



UNIVERSIDAD
POLITÉCNICA
DE MADRID



Desarrollo con el Framework Spring de la aplicación statGambler

Proyecto fin de grado Ingeniería del Software

Autor: Fernando Barcala Rodríguez

Tutor: Juan Alberto de Frutos Velasco

Curso: 2018/2019

Resumen

El objetivo principal del proyecto es realizar un estudio del framework Spring y una aplicación utilizando esta herramienta, capaces de mostrar al lector el funcionamiento de este Framework desde un punto de vista teórico y práctico.

La primera parte del trabajo tendrá por objetivo analizar los distintos componentes que forman Spring, incluyendo algunas dependencias con Maven. También se incluyen las arquitecturas y patrones que implementa el Framework, la inyección de dependencias, cómo se realiza la configuración, tanto en XML como en Java y cómo se usan las anotaciones.

La segunda parte incluye la realización de una aplicación web que permita al usuario visualizar estudios estadísticos relativos a distintos juegos de azar, o crear los suyos propios. Para ello, se utilizarán las distintas partes que componen Spring, relacionadas con la creación de webs, manejo de bases de datos y Spring Core Container entre otros, incluyendo además distintas implementaciones de arquitecturas y patrones ofrecidos por Spring.

Summary

The main goal of this project is to write a study about Spring framework and to develop an application using this tool, in order to show the reader the capabilities of this framework from a theoretical and practical point of view.

The goal of the first part of the project is to analyze the different parts of Spring, including the Maven dependencies. The architectures and the patterns implemented by the framework are also included, as well as dependency injection, how to add configuration in XML and Java and how to use annotations.

The second part includes the development of a web application that let the user to visualize static studies relative to different gambling games, or to create them owns. In order to do so, the different parts that makes Spring will be used, related with web development, database management and Spring Core Container and so, including different implementations of architectures and patterns offered by Spring.

Resumen.....	1
Summary	2
Introducción	5
1 Estudio teórico.....	7
1.1 Historia y Motivación.....	7
1.2 Spring Tool Suite IDE.....	8
1.3 Introducción a la familia Spring.....	9
1.4 Arquitecturas y patrones implementados por Spring.....	20
1.5 Configuración de Spring	30
1.6 Spring Boot	39
1.7 Modulo Web.....	44
1.8 Modulo Datos	46
1.9 Modulo Test.....	49
2 Aplicación práctica	56
2.1 Descripción de la aplicación	56
2.2 Diseño de la aplicación	¡Error! Marcador no definido.
2.3 Desarrollo de la aplicación	¡Error! Marcador no definido.
2.4 Pruebas realizadas	75
2.5 Despliegue de la aplicación	77
3 Conclusiones.....	80
3.1 Problemas afrontados	80
3.2 Posibles mejoras.....	81
4 Bibliografía	83

Introducción

Un framework es un marco de trabajo software, universal y reutilizable, en el que se implementan un conjunto de herramientas, librerías y buenas prácticas reconocidas por su utilidad para resolver un problema determinado. Así, un framework, gracias a las implementaciones que añade, puede ayudar a un desarrollador a conseguir un código más sencillo y eficiente, facilitando por tanto su escritura y mantenimiento.

Spring es uno de los frameworks más populares para el desarrollo Java. Es utilizado por miles de programadores y empresas en todo el mundo.

El framework Spring comenzó como un software sencillo, cuyo objetivo era facilitar el desarrollo de aplicaciones web con Java. Poco a poco se le fueron añadiendo funcionalidades hasta llegar a lo que es hoy, conformando una familia de frameworks que facilitan el desarrollo de distintas partes del software.

Spring es bastante aceptado por muchos desarrolladores y en una encuesta reciente de Stack Overflow, mostrada en la siguiente imagen, Spring consiguió formar parte del top de los frameworks más queridos.

Most Loved, Dreaded, and Wanted Frameworks, Libraries, and Tools

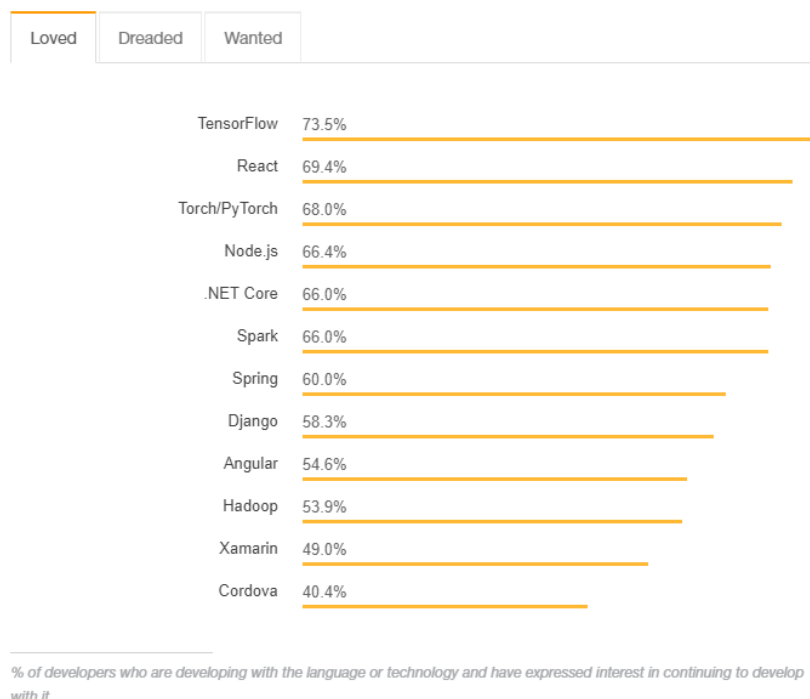


Figura 1.1.1:top herramientas [14]

Mediante el uso de Spring, se puede mejorar la capacidad de prueba y mantenibilidad del código, además, el framework ayuda a desarrollar software con una mayor adaptabilidad, con mayor capacidad para el cambio y el aumento de responsabilidades.

Ayuda a desacoplar el código de tal forma que se pueden añadir herramientas como cacheo sin necesidad de reescribir el código base. Además, el framework ayuda a reducir la complejidad del código, consiguiendo facilitar así el trabajo a los desarrolladores.

De esta forma, desde un punto de vista de negocio, ayuda a obtener el código más rápidamente, mejorando beneficios.

1.1 Objetivos

Para mostrar y comprobar la capacidad de Spring a la hora de desarrollar una aplicación, en este proyecto, además de ofrecer un estudio teórico del framework, se desarrollará una aplicación sencilla para poder añadir una parte práctica y comprobar de primera mano el funcionamiento del framework.

Esta parte práctica consistirá en una aplicación web sencilla que ofrezca al usuario una información estadística y personalizada de distintos juegos de azar.

En esta aplicación, se tratará la creación, lectura, actualización y borrado de cada uno de los resultados de los juegos de azar, además de realizar el tratamiento de usuarios y su consiguiente tratamiento de seguridad.

Para realizar esta aplicación web, se utilizarán elementos de Spring MVC para poder utilizar vistas, controladores y modelos que permitan un desarrollo de la aplicación sencillo y eficiente.

Además, se utilizarán otros componentes como Spring Security para el tratamiento de usuarios, Hibernate para el tratamiento de datos JUnit para la realización de distintos test o flyway para las migraciones de la base de datos.

De esta forma, se obtendrá un estudio del framework tanto teórico como práctico que permita dar una visión global del funcionamiento de la familia Spring.

2 Estudio teórico

2.1 Historia y Motivación

Spring Framework fue el comienzo de la familia Spring. En el momento de su creación, alrededor de 2003, el desarrollo de aplicaciones empresariales con Java era un proceso estricto y complicado. Entonces, Rod Johnson, el creador de Spring, se propuso solucionar este problema creando un framework que simplificase el proceso de creación de las aplicaciones.

El framework Spring llegó a tener éxito en este aspecto y, con el tiempo, creció y evolucionó hasta llegar a lo que es hoy, bastante diferente de como comenzó. Además de simplificar el desarrollo de aplicaciones Java, se convirtió en la base para el resto de los proyectos de la familia Spring. De este modo, las aplicaciones creadas con Spring podrían utilizar estos proyectos, además del mismo framework, para simplificar aún más su desarrollo.

El objetivo de este framework era ayudar a que desarrollos como el de webs y el acceso a datos fuesen más fáciles de implementar para los programadores. Además, también buscaba reducir el código repetitivo, es decir, código que es necesario por la lógica de la aplicación, pero que tiende a desordenar y desviar la atención de la lógica principal.

El éxito del framework llevó a sus desarrolladores a continuar con la creación de varios proyectos Spring, contruidos sobre el framework y hechos a medida para distintos dominios y necesidades.

