if (!file.isEmpty()) {
            String name = file.getOriginalFilename();
            try {
                byte[] bytes = file.getBytes();
                BufferedOutputStream stream =
```

```
                        new BufferedOutputStream(
                                new FileOutputStream(
                                        new File(fileLocation + name)));
                    stream.write(bytes);
                    stream.close();

                    return fileDirection + name;
                } catch (Exception e) {
                      response.setStatus(HttpServletResponse.
                                        SC_INTERNAL_SERVER_ERROR);
                    return "Failed to upload " + name + " => " + e.getMessage();
                }
            } else {
                response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
                return "Empty file.";
            }
        }

}
```

Se puede apreciar que el controlador no tiene ningún tipo de configuración especial para ser capaz de procesar un archivo. El método que se encarga de recibir el archivo tampoco necesita configuración específica aparte de ser una petición de tipo POST, pero es importante recibir al archivo con el tipo adecuado. `@RequestParam("file")` es el archivo enviado por parte del cliente, como se verá más adelante, y debe ser de tipo `MultipartFile` para poder ser manipulado.

En este caso, el archivo es almacenado directamente en un directorio específico del servidor, utilizando búferes de Java. En la práctica, se considera mejor práctica utilizar directorios temporales, una base de datos o almacenamiento de tipo NoSQL. Se puede observar también que no existe ningún control sobre el tipo de archivo o si el nombre de archivo ya está siendo utilizado.

Sí que se realizan cambios del estado de la respuesta, indicando correctamente el motivo por el cual no se ha conseguido subir correctamente. En cuanto al objeto `MultipartFile`, éste proporciona información como el nombre del archivo, el tipo del archivo, el tamaño y su propio contenido en bytes.

El directorio donde se almacena el archivo probablemente no sea accesible por parte del cliente. Por esto se definen dos raíces, una interna para el almacenamiento y otra externa para acceder a él a través de la web. Esto es así porque el directorio interno pertenece al directorio de un servidor con acceso público. Las raíces se almacenan en las propiedades de la aplicación para poder configurarlas cómodamente dependiendo del entorno en el que se trabaja.

En cuanto a la vista, se utilizan los formularios de HTML5 para recoger el archivo y cualquier otro dato si fuera necesario. En este caso no se envía ningún parámetro adicional.

```
<form method="POST" enctype="multipart/form-data"
  action="/upload" id="recipeImageForm" name="userImageForm">
      <div class="form-group">
            <spring:message code="fileToUpload"/> <br>
            <input type="file" name="file" class="btn btn-default">
            <br />
            <spring:message code="upload" var="upload"/>
            <input type="submit" class="btn btn-default" value="${upload }">
            <spring:message code="pressHere"/>
      </div>

</form>
```

Esto genera un formulario con un elemento que permite escoger un archivo junto con un botón para realizar la subida. Nada más es necesario en la vista, ya que el controlador se puede encargarse del archivo enviado.

Una vez implementado esto, puede ser necesario cambiar las configuraciones del `MultipartConfigElement` para cambiar los límites de la subida. Las siguientes propiedades pueden configurarse en application.properties:

- multipart.maxFileSize: Por defecto a 128KB. Limita el tamaño total del archivo.
- multipart.maxRequestSize: Por defecto a 128BK. Limita el tamaño máximo de la petición. Esto significa que el tamaño de "multipart/form-data" debe estar por debajo de lo configurado.

10.3. Generación de archivos de descarga

A la hora de descargar archivos desde el servidor, también se pueden utilizar búferes en el controlador para ofrecer el archivo deseado. En este caso, el retorno del método de la petición sería *void*, ya que no devuelve una vista.

Sin embargo, en algunos casos se pueden utilizar librerías adicionales para generar los archivos dinámicamente. En este caso el objetivo es implementar un sistema de generación de archivos de tipo PDF generados a partir de un modelo.

Para esto, se utiliza la librería iText, que permite generar documentos PDF. Se debe añadir la librería al gestor de dependencias.

```
<dependency>
	<groupId>com.lowagie</groupId>
	<artifactId>itext</artifactId>
	<version>1.3</version>

</dependency>
```

Con esta librería reemplazaremos la vista común generada por plantillas. Se utiliza un objeto `ModelAndView` de Spring para enlazar la vista personalizada con el modelo deseado:

```
@RequestMapping(value = "/pdf/{recipeId}", method = RequestMethod.GET)
public ModelAndView pdf(@PathVariable long recipeId,
                        HttpServletRequest request) {
	Recipe recipe = recipeService.findById(recipeId);
    return new ModelAndView(new RecipePdfView(
                localeResolver.resolveLocale(request)), "recipe", recipe);

}
```

El modelo `ModelAndView` requiere la vista, el modelo y el nombre por el cual se mapea el modelo.

La vista personalizada extiende de `AbstractPdfView`. Aquí es donde se genera el PDF en base al modelo asociado:

```
public class RecipePdfView extends AbstractPdfView {
	private Locale loc;

	public RecipePdfView(Locale loc) {
		this.loc = loc;
	}
	@Override
	protected void buildPdfDocument(Map<String, Object> model,
			Document document, PdfWriter writer,
			HttpServletRequest request,
			HttpServletResponse response) throws Exception {

		Recipe recipe = (Recipe) model.get("recipe");

		Font titleFont = new Font(2);
		titleFont.setSize(30);

		Font headerFont = new Font(2);
		headerFont.setSize(20);

		document.open();
		Paragraph title = new Paragraph(recipe.getName(), titleFont);
		title.setSpacingAfter(10);
		document.add(title);
		document.add(new Paragraph(recipe.getDescription()));
		…
		document.add(
			new Paragraph(Utils.getMessage("steps", loc), headerFont));
		List stepsList = new List(true, 20);
```

```
            for (Step step : recipe.getSteps()) {
                  String l = step.getText();
                  stepsList.add(new ListItem(l));
            }
            document.add(stepsList);

            document.close();
      }
}
```

Se inicializa el objeto con la localización para facilitar la internacionalización de la información escrita en el PDF. El método `buildPdfDocument` se llama automáticamente gracias a la clase heredada. Esto permite centrarse únicamente en el documento.

La creación de la vista se basa en añadir elementos al objeto `Document`. Estos elementos pueden ser párrafos, listas y otros elementos comunes. A todos los elementos se les puede asociar un tipo de fuente, donde se configuran tamaños y colores. Los elementos también tienen aspectos configurables como márgenes y alineaciones. Al documento se le puede añadir metadatos como el autor y nombre de archivo.

Producir un documento complejo puede resultar costoso, pero para representaciones simples es una solución rápida.

11. API REST

11.1. Introducción

La Transferencia de Estado Representacional (Representational State Transfer), conocida como REST, es una técnica de arquitectura software para diseñar sistemas distribuidos. REST fue introducido en el año 2000 por Roy Fielding. Más que un estándar, es una serie de obligaciones como la carencia de estado, la relación entre cliente y servidor y tener una interfaz uniforme. Aunque REST no se limita a HTTP, se usa comúnmente en este dominio.

Precisamente, los principios de REST se resumen en:

- Los recursos son accesibles a través de un directorio de URIs estructurado.
- Los datos se representan a través de JSON o XML, reflejando objetos y sus atributos.
- Operaciones definidas aplicadas a todos los recursos que simulan operaciones de tipo CRUD.
- Las interacciones no mantienen estados ni contexto del cliente.

En HTTP, las operaciones aplicables son las siguientes peticiones:

- GET: Para recibir información.
- POST: Sirven para aplicar una acción a una entidad, como crearla o modificarla.
- PUT: También sirve para crear o actualizar una entidad, con la diferencia de que esta petición es idempotente.
- PATCH: Actualiza campos específicos de una entidad.
- DELETE: Solicita la eliminación de un recurso.

Las peticiones HTTP de estas operaciones siguen, claro está, los código de estado HTTP. Es importante tenerlos en cuenta ya que es la única forma de saber si la acción solicitada ha sido exitosa o si, en cambio, hay que realizar una acción del tipo "rollback". Los códigos siguen una agrupación general:

- 1XX: Información.
- 2XX: Éxito.
- 3XX: Redirección.

- 4XX: Error del cliente.
- 5XX: Error del servidor.

También se pueden utilizar las cabeceras "Accept" y "Content-Type" para asegurar el formato de los datos. El primero lo utiliza el cliente para solicitar el tipo mientras que el segundo especifica el tipo en el que realmente se está enviando la información. Los más vistos en APIs REST son "application/json" y "application/xml".

11.2. Implementación de la interfaz

Aplicar REST en una API de Spring es sencillo, ya que se puede utilizar el mismo ORM utilizado en el modelo MVC implementado para la aplicación web. Utilizar Spring exclusivamente para una API de tipo REST también es posible ya que se puede implementar el ORM específicamente para usar con el servicio API.

Como se ha mencionado en el apartado sobre controladores, existe una anotación especial para este tipo de servicios: `@RestController`

Esta anotación evitaba tener que indicar mediante anotaciones que el resultado de la operación no es una vista si no el cuerpo de una respuesta.

El controlador implementado servirá de interfaz para definir las distintas operaciones que el servicio REST permitirá realizar. A continuación se muestra un ejemplo simple que permite recibir entidades del tipo `User` y entidades del tipo `Recipe`.

```
package com.welbits.spring.core;
…
@RestController
@RequestMapping("/api")
public class RestApiController {
    @Autowired
    private RecipeService recipeService;
    @Autowired
    private UserService userService;

    @RequestMapping(value = "/recipe/{recipeId}",
                    method = RequestMethod.GET)
    public Recipe viewRecipe(@PathVariable long recipeId) {
      Recipe recipe = recipeService.findById(recipeId);
      return recipe;
    }
    @RequestMapping(value = "/user/{userId}", method = RequestMethod.GET)
    public User viewUser(@PathVariable long userId) {
      User user = userService.findById(userId);
      return user;
    }
}
```

Como se puede observar, el tratamiento es simple ya que no se debe modificar el estado. Se hace uso de los repositorios de cada entidad para obtener un objeto de cierto tipo dado un identificador para cada petición de tipo GET. La anotación `@RequestMapping("/api")` reserva un espacio específico de URIs para el servicio. La respuesta por defecto de Spring es de tipo JSON.

El mapeo de direcciones para las peticiones es importante, ya que es la base de uso del servicio y es uno de los principios REST. Es buena práctica utilizar directorios específicos para cada tipo de entidad y, dentro de ellos, directorios distintos para cada tipo de acción.

Por último, el controlador debe encargarse también del estado de la respuesta, el ejemplo previo no lo hace, pero se puede modificar:

```
@RequestMapping(value = "/user/{userId}", method = RequestMethod.GET)
public User viewUser(@PathVariable long userId, HttpServletResponse
response) {
	User user = userService.findById(userId);
	if (user == null) {
		response.setStatus(HttpServletResponse.SC_NOT_FOUND);
	}
	return user;

}
```

Se ha visto previamente que inyectando el objeto `HttpServletResponse` se puede cambiar el código de estado. En este caso se comprueba si se ha encontrado o no un objeto bajo el identificador ofrecido, resultando en un error de tipo 404 (Not Found) en caso de no encontrar dicho objeto.

11.3. Modificaciones del modelo

En principio no es necesario modificar los modelos, ya que Spring se encarga de transformarlos a formato JSON tratándose de objetos serializables. Sin embargo, dada la naturaleza de la arquitectura, los objetos se relacionan entre sí.

La relación entre objetos puede llevar a una referencia circular. Es decir, si el objeto "Recipe" contiene una lista de objetos "Step", el primero tendrá una referencia a cada uno de los objetos de la lista y, a su vez, cada uno de ellos tendrá una referencia al objeto de tipo "Recipe". Al intentar serializar, se entraría en un bucle ya que un objeto añade al otro.

Para evitar esto se puede utilizar una serie de anotaciones. La más común es `@JsonIgnore`, que directamente indica que el atributo no debe ser tomado en cuenta en procesos de serialización. En el ejemplo previo, esta anotación deberá ir en el atributo de tipo "Recipe" en el objeto "Step", ya que la entidad principal (y la que se quiere enviar) es Recipe.

Esta no siempre es la solución, a veces la intención es incluir ambas entidades en el servicio REST. Se pueden utilizar las anotaciones `@JsonManagedReference` y `@JsonIgnore` para solventar problemas de referencia bi-direccional.

11.4. Consideraciones

Si se trata de una aplicación con seguridad que además contiene un servicio REST, el acceso a este debe tratarse individualmente.

Un posible caso es que las operaciones son simplemente peticiones de tipo GET y los datos no son sensibles, en este caso se puede hacer la dirección accesible a todos mediante la configuración de seguridad de Spring.

Si este no es el caso, tampoco se puede utilizar el acceso que se utiliza en la web, ya que no hay interfaces. Se pueden utilizar sistemas como *OAuth* u otros sistemas simples basados en *tokens*.

Una práctica simple es asociar tokens de acceso a la API a cada usuario. Si el usuario lo solicita, se le asocia un token único. Este token debe enviarse con cada petición, indicando la identidad del solicitante y permitiendo la decisión sobre la información que este puede recibir.

12. SPRING ACTUATOR

12.1. Monitor de aplicaciones

Actuator es un submódulo que permite añadir direcciones de la aplicación que dan acceso a información sobre la información. Esto permite monitorizarla y, en ciertos casos, interactuar. Spring Boot ofrece una serie de direcciones por defecto. Con esta información se puede saber el estado de muchos componentes configurados de forma automática, facilitando la inspección del código y el comportamiento de la aplicación.

Para utilizar Spring Actuator con Spring Boot hay que añadir la siguiente dependencia:

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>
```

Una vez integrado, se pueden utilizar las siguientes direcciones por defecto:

- autoconfig: Muestra todos los componentes de auto-configuración indicando por qué han sido aplicados o por qué no.
- beans: Muestra una lista con todos los Beans de Spring de la aplicación.
- configprops: Lista de `@ConfigurationProperties`
- dump: Realiza un volcado de los hilos en funcionamiento.
- env: Muestra propiedades del entorno. Estas propiedades se almacenan en ConfigurableEnvironment, proporcionado por Spring.
- health: Muestra información del estado actual de la aplicación.
- info: Muestra información de la aplicación.
- metrics: Muestra datos estadísticos de la aplicación.
- mappings: Muestra el mapeo de las direcciones anotadas con `@RequestMappings`
- shutdown: Permite detener la aplicación (no está activado por defecto).
- trace: Muestra por defecto las últimas peticiones HTTP.

Hay que tener en cuenta que estas direcciones se reservan para dichas llamadas. Se puede añadir una raíz a la dirección en application.properties.

```
management.contextPath: /manage
```

Así se consigue que la aplicación mapee estas direcciones a un directorio que no se va a utilizar para el resto de la web. Por ejemplo, en vez de acceder a "/health", se debe acceder a "/manage/health".

13. DESPLIEGUE DE APLICACIONES

13.1. Introducción

Una vez obtenida una versión de una aplicación que se pueda lanzar al mercado, se debe llevar el proyecto a un entorno de producción. Normalmente, una aplicación web se aloja finalmente en un servidor en el cual todo el público tiene acceso.

Una aplicación hecha con Spring debe compilarse y empaquetarse en un archivo WAR. WAR es una versión web de un archivo JAR, preparado para alojarse en un *servlet* o contenedor de JSP. Este apartado explica, con un ejemplo, que pasos hay que seguir para lograr tener una aplicación en funcionamiento en un entorno de producción.