El Framework Spring y sus proyectos, continuaron evolucionando y mejorando, poco a poco, llegando a la creación de un proyecto con un gran impacto llamado Spring Boot. Spring Boot proporcionó una manera nueva y mucho más rápida de desarrollar aplicaciones basadas en Spring.

Hasta la llegada de Spring Boot, la creación de una aplicación basada en Spring siempre implicaba muchas decisiones, configuraciones y un modelo de despliegue incómodo. Spring Boot consiguió eliminar estos problemas con una vista de la construcción de aplicaciones basadas en Spring mucho más guiada y cerrada, incluyendo valores por defecto adecuados para la elección de librerías y añadiendo componentes inteligentes para la detección y configuración automática de otras configuraciones comunes.

Por último, aparece el proyecto Spring Cloud. Spring Cloud fue construido sobre Spring Boot y simplificó el desarrollo de aplicaciones que hacen uso de arquitecturas distribuidas, como la de microservicios. Estos tipos de arquitectura, a menudo, tienen muchos patrones comunes implementados, como descubrimiento de servicios o configuración distribuida, y Spring Cloud ayuda a los desarrolladores a construir aplicaciones que utilizan esos patrones.

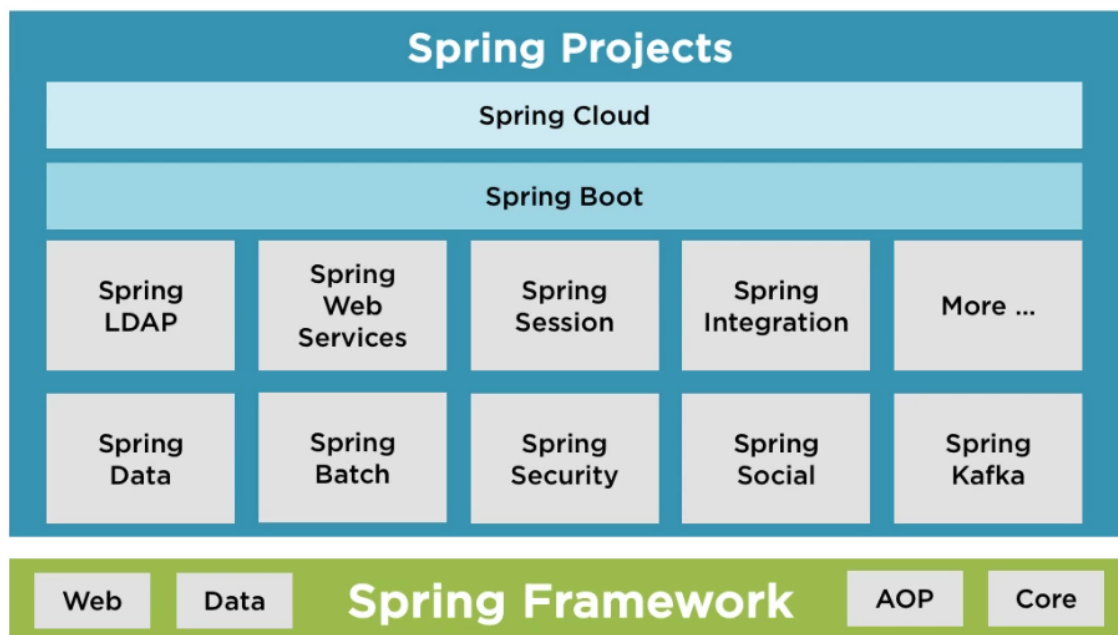


Figura 2.1.1: familia Spring

2.2 Spring Tool Suite IDE

El IDE Spring Tool Suite, está basado en Eclipse y trae en su descarga los módulos necesarios de Spring y Maven para que el usuario pueda trabajar sin problemas. Por este motivo, es el IDE elegido para la realización del proyecto, ya que facilita la gestión de los paquetes de Spring y, además, el autor ya está familiarizado con el entorno Eclipse, ya que ha sido tratado durante la mayor parte del grado.

Spring Tool Suite ha sido diseñado para desarrollar aplicaciones utilizando Spring Framework y Spring Boot. Además de Spring Tool Suite de Eclipse, siendo este un entorno de desarrollo completamente integrado, las herramientas Spring Tools incluyen integración con Visual Studio Code y Atom, menos pesados en el desarrollo. De esta forma, el usuario puede utilizar el IDE que prefiera y añadir las herramientas Spring. [9]

Esta nueva generación de herramientas Spring ha sido en su mayoría construida desde cero, incluyendo nuevas tecnologías y arquitecturas de desarrollo. Se ejecuta en procesos separados, está construido teniendo en mente la eficiencia y conoce las novedades de Spring.

Además, Spring cuenta en su página web con Spring Initializr, que permite al desarrollador configurar y comenzar con un proyecto en poco tiempo.

Spring Tool Suite, trata de entender el código desarrollado basado en Spring y permite al programador tener una vista general y navegar entre las piezas de las aplicaciones. De esta forma, se facilita la búsqueda y navegación de elementos Spring.

Las herramientas nuevas de Spring incluyen sugerencias de código inteligentes para los elementos Spring dentro de la aplicación.

Spring Tools 4 ahora solventa la brecha entre el código fuente y la ejecución de aplicaciones Spring Boot. Al sacar partido de los actuadores de Spring Boot, Spring ha enriquecido el código con información detallada de la ejecución de la aplicación (conexión de beans, reportes, configuración y detalles, entre otros).

Aunque, preparar y dejar listo un proyecto de Spring con Spring Tool Suite es más sencillo que utilizar el IDE simple, que no cuente con la integración de los plugins de Spring, siempre habrá que tener en cuenta qué plugins se añaden y si son necesarios o no, ya que, en algunos casos, podría ser contraproducente y acabar por ralentizar el entorno.

2.3 Introducción a la familia Spring

El framework Spring utiliza una arquitectura modular, lo que significa que está dividido en varios componentes distintos que se pueden conectar entre sí. Cada uno de esos componentes es responsable de proveer una funcionalidad específica, y todos juntos, forman el framework Spring.

El framework Spring puede ser dividido en seis áreas clave distintas. Estas áreas son Core, Web, AOP (Programación orientada a aspectos), Acceso a datos, integración y testing.

Un módulo es una parte específica del framework que encierra una funcionalidad por completo y puede estar compuesto a su vez por varios módulos de menor tamaño, para ayudar a organizar el software. Sirven para aislar los módulos lo máximo posible y organizar el framework, respetando los principios SOLID y consiguiendo que cada módulo esté desacoplado y dependa lo mínimo posible de los demás. En la siguiente imagen, se muestran los módulos que forman Spring. [13]

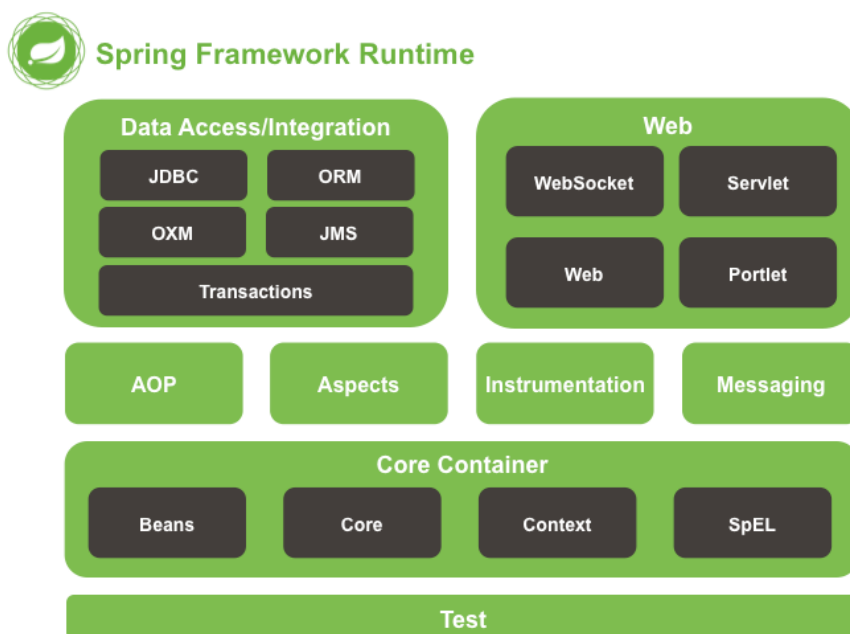


Figura 2.3.1: Módulos de Spring [2]

Spring se compone de varios módulos, que tratan de dividir el framework en varias partes diferenciadas, para poder organizarlo y ayudar al desarrollador durante su uso. Cada módulo se encarga de implementar una parte concreta del framework.