13.2. Configuración externa

Aspectos como el servidor utilizado, la base de datos, claves privadas y credenciales normalmente cambian al cambiar de entorno. Con Spring, se puede almacenar toda esta información en el archivo application.properties, donde debe ir toda la configuración. Una de las características de las propiedades es que se pueden configurar externamente, permitiendo utilizar un único archivo WAR para múltiples entornos.

En el caso de alojar la aplicación en un servidor propio, la forma de sobrescribir los datos de configuración internos es añadiendo un archivo application.properties en el mismo directorio que el archivo WAR. De esta forma, toda propiedad indicada en este fichero externo tendrá prioridad sobre las propiedades indicadas internamente, por lo tanto permitiendo modificar la configuración sin necesidad de empaquetar una nueva aplicación.

Si se elige alojar la aplicación en un servicio Paas (Platform-as-a-Service), es probable que el propio servicio ofrezca la posibilidad de editar estas configuraciones dentro de su plataforma.

13.3. Despliegue con Pivotal Cloud Foundry

Generalmente, hay tres tipos de opciones para alojar una aplicación:

- Tener un servidor físico propio o alquilado en un centro de datos.
- Utilizar un servicio de alojamiento en la nube como Amazon Web Service.
- Alojamiento en un servicio PaaS, como Pivotal Cloud Foundry o Heroku.

Los servicios PaaS normalmente son los que consiguen un resultado de la forma más rápida, aunque también son los menos flexibles. No obstante, es una opción económica, ofreciendo escalabilidad de forma dinámica cambiando el coste según la configuración.

En caso de utilizar Pivotal Cloud Foundry, el primer paso es crear un usuario en la página oficial (https://run.pivotal.io), crear una "org" (organización) y dentro crear un "space", donde se alojará la aplicación. Lo siguiente es instalar CF CLI (Cloud Foundry Command Line Interface), con el cual se subirán las aplicaciones.

A continuación vienen una serie de conceptos que deben cambiarse en general (independientemente del tipo de alojamiento) antes de generar el archivo WAR:

- En el archivo pom.xml, indicar la versión actual de la aplicación.
- Para algunos servidores, es necesario activar el "RemoteIpValve" de tomcat:

```
server.tomcat.remote_ip_header=x-forwarded-for
server.tomcat.protocol_header=x-forwarded-proto
```

- Si se desea, se puede forzar el uso de https en el entorno de producción en la configuración de seguridad:

```
@Configuration
@EnableWebMvcSecurity
public class WebSecurityConfig  extends WebSecurityConfigurerAdapter {
    @Value("${spring.profiles.active}")
    private String env;
    @Value("${rememberMe.privateKey}")
    private String rememberMeKey;

    …

    @Override
    protected void configure(HttpSecurity http)
                                        throws Exception {
        …
        if (env.equals("prod"))
            http.requiresChannel()
                                    .anyRequest()
                                    .requiresSecure();
    }
    …

}
```

Lo siguiente es generar el WAR (o JAR, si se desea) y subirlo. En el CF CLI, primero hay que iniciar sesión:

```
cf login -a https://api.run.pivotal.io
```

Luego hay que introducir la siguiente línea para subirlo:

```
cf push <nombre-app> -p  <directorio-war>/<archivo-war>
```

Donde:

- <directorio-war> es el directorio en el que se encuentra el archivo WAR o JAR de la aplicación.
- <nombre-app> es el nombre utilizado para referirse a la aplicación en Pivotal, también se utiliza en la url.
- <archivo-war>: El nombre del archive WAR o JAR generado
- --no-start se puede añadir al final de la línea si no se desea iniciar la aplicación en este momento.
- Esta misma línea sirve para actualizar la aplicación

Una vez subida la aplicación, es necesario enlazarla a un servicio de bases de datos para almacenar los modelos. Pivotal Cloud Foundry ofrece servicios, entre ellos MySQL (ClearDB).

En este punto, se deben configurar externamente los valores para:

- Actualizar los perfiles activos para un entorno de producción.
- Acceso a la base de datos.
- Direcciones absolutas. (Url de la app, acceso a directorios del servidor, etc)
- Claves privadas, como la de recordar usuario en el apartado de seguridad.
- Credenciales de servicios, como ocurre con el servicio de correo electrónico.

Una vez hecho esto se puede ejecutar la aplicación. En este caso, se utiliza la plataforma web de Pivotal para iniciar y monitorizar la app.

El tipo de alojamiento y la forma en el que se monta el entorno externo depende principalmente de las circunstancias. Mientras tanto, Spring ofrece la máxima flexibilidad posible para adaptarse al entorno sin necesidad de realizar cambios internos.

III. APLICACIÓN PRÁCTICA

1. REQUISITOS

1.1. Especificación

Se va a desarrollar una aplicación web encargada de gestionar una comunidad de usuarios que comparten recetas de cocina. La aplicación será capaz de gestionar tanto a los usuarios que interactúan con la aplicación como a los recursos utilizados para conseguir el objetivo. Los usuarios pueden ser administradores o usuarios comunes. Los recursos son las recetas y sus componentes. Lo usuarios comunes serán capaces de gestionar sus propios datos y recetas y podrán ver los datos y recetas de los demás. El administrador tendrá las capacidades de un usuario común, además de una serie de privilegios adicionales.

Los usuarios podrán realizar las siguientes tareas:

- Registrarse
- Verificar su cuenta
- Ver sus datos y de otros usuarios
- Modificar sus datos
- Crear recetas
- Ver sus recetas y las de los demás usuarios
- Buscar recetas
- Modificar y borrar sus recetas

El administrador podrá realizar las mismas acciones que un usuario común además de las siguientes:

- Ver una lista de usuarios
- Modificar datos de usuarios
- Bloquear y desbloquear usuarios
- Dar y quitar permisos de administración
- Modificar recetas de otros usuarios
- Borrar recetas de otros usuarios

1.2. Casos de uso

1.2.1. Diagrama general de casos de uso

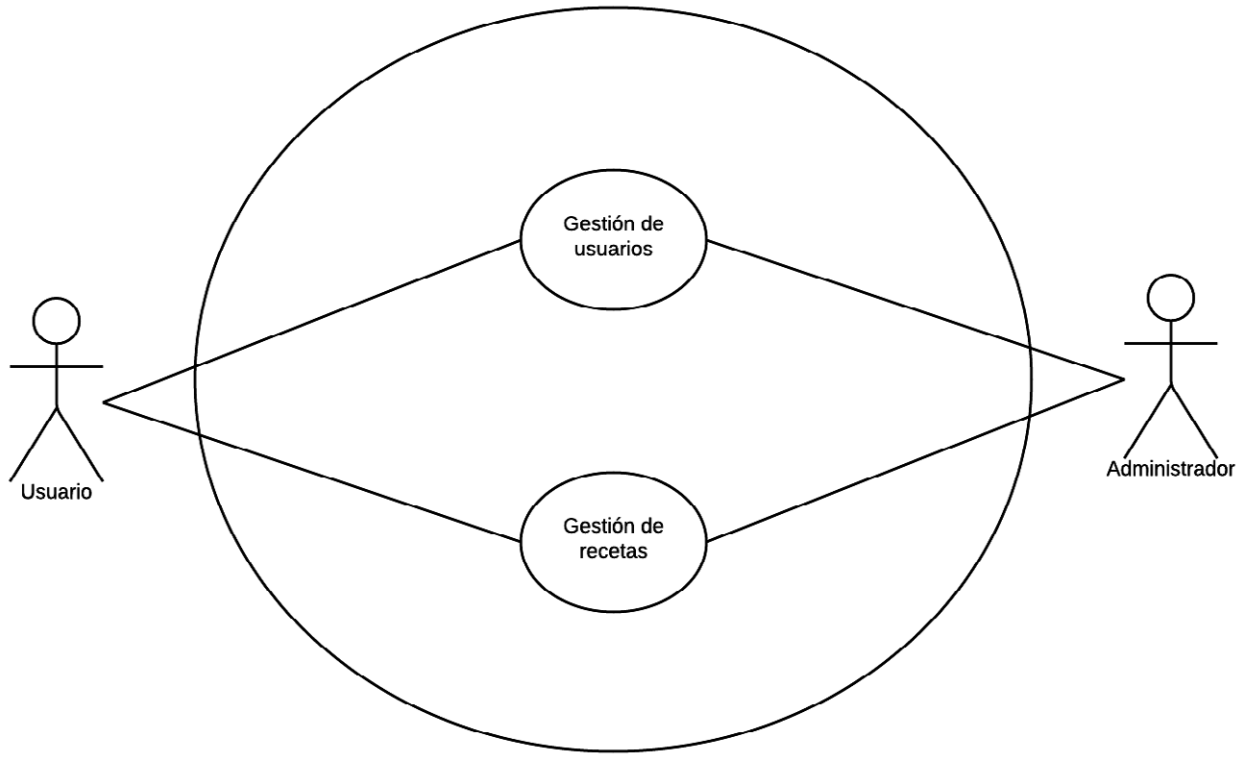
Atendiendo a la especificación de los requisitos, se pueden separar dos grupos de funcionalidades distinguibles. Cada uno de estos grupos representa la gestión de los componentes base de la aplicación: usuarios y recetas.

Por otro lado, se pueden identificar dos actores distintos:

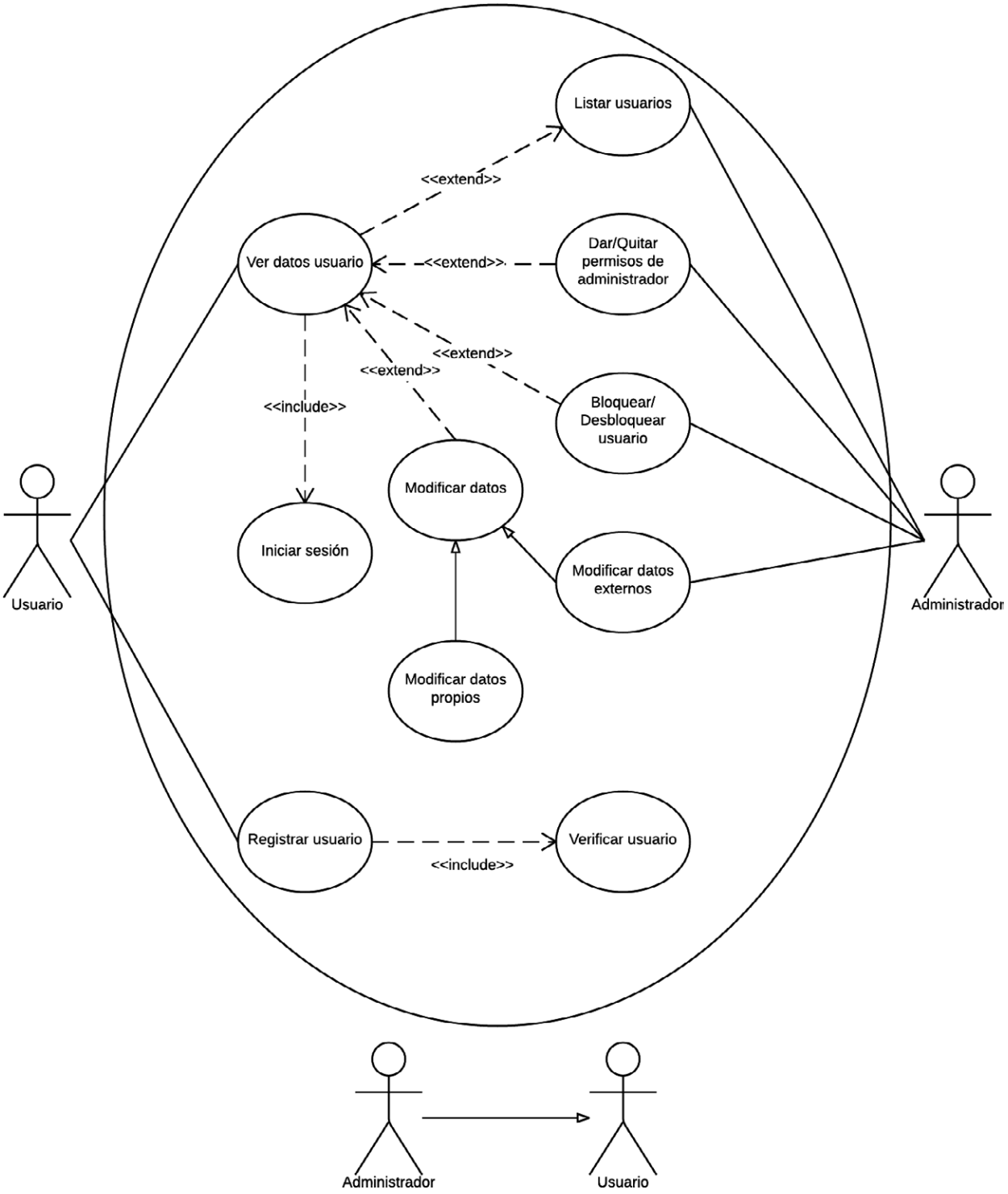
- El usuario común. Representado como "Usuario". Tiene la capacidad de visualización de todos los componentes de la aplicación y la capacidad de modificar sus datos, crear recetas, modificar y borrar recetas propias.

- El usuario administrador. Representado como "Administrador". Tiene la capacidad de actuar como un usuario común además de poder modificar datos de otros usuarios.

Aunque se puede observar una clara herencia, el diagrama general representa a cada actor de forma individual para reflejar que internamente no realizan las acciones por igual.



1.2.2. Diagrama de casos de uso para la gestión de usuarios



Nombre	**Iniciar sesión**
Fecha	12/05/2015
Descripción	Permite autenticar al usuario para el uso de la aplicación.
Actores	Administrador, usuario común.
Precondiciones	El usuario debe estar dado de alta en el sistema.
Flujo normal	1) El actor pulsa sobre el botón de iniciar sesión. 2) El sistema muestra un formulario para introducir nombre de usuario y contraseña. 3) El actor introduce los valores y pulsa sobre un botón para validar. 4) El sistema comprueba que los datos son válidos y genera una sesión. 5) El sistema redirige al usuario a la página que requería una sesión.
Flujo alternativo	4) El Sistema comprueba que los datos no son válidos y envía un mensaje de error al actor.

Nombre	**Registrar usuario**
Fecha	12/05/2015
Descripción	Permite dar de alta a un nuevo usuario de la aplicación.
Actores	Usuario.
Precondiciones	No debe haber una sesión ya iniciada.
Flujo normal	1) El usuario pulsa sobre el botón "registrarse". 2) El sistema muestra un formulario para introducir correo, contraseña y el nombre real del actor. 3) El actor introduce estos valores y pulsa el botón "Registrar". 4) El sistema comprueba la validez de estos datos y los almacena. 5) El sistema envía un correo de verificación al usuario. 6) El sistema redirige a la página de inicio.
Flujo alternativo	4) El Sistema comprueba que el correo ya está siendo utilizado y envía un mensaje solicitando que se utilice uno distinto.
Poscondiciones	Se almacenan los datos del usuario en una base de datos para identificarle en futuros usos y se le añade un rol de "no verificado" hasta que realice la verificación.

Nombre	**Verificar usuario**
Fecha	12/05/2015
Descripción	Permite asegurar que el usuario es propietario del correo utilizado.
Actores	Usuario.
Precondiciones	El usuario se ha registrado, provocando el envío de un correo de verificación.
Flujo normal	1) El actor pulsa sobre el enlace de verificación enviado. 2) El sistema comprueba que el enlace es correcto y solicita al actor que indique sus credenciales. 3) El actor rellena el formulario y lo envía. 4) El sistema comprueba su validez y verifica al usuario.
Flujo alternativo	2) El sistema comprueba que el enlace no es válido y envía un mensaje al actor.
Flujo alternativo	4) El sistema comprueba que los datos no son válidos o que el usuario ya está verificado y se lo notifica al actor.
Poscondiciones	El rol de "no verificado" es retirado para el usuario en cuestión.