A continuación, se listan los módulos principales que componen Spring, explicando, para cada uno, la función que realiza:

2.3.1 Spring Core

El módulo central de Spring, o Spring Core, es una de las partes más importantes, si no la más importante del framework Spring. Sirve de cimientos sobre los que se construyen el resto de los módulos.

Spring Core es responsable de proveer distintas funcionalidades, como soporte a la internacionalización, a la validación, al enlace de datos, a la conversión de tipos y muchas otras.

Pero, una de las partes más importantes de Spring Core es la **inyección de dependencias**. Cuando se desarrolla software mediante programación orientada a objetos, se crean objetos que representan o modelan elementos concretos y estos objetos, normalmente, dependen unos de otros.

La inyección de dependencias trata con el modo en que los objetos obtienen sus dependencias. Principalmente, hay dos opciones para completar las dependencias de un objeto. La primera es que el objeto complete sus propias dependencias y la segunda es que el objeto pueda declarar cuáles son sus dependencias y confiar en que otro elemento las complete.

La primera elección puede parecer sencilla y, quizá, la mejor opción, pero tiene limitaciones. Si un objeto es el responsable de completar sus propias dependencias, entonces, el objeto y sus dependencias estarán fuertemente ligados entre ellos.

La segunda elección, es mucho más flexible. El objeto y sus dependencias siguen ligados, pero son considerados ligeramente ligados, porque se puede cambiar uno sin tener que cambiar los demás.

Esta segunda opción es lo que se llama inyección de dependencias, y Spring Core es considerado como un contenedor de inyección de dependencias. Se puede utilizar para declarar objetos y sus dependencias y Spring Core creará, administrará y conectará los objetos y sus dependencias.

El resultado directo de la inyección de dependencias es que se le quita carga de trabajo al desarrollador, ya que tendrá menos tareas que manejar.

Spring Core, cuenta con Spring Core Container para poder implementar estas funcionalidades, es necesario para realizar la implementación de la inversión de control.

Aunque no es necesario utilizar este Container, sí que es recomendable, ya que facilita la gestión del ciclo de vida de los objetos manejados durante el flujo del programa, incluyendo su creación, enlace, configuración y destrucción. En Spring, estos objetos gestionados son los llamados **Beans**.

Los Beans de Spring son clases simples, formadas por getters y setters, configurados mediante Spring. Son POJOs (Plain Old Java Object) que han sido instanciados a través de Spring y están conectados para ser utilizados en el framework a través del container.

2.3.2 Spring Web

El módulo web de Spring, o Spring Web, es un framework para el manejo de peticiones web. Las peticiones web se pueden manejar de dos maneras distintas, ya sea vía Spring MVC o Spring Webflux. Antes de ver la diferencia entre estos dos métodos, es necesario entender qué es y para qué se utiliza un servlet.

El lenguaje Java introdujo un soporte muy básico para interactuar con la web, mediante un framework llamado Servlet API. Llamado así por el componente clave, el servlet. Un servlet es, básicamente, un objeto que recibe una petición y genera una respuesta basada en esa petición.

Con Spring MVC, la petición llega al Servlet web y pasa al Servlet API que procesan la información que reciben. A continuación, pasa por Spring MVC y la lógica de la aplicación puede entonces hacer uso de Spring Web MVC y generar un resultado.

Por último, se devuelve el control a Spring Web MVC, que devolverá el control al Servlet API y finalmente, generar una respuesta, como muestra la siguiente figura.

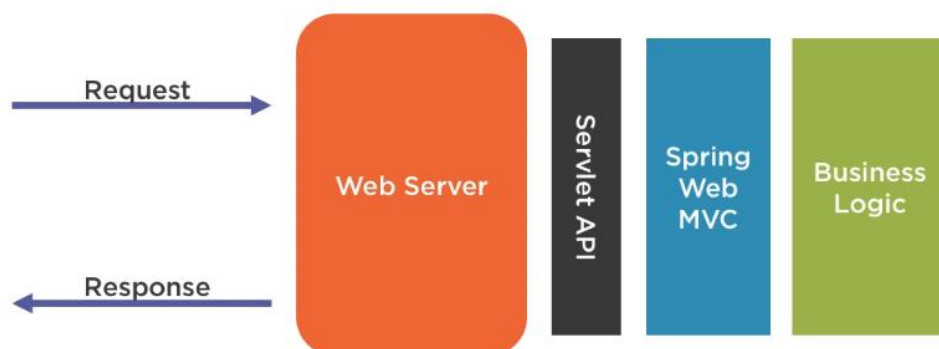


Figura 2.3.2: Petición Web con Spring Web MVC

La ventaja del framework Spring Web MVC es que proporciona al desarrollador una API de más alto nivel para interactuar con ella. Esto resulta en una utilización más sencilla y más productividad. Además, este alto nivel, hace que sea más sencillo desarrollar código que siga principios de diseño apropiados.

Spring Webflux es otra forma de manejar peticiones web soportado por el framework Spring. Este método de manejo de peticiones web se basa en la programación reactiva.

La programación reactiva es un paradigma de programación declarativa preocupado por los flujos de datos y la propagación del cambio. En otras palabras, es una forma de programar que se centra en flujos de datos y cómo estos cambian.

El concepto clave detrás de la programación reactiva es que se reacciona al cambio, en lugar de esperarle.

Spring Webflux también tiene como objetivo manejar peticiones, pero el modo en que lo hace es distinto. Concretando, las peticiones se ejecutan de forma asíncrona y no se bloquean ni esperan, lo que resulta en un mejor aprovechamiento de recursos.

Para conseguir esto, en lugar de esperar, el código pide ser notificado cuando una operación se completa y continúa ejecutando otras operaciones. Una vez que es notificado, el código termina ejecutando los pasos restantes.

2.3.3 Spring AOP

AOP son las siglas de Programación orientada a aspectos (Aspect-Oriented Programming). Es un paradigma de la programación cuyo objetivo es incrementar la modularidad, permitiendo la separación de objetivos comunes.

Mediante esta forma de desarrollo, se pretende otorgar al código una mayor organización, al separar código redundante, que afecta a muchas otras partes, del código de la lógica de negocio.

Un ejemplo de objetivo común es la seguridad, ya que no se puede contener en un área concreta de la aplicación, en su lugar, aparece en múltiples áreas de la aplicación.

Sin AOP, resolver problemas que son comunes en distintas partes de la aplicación, resulta en código dividido y duplicado en muchas partes de una aplicación.

Spring AOP es una implementación de AOP en el Framework Spring y principalmente tiene dos usos.

El primero es implementar ciertas características, mediante el uso de AOP, en el mismo framework Spring, y la segunda es ser una herramienta valiosa para los desarrolladores para resolver problemas que aparecen en distintas capas de una aplicación.

2.3.4 Spring Data Access Support

Ya sea para almacenar, recuperar información o ambas, casi cualquier aplicación interactúa con datos. Dada la importancia de los datos, cuanto más fácil sea manejarlos, mejor. Ese es el trabajo del módulo framework Spring Data Access.

Para comenzar, se presenta un ejemplo. En la siguiente figura, se presenta un código utilizado para recuperar datos utilizando JDBC. Como se puede observar, se requiere bastante código para recuperar los datos y sólo hay una pequeña parte que es importante para la lógica, que es el SELECT donde se indica la operación. El resto del código sólo se necesita para usar el API.

```
try {
    Statement stmt =
        conn.createStatement();
    try {
        ResultSet rs = stmt.executeQuery(
            "SELECT COUNT(*) FROM foo");
        try {
            rs.next();
            int cnt = rs.getInt(1);
        } finally {
            rs.close();
        }
    } finally {
        stmt.close();
    }
} catch (SQLException e) {
    // handle error
} finally {
    try {
        conn.close();
    } catch (SQLException e) {
        // handle error
    }
}
```

Figura 2.3.3:datos con JDBC

Este código del API es repetitivo, lo que hace que sea muy tedioso escribirlo cada vez que se necesita acceder a la base de datos, además de aumentar duplicidades y la dificultad de mantener el código.

En cambio, utilizando el framework Spring Data Access el mismo código quedaría tal como se muestra en la siguiente figura.

```
int cnt = new JdbcTemplate(ds)
    .queryForInt("SELECT COUNT(*) FROM foo");
```

Figura 2.3.4:Datos con Spring

Como se puede apreciar, el código es muchísimo más pequeño, además, se ha ganado claridad y facilidad de lectura de este. Ya no hay que abrirse paso a través del código repetitivo sólo para entender su objetivo.

El soporte de Spring Data Access hace muy fácil el uso de transacciones de bases de datos. Estas transacciones son unidades de trabajo que ocurren como un bloque, es decir, o se ejecutan por completo o no se ejecutan.

Otra característica interesante dada por el módulo de acceso a datos es algo llamado traducción de excepciones. Cada base de datos tiene errores distintos y distintas excepciones para el mismo tipo de error. El módulo de acceso a datos de Spring Framework lo que hace es tomar estos errores específicos de cada base de datos y traducirlos a un conjunto de excepciones conocido.

Por último, el soporte Data Access, con el Framework Spring ayuda a hacer que las pruebas sean más sencillas. Como el framework Spring está manejando la configuración y prepara el modo en el que una aplicación accede a sus datos, es sencillo cambiar esa configuración para pruebas.

Por ejemplo, durante las pruebas es común utilizar una base de datos embebida, en lugar de la real para evitar manipular datos reales. Spring Data Access ayuda a preparar esa base de datos embebida y asegura que el código está configurado para esos test.

2.3.5 Spring Data

Tal como se indica en su página web, el objetivo del proyecto Spring Data es proporcionar un modelo de programación basado en Spring, familiar y consistente para el acceso a datos mientras se mantienen los tratos especiales del almacenamiento de datos subyacente.

Esto quiere decir que, sin importar el tipo de datos que se almacenen, Spring Data utiliza algunos patrones comunes a través de todo el almacenamiento de datos, con el objetivo añadido de asegurar que cualquiera de las características específicas de un almacenamiento de datos concreto, siguen disponibles.

La diferencia entre Spring Data y el soporte del acceso a datos del Framework Spring es que Spring Data extiende las capacidades de acceso a datos proporcionadas por el Framework Spring.

Mientras que Spring Framework está centrado en un solo tipo de base de datos, bases de datos relacionales, Spring Data agrega formas nuevas de interactuar con bases de datos relacionales, así como soporte a muchos otros tipos de bases de datos, como Hadoop con Big Data.

Spring Data es, en realidad, un proyecto general, esto quiere decir que no es un proyecto en sí mismo, sino un conjunto de subproyectos, en este caso, para el soporte al almacenamiento en base de datos.

En su página web, se indican todos los tipos de almacenamiento de datos disponibles. Spring Data cuenta con una gran cantidad de submódulos con muchos tipos de almacenamiento de datos cada uno. Cuenta con módulos principales, soportados bajo Spring Data, y también cuenta con módulos de contribución de la comunidad, desarrollados y mantenidos por otros, fuera del equipo de Spring Data.

A modo de ejemplo, se muestra el código de la siguiente figura, en el que se muestra cómo crear un objeto que mantiene algunos datos. Spring Data hará sencillo guardar, borrar y encontrar cualquiera de esos datos con métodos ya contruidos y sin esfuerzo para el desarrollador.

```
@Entity
public class Employee {

    private @Id @GeneratedValue Long id;
    private String firstName, lastName, description;

    private Employee() {}

    public Employee(String firstName, String lastName, String description) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.description = description;
    }
}
```

Figura 2.3.5:Entidad con Spring

Spring Data, al igual que otras tecnologías, como JPA, hace el trabajo duro de obtener los datos almacenados en el objeto y traducirlos, mapearlos o convertirlos a algún tipo que pueda ser almacenado en la base de datos objetivo.

2.3.6 Spring Integration

Las aplicaciones no trabajan aisladas, a menudo, necesitan compartir datos o trabajar con otras aplicaciones. La integración trata de hacer que diferentes sistemas y aplicaciones trabajen juntos.

El problema a resolver es cómo una aplicación se comunica con otra. Y este problema tiene varios frentes. Habrá que tener en cuenta cómo exponer operaciones a otros sistemas y cómo llamar o invocar operaciones en otros sistemas.

Hay muchas maneras distintas de exponer operaciones a otros sistemas, como RMI (Remote Method Invocation) o mediante sistemas de mensajería. El framework Spring soporta ambos métodos. Pero una forma común de exponer operaciones son los servicios web.

Los servicios web son operaciones expuestas y accesibles a través de la web. Spring Framework hace que resulte más fácil exponer e invocar estos servicios web. Como ejemplo de un servicio web se presenta el código de la siguiente figura, usado para recuperar información de una cuenta.

```
@RestController
public class AccountController {

    @GetMapping("/account/{id}")
    public Account
        find(@PathVariable int id) {
        // look up account by id
    }
}
```

Figura 2.3.6:Controlador REST con Spring

Este servicio web utiliza tres anotaciones. **@RestController**, que indica que el código va a exponer acciones utilizando REST, que es utilizado para implementar servicios web, **@GetMapping** indica la operación y la ruta que se utiliza para invocar esta operación y **@PathVariable** asocia un valor al id de la ruta definida por la anotación anterior. De este modo, se puede exponer un servicio que se puede utilizar para buscar una cuenta mediante su id.

Por otro lado, para invocar una operación, el framework Spring proporciona soporte para invocar el servicio REST de forma programática, utilizando un **RestTemplate**. Esta plantilla, abstrae los detalles tediosos de la interacción con un servicio web como abrir conexión con el servicio, mandar el comando y manejar la respuesta.

2.3.7 Spring Testing

Las pruebas son una parte muy importante en el desarrollo software, por ello, Spring también cuenta con un módulo enfocado en esta área.

Hay muchos niveles en los que hacer pruebas a una aplicación. Spring Framework se enfoca en dos, test unitarios y test de integración.

Los test unitarios son un proceso del desarrollo software en el que las partes sobre las que se realizan pruebas más pequeñas de una aplicación, llamadas unidades, son probadas de forma individual e independiente.

El soporte explícito para test unitarios en Spring Framework es, en realidad mínimo. El soporte del framework Spring para realizar pruebas, mayormente, viene como un efecto secundario de utilizar la inyección de dependencias.

En los test unitarios la idea es probar la unidad más pequeña de código posible, cuando el código que se está probando no tiene dependencias, el test resulta sencillo de realizar. Pero si las tiene, este se complica.

Es importante ser capaz de controlar cómo esas dependencias se comportan para restringir el test a una sola unidad y no ampliarlo a sus dependencias.

Si el código se escribe con inyección de dependencias en mente, las pruebas se vuelven mucho más fáciles, ya que el desarrollador debe declarar qué dependencias tiene el código y este espera recibir esas dependencias. El código no se preocupa de dónde vienen, sólo de completar sus dependencias.

Así que, durante las pruebas, el código de las dependencias puede ser sustituido por código que se comporta de un cierto modo. Esta acción de reemplazar piezas del código con código controlador es conocido como Mock.

En las pruebas unitarias, no se utiliza la inyección de dependencias para inyectarlas en los test unitarios. Sin embargo, el modo en el que el código se configura para la inyección de dependencias, principalmente porque el código debe declarar sus dependencias, ayuda a hacer que las pruebas sean más sencillas.

En caso de que el código no se escriba con la inyección de dependencias, sería mucho más difícil realizar las pruebas ya que no se podría controlar del mismo modo lo que los objetos de los que depende devuelven.

Como parte del soporte de los test unitarios, Spring Framework incluye unos mocks preprogramados que se pueden utilizar durante las pruebas. Con lo que se consigue que realizar test sea más fácil y rápido para el equipo de desarrollo porque les evita tenerlos que escribir por ellos mismos.

Los test de integración, por otra parte, son la fase en los que módulos individuales de software se combinan y se prueban en conjunto. Se debe realizar tras los test unitarios.

Los test de integración se centran en cómo las distintas partes del software se integran entre sí. Permite ejecutar piezas completas de una aplicación y probar cómo trabajan juntas, no sólo de forma aislada como los test unitarios.

El Framework Spring juega un papel importante en los test de integración, ya que puede ser utilizado para unir varias partes de la aplicación para las pruebas. Proporciona soporte a los test de integración comunes, como configurar y probar datos de acceso o probar partes completas de una aplicación web para asegurar su funcionalidad.

Otra característica proporcionada por el Framework Spring es la capacidad de limpieza tras una prueba. Al realizar un test de integración, es común cambiar el estado de la aplicación tras ejecutarlo, por ejemplo, se podrían introducir datos al realizar un test de acceso de datos. Si estos cambios no se limpian, podrían afectar a la ejecución de otras pruebas.

2.3.8 Spring Security

Spring Security es un framework para trabajar la seguridad de aplicaciones basadas en Spring. Maneja la autenticación y autorización y cubre todas las formas estándar de manejo de seguridad, además de cualquier necesidad personalizada, si fuese necesario.