Nombre	**Ver datos usuario**
Fecha	12/05/2015
Descripción	Prmite ver todos los datos de un usuario registrado en el sistema.
Actores	Aministrador, usuario.
Precondiciones	El actor está autenticado.
Flujo normal	1) El actor pulsa sobre un enlace que redirige al perfil de un usuario. 2) El sistema reconoce el id del usuario y muestra sus datos (excepto la contraseña).

Nombre	**Listar usuarios**
Fecha	12/05/2015
Descripción	Permite ver un listado de todos los usuarios registrados en la aplicación.
Actores	Administrador.
Precondiciones	El usuario debe estar autenticado y con rol de administrador.
Flujo normal	1) El actor pulsa sobre el botón "Usuarios". 2) El sistema muestra un listado de los usuarios que están dado de alta, mostrando su nombre, correo y botones para visualización y edición de datos.

Nombre	**Dar/Quitar permisos de administrador**
Fecha	12/05/2015
Descripción	Permite cambiar el rol de administrador de otros usuarios.
Actores	Administrador.
Precondiciones	El actor está autenticado y tiene el rol de administrador.
Flujo normal	1) El actor pulsa sobre el botón de dar o quitar permisos de usuario. 2) El sistema reconoce si el usuario tiene el rol o no y realiza la opción opuesta. 3) El sistema cambia el botón en el perfil del usuario para que se pueda realizar la acción opuesta.
Poscondiciones	Se modifican los roles del usuario afectado indicando si es administrador o no.

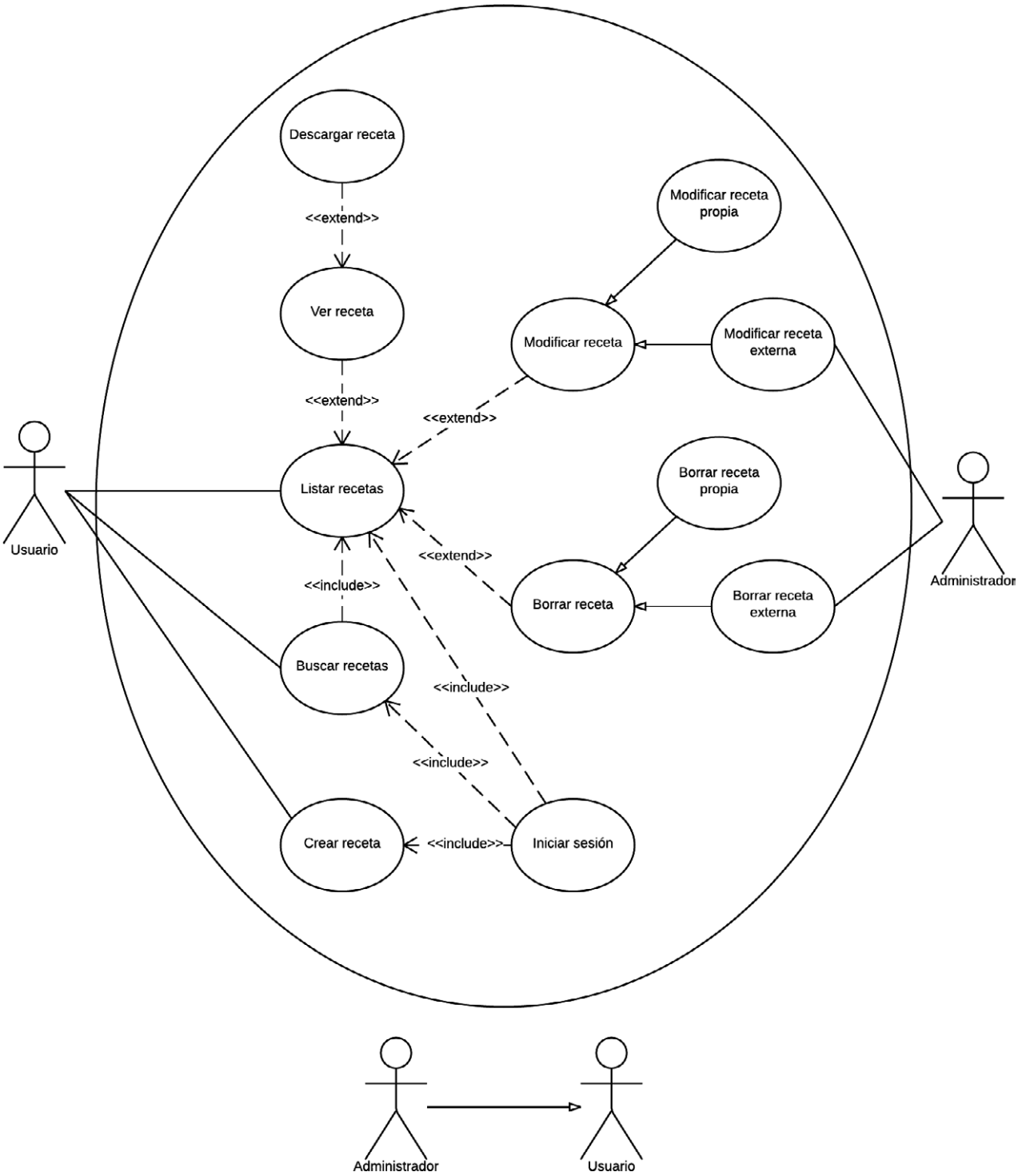
Nombre	**Bloquear/Desbloquear usuario**
Fecha	12/05/2015
Descripción	Bloquear o desbloquear a un usuario del sistema.
Actores	Administrador.
Precondiciones	El actor está autenticado y tiene el rol de administrador.
Flujo normal	1) El actor pulsa sobre el botón bloquear o desbloquear usuario. 2) El sistema añade o elimina el rol de bloqueado del usuario. 3) El sistema muestra el botón opuesto para poder realizar la acción contraria.
Poscondiciones	Se modifican los roles del usuario afectado, permitiéndole acceder o no al sistema.

Nombre	**Modificar datos**
Fecha	12/05/2015
Descripción	Permite modificar los datos de un usuario.
Actores	Administrador, usuario.
Precondiciones	El usuario debe estar autenticado en el sistema.
Flujo normal	1) El actor pulsa sobre el botón "Editar usuario". 2) El sistema muestra un formulario relleno con los datos actuales del usuario. 3) El actor modifica los datos y envía el formulario. 4) El sistema valida los datos y los almacena.
Flujo alternativo	4) El Sistema comprueba que los datos no son válidos y envía un mensaje al actor indicando los errores.
Poscondiciones	Los datos nuevos reemplazan los antiguos datos en la base de datos.

Nombre	**Modificar datos propios**
Fecha	12/05/2015
Descripción	Permite modificar los datos de un usuario.
Actores	Administrador, usuario.
Precondiciones	El usuario debe estar autenticado en el sistema.
Flujo normal	1) El actor pulsa sobre el botón "Editar usuario". 2) El sistema muestra un formulario relleno con los datos actuales del usuario. Los roles no son modificables. 3) El actor modifica los datos y envía el formulario. 4) El sistema valida los datos y los almacena.
Flujo alternativo	4) El Sistema comprueba que los datos no son válidos y envía un mensaje al actor indicando los errores.
Poscondiciones	Los datos nuevos reemplazan los antiguos datos en la base de datos.