Spring Security no es sólo para autenticación y autorización, cuenta con características adicionales que ayudan a proteger una aplicación de ataques comunes como cross-site scripting o click jacking.

A modo de ejemplo, se muestra el código de la figura siguiente. En él, primero se añade la anotación de clase **@EnableWebSecurity**, como su nombre sugiere, sirve para configurar la aplicación para que sea capaz de manejar peticiones web seguras.

En el método `configure`, se indican las rutas permitidas y cuáles requieren autorización y, además, está configurando la localización de la página de acceso de la aplicación.

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/css/**", "/index").permitAll()           ❶
                .antMatchers("/user/**").hasRole("USER")              ❷
                .and()
            .formLogin()
                .loginPage("/login").failureUrl("/login-error");       ❸
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .inMemoryAuthentication()
                .withUser("user").password("password").roles("USER");
    }
}
```

Figura 2.3.7:Ejemplo con Spring Security

Como se puede ver, el código se deja leer de manera cómoda. Se puede ver que cualquier petición para la ruta de `/CSS` o `/index` siempre se permite, pero las

peticiones para /user requieren que el usuario esté autenticado y autorizado con el rol de usuario.

Este código es común y muestra lo sencillo que es permitir algunas rutas de una aplicación web para ser siempre permitidas o a otras que requieran seguridad.

2.3.9 Spring Boot

Spring Boot hace que aprender Spring sea rápido y fácil, los desarrolladores pueden comenzar a trabajar con el framework sin llegar a entender en profundidad cómo funciona el framework más allá de lo que se puede ver en un primer plano. Una vez que hayan adquirido experiencia, siempre podrán retomar estos detalles y llegar a comprenderlos.

Spring Boot, forma parte de la familia Spring, y sirve para crear aplicaciones de cualquier tipo, incluyendo web entre otros. Spring Boot tiene unas características clave que lo hacen único y fácil de usar. A continuación, se listan algunas de estas características.

Una de las características más importantes de Spring Boot es la autoconfiguración. Spring Boot es capaz de configurar y preparar una aplicación automáticamente, basándose en algunas de sus características e indicaciones dadas por el desarrollador.

Siguiendo la documentación, la autoconfiguración intenta configurar, de forma automática, una aplicación Spring basándose en las dependencias que se han añadido. Y esto se consigue gracias a la inteligencia de Spring y su consciencia del contexto. Gracias a esto, Spring es capaz de ahorrar tiempo al desarrollador, ya que hace una mejor suposición de las dependencias necesarias en cada momento.

Por ejemplo, si Spring Boot detecta que una aplicación tiene alguna dependencia que está relacionada con una base de datos, puede hacer una suposición razonable de que probablemente debería configurar ciertos elementos para acceder a esa base de datos.

Es más, si esa dependencia es para una base de datos muy específica como Oracle o MySQL, Spring Boot podrá realizar una suposición aún mejor y dejar listas características que podrían ser específicas de esta base de datos en concreto.

Para un desarrollador, preparar la autoconfiguración es sencillo. Sólo tendrá que añadir anotaciones a su código de la aplicación de Spring Boot. Spring Boot trata de ser lo menos invasivo posible, por tanto, son tan fáciles de añadir como de retirar.

Otra característica de Spring Boot es que es autónomo. No es necesario desplegar la aplicación en un servidor web o un entorno especial, simplemente se puede ejecutar la aplicación con un solo comando, tal como se haría con cualquier otra aplicación.

Gracias a esta característica, las aplicaciones de Spring Boot simplemente se ejecutan. Sólo se deben empaquetar y ejecutarlas con un simple comando: "java -jar mi-aplicación.jar". Spring Boot se encarga del resto, arrancando un servidor web

embebido con la aplicación, lo configura con valores por defecto, y ejecuta la aplicación.

Por último, Spring Boot es sugerido, es decir, que Spring Boot tiene una forma ya establecida de realizar sus acciones por defecto. Aunque pueda parecer, a priori, una característica poco interesante, es, en realidad, una buena característica a tener en cuenta.

Al construir aplicaciones Java, se tienen que hacer demasiadas decisiones, desde el tipo de framework web, las librerías, la configuración o las herramientas, entre otros. Para un desarrollador nuevo en el entorno, tantas elecciones pueden ser agobiantes. Y normalmente se siguen las mismas elecciones basándose en estándares y popularidad.

Spring Boot ayuda a quitar esta carga para que el desarrollador pueda obtener una aplicación ejecutable lo más rápido posible. Esto significa que pueden aprovechar más tiempo centrándose en escribir código que resuelve las necesidades reales de la aplicación.

Además, aunque Spring Boot es sugerido, es sencillo sobrescribir cualquiera de las elecciones ya determinadas.

2.3.10 Spring Cloud

Spring Cloud es un proyecto construido sobre Spring Boot, y hace más fácil construir sistemas distribuidos, siendo un sistema distribuido un conjunto de aplicaciones relacionadas a través de una red.

Un ejemplo común de un sistema distribuido es la arquitectura de microservicios. Cada microservicio es una aplicación pequeña, con un alcance bien definido a un solo propósito o dominio. La arquitectura de microservicios es un conjunto de microservicios que se comunican a través de la red.

Al implementar una arquitectura como la de microservicios, se acaban teniendo problemas conocidos, con soluciones también conocidas. Por ejemplo, cómo un microservicio encuentra a otro, es decir, descubrimiento de servicios (service Discovery).

Es para este tipo de problemas Spring Cloud ayuda a los desarrolladores a implementar la solución. En el caso del descubrimiento de servicios, Spring Cloud añade una anotación mediante la que se puede realizar esta operación sin esfuerzo, como se muestra en la siguiente imagen.

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

Figura 2.3.8: Ejemplo de uso de anotaciones con Spring Cloud

Utilizando Spring Cloud y añadiendo esta anotación a la aplicación, el servicio podrá encontrar otros servicios y, además, podrá ser descubierto por otros servicios.

El descubrimiento de servicios es sólo un ejemplo de lo que ofrece Spring Cloud. Al igual que Spring Data, Spring Cloud no es un proyecto en sí mismo, sino un proyecto general que contiene varios subproyectos.

Estos subproyectos ayudan con varios problemas como la configuración y los fallos de un sistema distribuido.

2.4 Arquitecturas y patrones implementados por Spring

Spring Implementa muchos patrones, lo que ayuda al desarrollador a evitar errores comunes y dificultades al tratar de implementarlos.

2.4.1 AOP (Programación orientada a aspectos)

AOP es uno de los principales fundamentos de Spring y, en el framework, es utilizado a través de Java EE. Es importante porque se usa para implementar las características de empresa típicas, que hacen que Spring Framework sea tan útil, estas características incluyen operaciones transaccionales o seguridad, entre otros.

AOP permite al desarrollador decidir dónde se deberían añadir los aspectos al código, lo que ofrece middleware configurable. Además, se pueden añadir características propias y permite simplificar el código desarrollado y eliminar mucho código básico.

La idea de la programación orientada a aspectos es poner cada una de las partes, o aspectos, de ese código en un aspecto y borrarlo del método.

De esta forma, usando AOP, se podrá retirar el código de transacción, de trazado y de excepciones, que se escribirán como aspectos y sólo la lógica de negocio será la que quede, retirando todo el código simple y mecánico.

2.4.1.1 Cómo funciona AOP

En un sistema grande, con muchos paquetes y clases, normalmente, en cada bloque, hay zonas de código comunes, que son implementados en muchas de las clases y métodos del proyecto, como se muestra en la siguiente figura.[7]

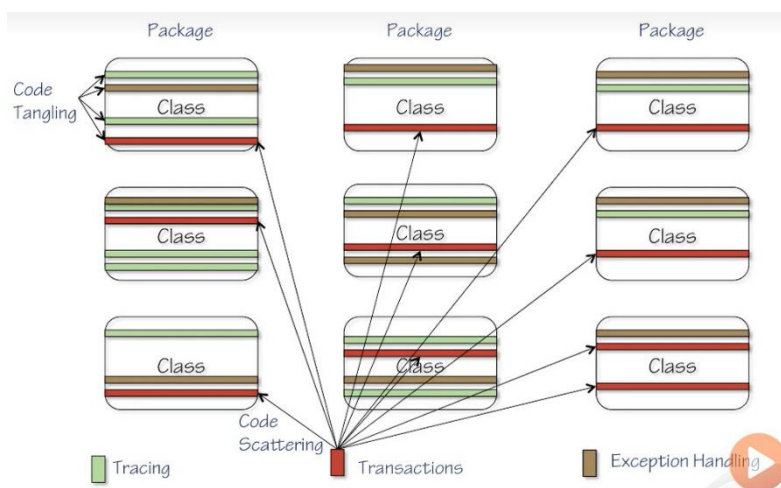


Figura 2.4.1: clases de un proyecto sin AOP

Esto genera dos problemas, el código es difícil de leer ya que, además de la lógica de negocio, contiene distintos aspectos, como puede ser manejo de excepciones, trazabilidad o transacciones, lo que hace difícil ver qué está realizando en realidad un método. Por otro lado, se genera código disperso, ya que, en el caso del manejo de excepciones, por ejemplo, el código queda separado.

Usar AOP permite resolver estos problemas, de forma que todos los fragmentos de código relativos al mismo aspecto se agrupan en un mismo lugar, evitando tener que implementarlos en cada una de las clases, dejando una separación clara entre la lógica de negocio y cada uno de estos aspectos.

2.4.1.2 *Cross-Cutting Concerns*

Un concepto importante en AOP son los intereses transversales o cross-cutting concerns.

Los ejemplos de aspectos expuestos anteriormente de trazado, transacciones y manejo de excepciones son ejemplos de intereses transversales, muchas clases y métodos deben implementarlos y de alguna forma, cortan el código. No se pueden implementar en un solo lugar sólo utilizando programación orientada a objetos, por tanto, no se podrá evitar la fragmentación de código y aumentará su dificultad de lectura, como se ha visto anteriormente. [12]

AOP permite centralizar la implementación estos intereses transversales en un solo lugar. De esta forma, un desarrollador, con el uso de AOP, podrá implementar primero la lógica de negocio y, después, preocuparse por implementar los intereses transversales, para lo que se puede utilizar algunos de los aspectos que Spring ofrece.

De esta forma se puede componer la infraestructura. Se puede elegir el momento en el que las transacciones de seguridad y el trazado deberían ejecutarse y su lugar en el código. Así, se puede crear un middleware propio, personalizado e individual.

Se puede usar Spring AOP o AspectJ para añadir los aspectos a la aplicación para que estos se puedan ejecutar en el momento adecuado. Estas dos implementaciones son de las más comunes y potentes.

2.4.2 **Inversión de Control**

La **Inversión de Control (IoC)** es un principio que altera el control tradicional del flujo de ejecución de un programa, de esta forma, el código especifica respuestas deseadas a ciertos sucesos o solicitudes de datos, dejando que un componente externo tome el control. Spring se basa en el API de Java Reflexion para implementar este patrón. [15]

Mediante el uso de frameworks, se utiliza este patrón de forma habitual, y es el mismo framework el que toma el control y define el flujo de actuación. De esta forma, quien ejecuta el código del usuario, es el framework.

El principio de reflexión es una propiedad de los programas para manipular las clases en tiempo de ejecución. Esto significa que, con esta técnica, se puede llegar a

alterar la estructura de atributos o métodos de una clase en tiempo de ejecución. En el caso de Spring, se implementa mediante inyección de dependencias.

Además de ser utilizado en frameworks, IoC se utiliza en algunos patrones como el Observer, Factory, Strategy o Template.

Este principio ayuda a ampliar la funcionalidad de una aplicación sin tener que modificar las clases. Por tanto, juega un papel importante en la extensibilidad del código.

2.4.3 Inyección de Dependencias

El patrón de inyección de dependencias o patrón del contenedor es un patrón que consiste en suministrar a una clase sus dependencias, en lugar de que sea esta quien las instancie. De esta forma, se consiguen dependencias ligeramente ligadas y se evitan las fuertemente ligadas.[3]

Para poder suministrar estas dependencias, el patrón hace uso del llamado container, que será el encargado de almacenar y conectar todas las dependencias necesarias para que la aplicación funcione.

Como ejemplo se tiene el caso práctico de la siguiente figura.

```
3. public class ServicioImpresion {
4.
5.     ServicioEnvio servicioA;
6.     ServicioPDF servicioB;
7.
8.     public ServicioImpresion(ServicioEnvio servicioA, ServicioPDF servicioB) {
9.
10.        this.servicioA= servicioA;
11.        this.servicioB= servicioB;
12.    }
13.    public void imprimir() {
14.
15.        servicioA.enviar();
16.        servicioB.pdf();
17.    }
18. }
```

Figura 2.4.2:ejemplo inyección de dependencias

En el ejemplo, se definen dos atributos, que son pasados por parámetro en el constructor y utilizados más tarde en el método imprimir. Desde el main, se inyectarán las dependencias a través del constructor definido para que la clase ServicioImpresión pueda hacer uso de las instanciaciones.[16]

De esta forma, el servicio no será el responsable de definir sus dependencias, dando la oportunidad de mejorar la extensibilidad del programa y el desacoplamiento entre clases.

Se podría ahora, en caso de necesitarlo, extender la funcionalidad de alguna de las clases inyectadas y con sólo cambiar la instanciación del objeto padre por el hijo en el main, la aplicación estaría ejecutando los cambios añadidos.

Este patrón permite inyectar clases y añadir funcionalidades transversales. En frameworks como Spring, se añaden múltiples aspectos a las clases implementadas por el usuario y aumenta la complejidad.

2.4.4 Inversión de Dependencias

El principio de **Inversión de Dependencias** es una parte importante de Spring ya que es muy utilizado por el framework. El principio indica que los componentes de alto nivel no deben depender de los de bajo nivel. [17]

Es el último de los principios SOLID y consiste en desacoplar módulos software, siguiendo dos reglas:

Los módulos de alto nivel no deben depender de los de bajo nivel, sino que ambos módulos deberían depender de abstracciones.

Las abstracciones no deben depender de los detalles, son estos los que deben depender de las abstracciones.

2.4.5 MVC

Este es un patrón que actualmente está bastante extendido por su facilidad de uso y adaptabilidad. Sus tres componentes son la vista, el modelo y el controlador.

La **vista** presenta la información y lógica de negocio del modelo correspondiente mediante una interfaz de usuario

El **modelo** es la representación de los datos de la aplicación. Gestiona los accesos a estos datos, ya sean consultas o actualizaciones. Envía a la vista los datos que se le solicitan para que los muestre. Estas peticiones de acceso o modificación llegan al modelo a través del controlador.

El **controlador** se encarga de tratar las peticiones del usuario y generar una respuesta a las mismas. La petición llega, y será dirigida a un controlador determinado, dependiendo del tipo de petición del usuario e interactuará con alguna lógica de negocio que producirá datos, podría ser un servicio web, una base de datos o simplemente una capa de lógica de negocio, que producirá una salida. [10]

Esta salida será devuelta al controlador que deberá seleccionar una vista, basándose en el tipo de respuesta dada por la lógica implementada y será devuelta al usuario.

El controlador interpreta las entradas del usuario y las transforma para un modelo. Para obtener esa información, debe volver de la lógica de negocio y construir con ello un modelo, que se lo devolverá a la interfaz de usuario.

También interpreta las excepciones que puedan ocurrir en la aplicación y cómo continuar a partir de ese punto. Estas excepciones pueden ser el control de datos de entrada correctos, para evitar errores introducidos por el usuario o que la base de datos no funcione correctamente y mostrar la página de error correspondiente [4].

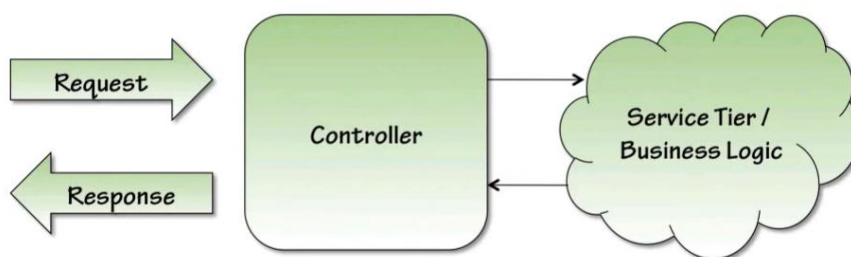


Figura 2.4.3: Manejo de peticiones

Una petición llega a la aplicación a través de la vista por cualquier evento causado por un usuario y después es interpretado por el controlador. El controlador puede realizar cambios en el modelo y, a continuación, seleccionará una vista basándose en la acción realizada. El modelo puede actualizar la vista mediante la base de datos cambiada por las acciones del usuario. En la siguiente figura se muestra el diagrama habitual de MVC. [10]

El modelo puede ser a veces confuso para aplicaciones de menor complejidad. En realidad, no se debe aplicar a tantas aplicaciones web, el concepto del patrón sigue presente, pero normalmente no es necesario que el modelo esté actualizando la vista al cambiarla.

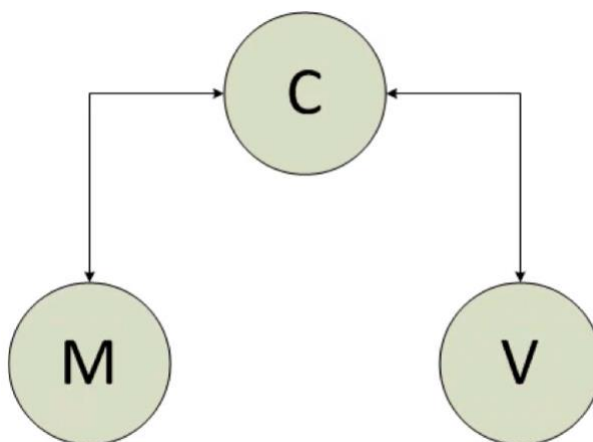


Figura 2.4.4: Diagrama MVC simplificado

El gráfico anterior es una mejor aproximación a la aplicación del patrón en desarrollo web ya que es más realista con el ciclo de vida utilizado en una aplicación web. La vista puede seguir accediendo al modelo, pero debe hacerlo a través de un controlador. Las peticiones seguirán llegando al controlador, que será encargado de devolver la vista correspondiente y actualizar el modelo.

2.4.6 Método Plantilla

El método plantilla, o Template Method, es un patrón de diseño útil para aprovechar reutilización del código, y es común encontrarlo en librerías y frameworks. Los Containers IoC usan este patrón para permitir incluir componentes en sus frameworks.

El objetivo real del patrón es envolver el algoritmo de la solución buscada. El método plantilla fuerza a utilizar un algoritmo, pero permite al usuario definir algunas de sus piezas.

El diseño del patrón está basado en una clase Abstracta Base, esta clase base será la responsable de llamar a sus clases hijas, y no al contrario. Sus operaciones deben ser sobrescritas, realizando un override. Este diseño se muestra en la siguiente figura.

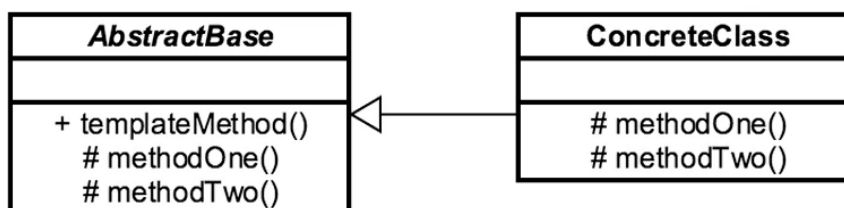


Figura 2.4.5:Diagrama de clases del método plantilla

El diagrama UML del patrón es, en realidad, sencillo, como se puede ver. La clase AbstractBase tiene un punto de entrada, templateMethod() en este caso, al que el cliente llama. Sus operaciones, methodOne() y methodTwo(), deben ser sobrescritas en la clase hija, ConcreteClass. Esta clase realiza el override de los métodos para completar el algoritmo definido en templateMethod().

Algunos de los errores comunes que se pueden cometer al intentar implementar este método son, por ejemplo, no definir adecuadamente las restricciones de acceso a los métodos, o crear una jerarquía de clases confusa, ya que el algoritmo se divide entre varias clases, cuando lo habitual es que esté contenida en una sola y, por último, dependiendo del número de plantillas que se añadan, puede resultar en un flujo de programa complicado.

2.4.7 Bean Scopes

El núcleo del framework Spring son los mecanismos de creación y gestión de beans dentro del container. Los beans en el container Spring se pueden crear en 5 ámbitos distintos, que son los siguientes: [19]

2.4.7.1 Singleton:

Es válido en cualquier configuración. El patrón de diseño Singleton hace que una clase sólo pueda ser instanciada una única vez. El Singleton es el Bean Scope por defecto dentro de Spring, por tanto, si a un Bean no se le asigna un Scope, automáticamente será tratado como un Singleton.

Este patrón garantiza una única instancia de un objeto y el control de un recurso y, además, es lazy load, que implica que el objeto no se instancia hasta que no se necesita, lo que encaja perfecto para un patrón de creación, como es el caso. Otro ejemplo de uso, además de los Beans de Spring son los controladores gráficos, en este caso, sólo se necesita una instancia para realizar la tarea.

El singleton es responsable de su creación y el control de su ciclo de vida es de naturaleza estática, aunque normalmente no se implementa utilizando una clase

estática. La razón por la que no se usa una clase estática es que necesita ser thread-safe y una clase estática no lo garantiza. Que sea thread-safe significa que debe garantizar el acceso a recursos compartidos de forma segura, independientemente del número de hilos.

Hay una instancia privada en el Singleton de la clase y un constructor privado, esto es porque se necesita que sea el singleton el que llame al constructor y sólo pueda él. No hay parámetros en el constructor, si los requiere, normalmente será un patrón factory y va contra las reglas de singleton. Esta arquitectura se muestra en el siguiente diagrama.



Figura 2.4.6: Clase Singleton

A modo de ejemplo, la siguiente figura representa el uso de un singleton. En ella, se crea una clase, DbSingleton, que instancia el patrón. Como se puede ver, hay una declaración privada de la misma clase y su constructor también es privado, por último, se añade un método que devuelve la instanciación de la clase.

Además, para conseguir que sea lazy-load, se instancia el atributo privado de la clase. Mediante este cambio, se consigue instanciar el objeto sólo cuando se necesita y no al arrancar la aplicación, lo que puede significar una gran mejora del rendimiento de la aplicación.

Por último, para añadir la seguridad en hilos, se añade la palabra reservada “volatile” a la instancia del singleton, de esta forma, se asegura que el objeto continúa siendo un singleton ante cualquier cambio de JBM.

Para asegurar que no se usa reflexión en el código se lanza una excepción en el constructor en caso de que su instancia no sea null. Se añade la sincronización en el get para asegurar seguridad en hilos, es importante comprobar que la instancia no es null dos veces para evitar problemas con otros hilos en caso de que otro esté en el mismo punto del código.