Nombre	**Modificar datos externos**
Fecha	12/05/2015
Descripción	Permite modificar los datos de cualquier usuario.
Actores	Administrador.
Precondiciones	El usuario debe estar autenticado en el sistema y debe tener el rol de administrador.
Flujo normal	1) El actor pulsa sobre el botón "Editar usuario". 2) El sistema muestra un formulario relleno con los datos actuales del usuario, incluyendo los roles. 3) El actor modifica los datos y envía el formulario. 4) El sistema valida los datos y los almacena.
Flujo alternativo	4) El Sistema comprueba que los datos no son válidos y envía un mensaje al actor indicando los errores.
Poscondiciones	Los datos nuevos reemplazan los antiguos datos en la base de datos.

1.2.3. Diagrama de casos de uso para la gestión de recetas



Nombre	**Crear recetas**
Fecha	15/05/2015
Descripción	Permite crear una receta para que aparezca en la aplicación.
Actores	Administrador, usuario.
Precondiciones	El usuario debe estar autenticado.
Flujo normal	1) El actor pulsa sobre el botón "Crear receta" 2) El sistema muestra un formulario para introducir nombre, descripción, tiempos, ingredientes y pasos de la receta. 3) El actor rellena el formulario y pulsa el botón "Crear". 4) El sistema comprueba la validez de los datos y crea la receta. 5) El sistema redirige al listado de recetas del usuario, donde podrá ver su nueva receta.
Flujo alternativo	4) El Sistema comprueba que los datos no son válidos y envía un mensaje de error al actor.
Poscondiciones	La receta se almacena en la base de datos y recibe un identificador único.

Nombre	**Buscar recetas**
Fecha	15/05/2015
Descripción	Permite buscar una receta por su nombre.
Actores	Administrador, usuario.
Precondiciones	El usuario debe estar autenticado en el sistema.
Flujo normal	1) El usuario introduce un nombre en el campo "Buscar receta" y pulsa "Buscar". 2) El sistema busca recetas cuyo nombre incluye el valor dado. 3) El sistema muestra los resultados mediante el caso de uso "Listar recetas"

Nombre	**Listar recetas**
Fecha	15/05/2015
Descripción	Permite ver un listado de las recetas disponibles.
Actores	Administrador, usuario.
Precondiciones	El usuario debe estar autenticado en el sistema.
Flujo normal	1) El usuario pulsa sobre el botón "Recetas". 2) El sistema lista todas las recetas, mostrando los botones "Ver", "Editar" y "Borrar" si el usuario tiene permiso para realizar esas acciones para cada receta.

Nombre	**Ver receta**
Fecha	15/05/2015
Descripción	Permite ver la información de una receta.
Actores	Administrador, usuario.
Precondiciones	El usuario debe estar autenticado en el sistema. Se han listado recetas en la acción previa.
Flujo normal	1) El actor pulsa sobre el botón "Ver" en un listado de recetas. 2) El sistema recupera la receta indicada y muestra su información al actor, junto a un botón "Versión descargable".

Nombre	**Descargar receta**
Fecha	15/05/2015
Descripción	Permite descargar un fichero PDF con la receta.
Actores	Administrador, usuario.
Precondiciones	El usuario debe estar autenticado en el sistema. El usuario se encuentra visualizando una receta.
Flujo normal	1) El actor pulsa sobre el botón "Versión descargable". 2) El sistema muestra una nueva vista de un PDF, con la información de la receta en blanco y negro y sin fotografías. 3) El usuario pulsa sobre el botón "Descargar". 4) El sistema descarga el archivo en el ordenador del actor.
Flujo alternativo	4) Se produce un error durante la descarga y el Sistema avisa al actor mediante un mensaje.

Nombre	**Modificar receta**
Fecha	15/05/2015
Descripción	Permite modificar la información de una receta.
Actores	Administrador, usuario.
Precondiciones	El usuario debe estar autenticado en el sistema. El usuario se encuentra en un listado de recetas.
Flujo normal	1) El actor pulsa sobre el botón "Modificar". 2) El sistema muestra un formulario con los datos de la receta actuales. 3) El actor modifica los datos que desea. 4) El actor pulsa sobre el botón "Modificar receta" para enviar el formulario. 5) El sistema valida los datos. 6) El sistema redirige al listado de recetas.
Flujo alternativo	5) El Sistema comprueba que los datos no son válidos e informa al actor mediante mensajes.
Poscondiciones	Los nuevos datos reemplazan a los antiguos en la base de datos.

Nombre	**Modificar receta propia**
Fecha	15/05/2015
Descripción	Permite modificar la información de una receta.
Actores	Administrador, usuario.
Precondiciones	El usuario debe estar autenticado en el sistema. El usuario se encuentra en un listado de recetas.
Flujo normal	1) El actor pulsa sobre el botón "Modificar". 2) El sistema muestra un formulario con los datos de la receta actuales. 3) El actor modifica los datos que desea. 4) El actor pulsa sobre "Modificar receta" para enviar el formulario. 5) El sistema valida los datos. 6) El sistema redirige al listado de recetas.
Flujo alternativo	5) El Sistema comprueba que los datos no son válidos e informa al actor mediante mensajes.
Poscondiciones	Los nuevos datos reemplazan a los antiguos en la base de datos.

Nombre	**Modificar receta externa**
Fecha	15/05/2015
Descripción	Permite modificar la información de una receta.
Actores	Administrador.
Precondiciones	El usuario debe estar autenticado en el sistema y tener el rol de administrador. El usuario se encuentra en un listado de recetas.
Flujo normal	1) El actor pulsa sobre el botón "Modificar". 2) El sistema muestra un formulario con los datos de la receta actuales. 3) El actor modifica los datos que desea. 4) El actor pulsa sobre el botón "Modificar receta" para enviar el formulario. 5) El sistema valida los datos. 6) El sistema redirige al listado de recetas.
Flujo alternativo	5) Los datos no son válidos y se informa al actor mediante mensajes.
Poscondiciones	Los nuevos datos reemplazan a los antiguos en la base de datos.

Nombre	**Borrar receta**
Fecha	15/05/2015
Descripción	Permite eliminar una receta del sistema.
Actores	Administrador, usuario.
Precondiciones	El usuario debe estar autenticado en el sistema. El usuario se encuentra en un listado de recetas.
Flujo normal	1) El usuario pulsa sobre el botón "Borrar". 2) El sistema verifica que el usuario tiene permisos para borrar y elimina la receta escogida. 3) El sistema redibuja el listado sin la receta eliminada y muestra un mensaje de éxito.
Flujo alternativo	2) La receta no ha podido ser eliminada y el sistema muestra un mensaje de fallo.
Poscondiciones	La receta se elimina permanentemente de la base de datos.