```
DbSingleton.java DbSingletonDemo.java
5 private static volatile DbSingleton instance = null;
6
7 private DbSingleton() {
8     if(instance != null) {
9         throw new RuntimeException("Use getInstance() method to create");
10    }
11 }
12
13 public static DbSingleton getInstance() {
14     if(instance == null) {
15         synchronized(DbSingleton.class) {
16             if(instance == null) {
17                 instance = new DbSingleton();
18             }
19         }
20     }
21
22     return instance;
```

Figura 2.4.7: Singleton

Algunos errores comunes del singleton a tener en cuenta son los siguientes.

Se utiliza demasiado, una vez descubierta la potencia y simplicidad del singleton, se tiende a utilizarlo todo como singleton cuando no es necesario, aunque no suele dar problemas de rendimiento, si se hace todo mediante este patrón, acabará por ralentizar la aplicación.

Muchas veces es difícil realizar pruebas unitarias en ellos debido a sus miembros privados.

Si no se tiene cuidado en su implementación, pueden no ser thread-safe.

A veces es confundido con el patrón Factory al hacer que el método getInstance necesite parámetros, una buena regla a tener en cuenta es que, si este método necesita argumentos, deja de ser un singleton para convertirse en un Factory.

2.4.7.2 Prototype

También válido en cualquier configuración. El patrón de diseño prototype garantiza una instancia única por petición. Cada vez que se realiza una petición de un bean al container, se garantiza una instancia única. Es lo contrario del Singleton, ya que da una instancia única con cada petición.

El patrón Prototype se utiliza cuando el tipo de objeto a crear está determinado por una instancia prototipo, que es clonada para crear la instancia, muchas veces se utiliza para obtener una instancia única del mismo objeto.

Uno de los motivos para elegir este patrón es evitar una creación costosa. Elegir el patrón prototype, muchas veces no es tan simple como elegir otro patrón. También evita crear subclases, normalmente se crea una instancia del prototipo con el que se trata de trabajar.

Normalmente el patrón no usa la palabra reservada “new”, la primera instancia podría utilizarla, pero después de ello, son clonadas. Aunque no es necesario, es bueno utilizar interfaces para trabajar con el patrón prototype. El patrón es normalmente implementado con algún tipo de registro, el objeto original es creado, después se almacena en el registro y, cuando se necesita otro objeto, se clona aquel.

Si un objeto es costoso de crear, pero se puede obtener todo lo necesario copiando las variables miembros, entonces el prototype es una buena opción. El patrón normalmente implementa Clone y Cloneable, método e interfaz, esto evita utilizar la palabra reservada “new” y, normalmente, si la creación es costosa, será al hacer la llamada a “new”.

Aunque se realiza básicamente una copia, cada instancia es única. El coste de la construcción no la soporta el cliente. Al contrario que en el singleton, se pueden utilizar parámetros al clonar si fuesen necesarios, aunque normalmente no lo son. El arquitecto deberá decidir si realizar una copia shallow o una en profundidad, una copia shallow copia las propiedades inmediatas, mientras que una copia en profundidad copiará cualquiera de las referencias a objetos también. Su arquitectura se muestra en la siguiente figura.

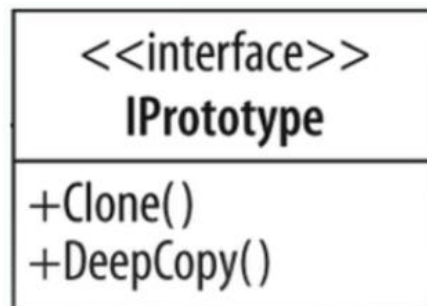


Figura 2.4.8: diagrama de Prototipo

Como ejemplo, se incluye el código de la imagen siguiente, de la clase Statement, en la que se implementa el patrón prototype. En esta clase, se prepara un constructor público en el que se guardan los atributos de la misma y se añade un método clone(), dado por la clase Cloneable, en el que se realiza una llamada al método de esta clase, en el que se realiza la copia del objeto creado.

En este ejemplo, se implementa un patrón prototype en profundidad, ya que, al clonar el objeto, se copian los objetos que necesite esta clase. Es decir, al crear dos instancias de Statement, una por constructor y otra por el método clone, los objetos record que implementa la clase, serán los mismos y tendrán la misma referencia.