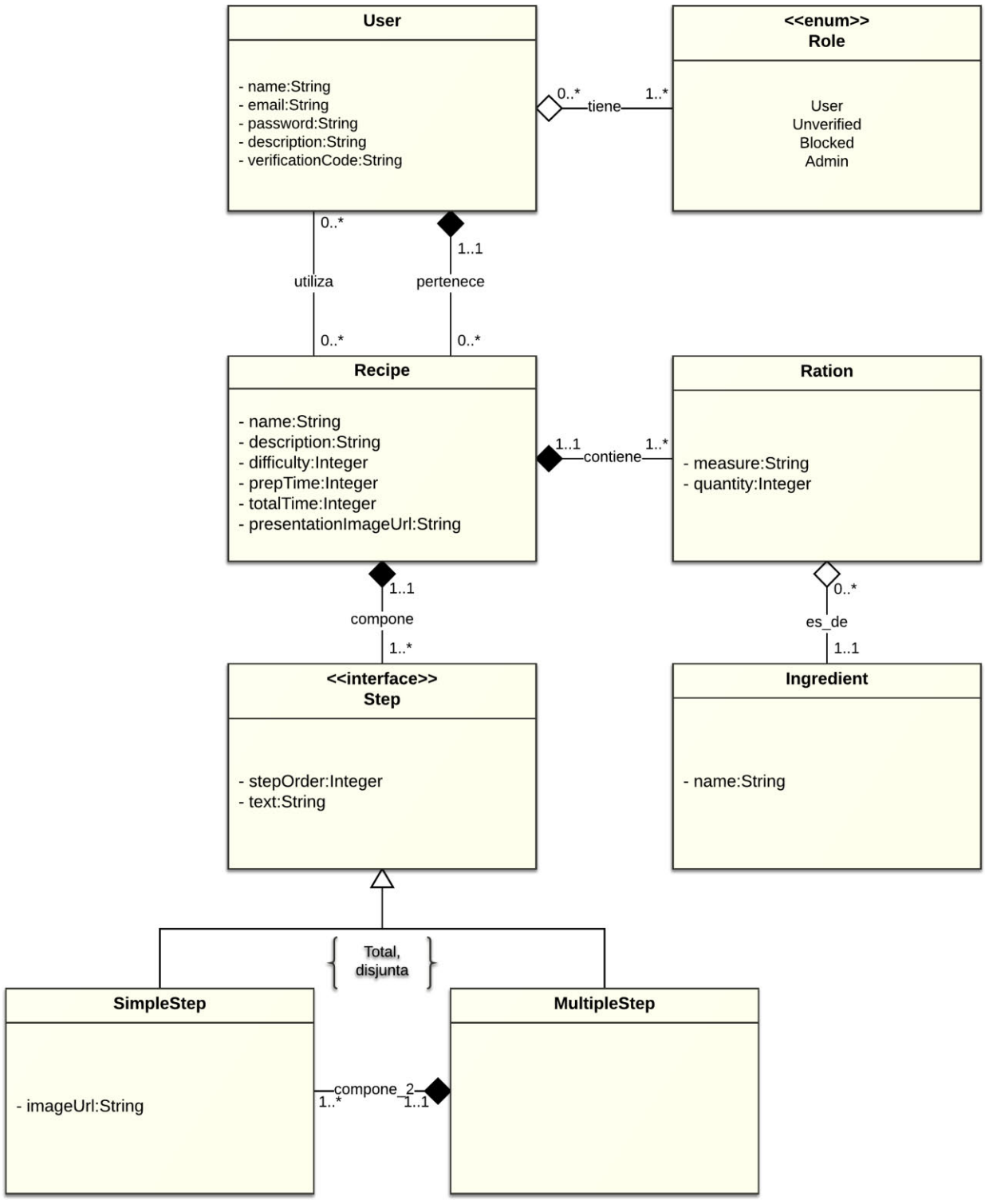
Nombre	**Borrar receta propia**
Fecha	15/05/2015
Descripción	Permite eliminar una receta del sistema.
Actores	Administrador, usuario.
Precondiciones	El usuario debe estar autenticado en el sistema. El usuario se encuentra en un listado de recetas.
Flujo normal	1) El usuario pulsa sobre el botón "Borrar". 2) El sistema verifica que el usuario tiene permisos para borrar y elimina la receta escogida. 3) El sistema redibuja el listado sin la receta eliminada y muestra un mensaje de éxito.
Flujo alternativo	2) La receta no ha podido ser eliminada y el sistema muestra un mensaje de fallo.
Poscondiciones	La receta se elimina permanentemente de la base de datos.

Nombre	**Borrar receta externa**
Fecha	15/05/2015
Descripción	Permite eliminar una receta del sistema.
Actores	Administrador.
Precondiciones	El usuario debe estar autenticado en el sistema y tener rol de administrador. El usuario se encuentra en un listado de recetas.
Flujo normal	1) El usuario pulsa sobre el botón "Borrar". 2) El sistema verifica que el usuario tiene permisos para borrar y elimina la receta escogida. 3) El sistema redibuja el listado sin la receta eliminada y muestra un mensaje de éxito.
Flujo alternativo	2) La receta no ha podido ser eliminada y el sistema muestra un mensaje de fallo.
Poscondiciones	La receta se elimina permanentemente de la base de datos.

2. ANÁLISIS

2.1. Diagrama de clases

De la especificación de requisitos se deduce el siguiente diagrama de clases:



Por un lado, un usuario tiene un nombre, una dirección de correo, una contraseña, una descripción, un código de verificación y una serie de roles:

- Usuario
- No verificado
- Bloqueado
- Administrador

Por otro lado, la receta tiene un nombre, una descripción, un número para indicar la dificultad, tiempos de preparación y totales en segundos, la dirección a una imagen principal, una serie de raciones y una serie de pasos.

Las raciones son una combinación de un ingrediente junto con la medida y la cantidad específica para la receta.

Los pasos son los que se deben seguir como instrucciones para hacer la receta. Los pasos, que por lo general llevan un orden y un texto explicativo, pueden ser de dos tipos:

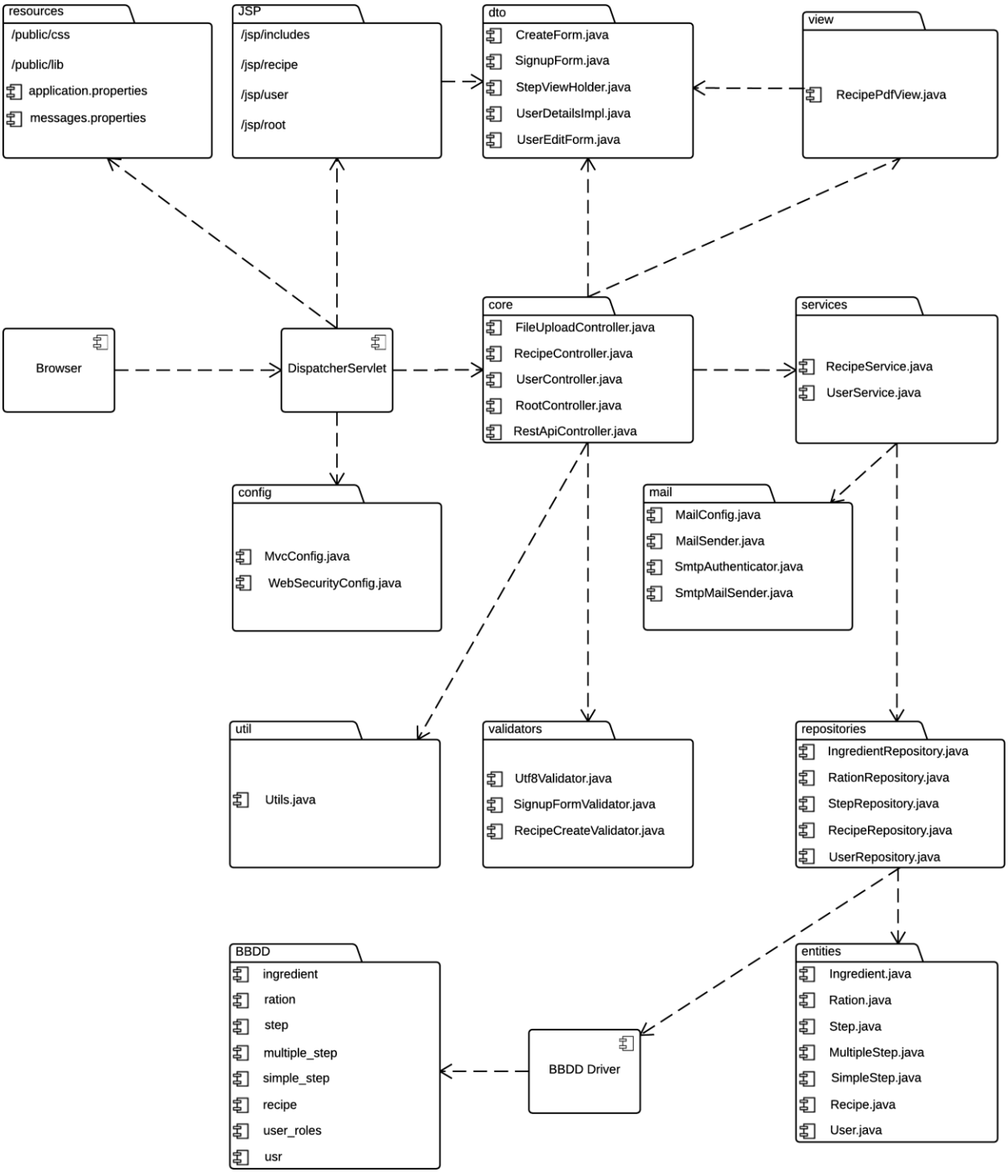
- Simples: Esto significa que sólo llevan el texto explicativo perteneciente a un paso y opcionalmente un enlace a una foto demostrativa del paso.
- Múltiples: Estos pasos agrupan una serie de pasos simples, para añadir un nivel de jerarquía en las instrucciones.

Sabiendo esto, se establecen las siguientes relaciones entre recetas y usuarios:

- Cada receta pertenece a sólo un usuario. Un usuario puede no tener recetas o tener las que desee.
- Los usuarios pueden utilizar todas las recetas que quieran, ya que todas las recetas se pueden utilizar por cualquier usuario.

3. DISEÑO

3.1. Diagrama de componentes



El diagrama de componentes mostrado contiene los siguientes paquetes:

- **resources**: contiene los ficheros de configuración, i18n, y los recursos necesarios para dar estilo y alguna funcionalidad a la página web.
- **JSP**: Contiene paquetes con archivos de tipo .jsp, utilizados para renderizar las distintas vistas.
- **config**: Contiene componentes que definen algunas de las direcciones que debe gestionar el DispatcherServlet. También contiene componentes de seguridad que indican cómo actuar frente a las distintas peticiones.
- **core**: Contiene los controladores. Son la base de la aplicación ya que el DispatcherServlet recurre a ellos con cada petición. Contiene un componente para gestionar la subida de archivos, un componente para gestionar usuarios, un componente para gestionar recetas, un componente para gestionar el resto de la web y un componente para gestionar una API Rest. El resto de direcciones no indicadas en config se indican en estos componentes mediante anotaciones.
- **view**: Contiene componentes cuya vista debe prepararse programáticamente, es decir, no son vistas renderizadas por el DispatcherServlet a partir de un nombre de fichero.
- **dto**: Contiene componentes que permiten seguir el patrón Data Transfer Object. Se utilizan para comunicar datos con las vistas.
- **util**: Contiene componentes con funciones de carácter estático, utilizados como herramienta general para todos los controladores.
- **validators**: Contiene componentes que permiten validar formularios de forma avanzada una vez recibida una respuesta que requiere dicha validación.
- **services**: Contiene componentes que se encargan de realizar acciones solicitadas por los controladores, procesando la información necesaria de los componentes del paquete "dto".
- **repositories**: Componentes que siguen el patrón Data Access Object (DAO).
- **entities**: Contiene componentes que representan las entidades representadas en el diagrama de clases.
- **BBDD**: Contiene las tablas generadas por Spring Framework necesarias para almacenar los datos de la aplicación.

4. HERRAMIENTAS DE DESARROLLO

Para la implementación de la aplicación web se han utilizado las siguientes herramientas:

- **Spring Tool Suite 3.6.3**. Entorno de desarrollo oficial de Spring. IDE basado en Eclipse, siendo ambos de código libre y gratuitos.
- **XAMPP 5.6.3**. Distribución gratuita de Apache que contiene las siguientes herramientas en la instalación:
  - **MySQL Community Server 5.6.21**. Gestor de base de datos.
  - **PhpMyAdmin 4.2.11**. Administrador de bases de datos MySQL.
  - **Apache Tomcat 7.0.56**. Implementación de un *servlet* de Java y de JSP.
- **Java Development Kit 1.8**
- **Maven.** Gestor de dependencias.

Se han utilizado las siguientes librerías de Java para implementar la aplicación:

- Spring Boot:
  - Web
  - Mail
  - Actuator
  - Data JPA
  - Security
  - Maven Plugin
- Tomcat Embed Jasper
- Javax JSTL
- Hibernate Validator
- MySQL Connector Java
- Spring Security Taglibs
- Apache Commons Lang 3
- Lowagie iText

Para las vistas web, se han utilizado los siguientes componentes:

- **Twitter Bootstrap 3.3.2**. Framework gratuito para HTML, CSS y JS que permite una implementación rápida de aplicaciones que se adaptan al tamaño de pantalla.
- **JQuery 1.11.2**. Librería Javascript que ofrece Ajax, gestión de eventos y manipulación de HTML de forma sencilla.

Para el despliegue de la aplicación se han utilizado las siguientes herramientas:

- **Pivotal Cloud Foundry**. PaaS que acepta aplicaciones Java.

  - **Pivotal Web Services**. Plataforma de monitorización y configuración de la aplicación subida.
  - **Cloud Foundry Command Line Interface 6.11.2**. Interfaz que permite desplegar aplicaciones.
  - **ClearDB**. Servicio interno que sirve como gestor de bases de datos.
- **MySQL Workbench 6.3**. Administrador de bases de datos MySQL. Permite conectar a bases de datos externas.

IV. CONCLUSIONES

Tras el estudio de Spring, Spring Boot y los módulos principales del framework, y la utilización de Spring en una aplicación web completa, puedo concluir destacando la simplicidad y facilidad que ofrece en cualquier implementación. Esto no significa que sea la mejor opción para cualquier tipo de aplicación web, sin embargo sí que ofrece suficiente como para poder considerar este framework en la mayoría de casos.

Entre las ventajas se puede apreciar lo siguiente:

- El contenedor de Spring y la utilización de inyección de dependencias permite obtener un código más desacoplado, permitiendo añadir y quitar módulos fácilmente.
- Los módulos que ofrece son completamente configurables y compatibles entre sí.
- Añadir módulos externos es posible gracias al contenedor de Spring.
- Spring MVC permite aplicar un modelo de diseño que permite desarrollar de forma ordenada y rápida.

Por otro lado, una desventaja es la limitación en los *bindings* con las vistas. A causa del funcionamiento entre servidores y clientes web, no es posible realizar cambios de forma dinámica sin refrescar la página. Cuando se desea hacer cambios dinámicos, es necesario añadir código en javascript para gestionar estos cambios antes de realizar un envío al servidor. Como solución, Spring podría ofrecer módulos que generen este tipo de código.

Para finalizar, incluir módulos como Spring REST permiten extender el proyecto más allá de una aplicación web. Cada vez es más común extender las aplicaciones a móviles y esto permite utilizar la misma lógica de servidor para persistir los datos y mostrarlos utilizando distintas tecnologías.

V. BIBLIOGRAFÍA Y REFERENCIAS

Spring	Febrero 2015
http://spring.io/	
Spring Guides	Febrero 2015
https://spring.io/guides/	
Overview (Spring Framework API)	Marzo 2015
http://docs.spring.io/spring/docs/4.1.x/javadoc-api/	
Spring Boot Reference Guide	Marzo 2015
http://docs.spring.io/spring-boot/docs/1.2.1.RELEASE/reference/html/	
Spring Framework Reference Documentation	Marzo 2015
http://docs.spring.io/spring/docs/4.1.x/spring-framework-reference/html/	
Spring Security Reference	Abril 2015
http://docs.spring.io/spring-security/site/docs/4.0.x/reference/html/	
Spring Data JPA - Reference Documentation	Abril 2015
http://docs.spring.io/spring-data/jpa/docs/1.8.0.RELEASE/reference/html/	
Spring Tutorial	Marzo 2015
http://www.tutorialspoint.com/spring/	
JSP Tutorial	Marzo 2015
http://www.tutorialspoint.com/jsp/	
Maven – Maven Documentation	Febrero 2015
http://maven.apache.org/guides/	
Natural Programmer | Using Spring framework effectively	Febrero 2015
http://naturalprogrammer.com/	