```
public class Statement implements Cloneable {

    private String sql;
    private List<String> parameters;
    private Record record;

    public Statement(String sql, List<String> parameters, Record record) {
        this.sql = sql;
        this.parameters = parameters;
        this.record = record;
    }

    public Statement clone() {
        try {
            return (Statement) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

Figura 2.4.9: Ejemplo de implementación de Prototipo

Alguno de los errores comunes al utilizar prototype son los siguientes. Se usa menos de lo que se debería, por no tener claro cuándo utilizarlo. Otro error es que normalmente se usa con otros patrones. Por último, a veces se necesita una copia en profundidad y la interfaz Cloneable sólo realiza una shallow, aunque se puede implementar, se necesita más código y es entonces cuando se replantea el uso de prototype.

2.4.7.3 Request

Válido sólo en proyectos web. Devuelve un Bean por cada petición HTTP, bastante parecido a Prototype, con la diferencia de que ciclo de vida del bean es muy corto, en cuanto se completa la petición, el bean queda fuera de ámbito y el recolector de basura se encarga de él.

```
1 // Beans basados en anotaciones @Bean
2 @Bean
3 @Scope(value = WebApplicationContext.SCOPE_REQUEST)
4 public BeanSample beanSample() {
5     return new BeanSample ();
6 }
7
```

Figura 2.4.10: ejemplo de uso de Request

2.4.7.4 Session

Session es solamente válido en proyectos web. Devuelve un Bean por cada sesión y se mantendrá instanciado tanto tiempo como dure la sesión del usuario.

```
1 // Beans basados en anotaciones @Bean
2 @Bean
3 @Scope(value = WebApplicationContext.SCOPE_SESSION)
4 public BeanSample beanSample() {
5     return new BeanSample ();
6 }
7
```

Figura 2.4.11: ejemplo de uso de Session

2.4.7.5 GlobalSession

Al igual que Session y Request, Global solo es válido en proyectos web. Cuando la aplicación trabaja con un contenedor portlet, cada portlet tiene su propia sesión, para hacer que una variable está disponible para todas estas sesiones, se debe declarar como GlobalSession.

```
1 @Bean
2 @Scope(value = WebApplicationContext.SCOPE_GLOBAL_SESSION)
3 public BeanSample beanSample() {
4     return new BeanSample ();
5 }
```

Figura 2.4.12:ejemplo de uso de GlobalSession

2.5 Configuración de Spring

2.5.1 Configuración mediante XML

La configuración mediante XML fue el primer método disponible en Spring. Algunas configuraciones son, todavía, más sencillas mediante este método que mediante una configuración con Java.

Utilizando un archivo XML separado para realizar la configuración, se puede ordenar el código por asuntos de forma orgánica y consiguiendo, en algunos casos, código más limpio y ordenado.

2.5.1.1 Añadiendo un fichero XML al proyecto

Para añadir esta configuración, hay que añadir un archivo xml al proyecto. Normalmente, el nombre del fichero suele ser applicationContext.xml o parecido. Aunque este nombre es indiferente, Spring puede detectar ficheros por su cuenta si se utiliza este convenio.

Para añadir un archivo xml de configuración se puede añadir un archivo xml vacío o utilizar el asistente de Spring. En caso de elegir esta última opción, utilizando eclipse, se preparará un archivo con la cabecera y configuraciones elegidas por el usuario, además de detectarlo como un archivo de configuración, dando así más opciones en su interfaz que en el caso de haber creado un xml vacío. Para utilizar este asistente es necesario instalar Spring 3.0, ya que esta versión da prioridad a trabajar con ficheros xml sobre la configuración en Java.

Los objetos creados en este fichero son poco más que parejas de nombre y valor. Puede ser utilizado como un registro en el que ver los Beans que hemos definido para Spring.

Al añadir un nuevo archivo de configuración XML se añaden varias partes en la cabecera de este, tal como se muestra en la siguiente figura.


```
<?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.xsd">

</beans>
```

Figura 2.5.1: cabecera configuración xml

Comienza con una etiqueta <beans>. En ella, y en orden de aparición, se encuentra el namespace de beans, la instancia del esquema del xsi y, por último, la localización del esquema.

Esta localización del esquema se añade al fichero xml y será el que nos de ayuda contextual dentro de la aplicación. De esta forma, al trabajar con el fichero dentro de un IDE, se sugerirán nombres para utilizar.

Por último, habrá que incluir en el código Java las sentencias incluidas en la figura siguiente, que hacen referencia al fichero XML, la primera línea carga el archivo XML y la segunda guarda una nueva referencia a un Bean declarado en este fichero. De esta forma, Spring podrá utilizar el archivo con las definiciones incluidas en el mismo.

```
public class Application {
    public static void main(String[] args) {

        ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");
        CustomerService service = applicationContext.getBean("customerService", CustomerService.class);
        System.out.println(service.findAll().get(0).getFirstName());
    }
}
```

Figura 2.5.2:Instanciación fichero xml

Ahora, al ejecutar el programa se obtendrá una cabecera en la consola que indica que el fichero xml se ha cargado, que se encuentran los beans que han sido declarados y, por último, arranca la aplicación, ejecutando el código dado por el programador.

2.5.1.2 Declaración de Beans en XML

Los Beans, básicamente son objetos, POJOs que se usan en el archivo de configuración. Definir Beans puede ser comparado con instanciar objetos. De esta forma, se pueden retirar del código todas estas instanciaciones de objetos y agruparlas en un archivo XML. Así, se pueden cambiar configuraciones sin tener que recompilar el código, simplemente cambiando archivos XML.

La definición de un Bean en XML se puede realizar de la forma de la siguiente figura, en la que se define un nombre para el Bean y la clase que lo implementa, siendo, en este caso, un Bean sin argumentos.

```
<bean name="customerRepository"  
      class="com.pluralsight.repository.HibernateCustomerRepositoryImpl"></bean>
```

Figura 2.5.3: Definición de Bean en XML

Este Bean se podrá utilizar mediante inyección por setters o inyección por constructor. Mediante la inyección por setters, se utilizan los getters y setters del Bean para establecer sus atributos definidos en el archivo XML y mediante la inyección por constructor se usan los constructores definidos para realizar la misma acción.

Gracias al uso de Beans con XML, se pueden ahorrar dependencias entre clases y declaraciones dentro del código java, extrayéndolas en un fichero separado y pasándolas por referencia, reduciendo el acoplamiento.

Por otro lado, la inyección por constructor se realiza mediante índices y no mediante nombre como la del setter.

La declaración de Beans mediante este método de inyección por constructor es bastante similar al de inyección por setters, sólo que en este caso habrá que declarar un index y el constructor necesario en la clase correspondiente.

Cabe añadir que estos dos métodos de inyección pueden ser utilizados de manera conjunta, en caso de que el desarrollador lo necesite. De esta forma, se podrán definir los atributos de un Bean utilizando su constructor y completarlos mediante setters.

2.5.1.3 Anotaciones en XML

Antes de poder utilizar anotaciones en Spring mediante un fichero XML, habrá que crear y configurar el mismo para que estas puedan ser entendidas por Spring. Para ello, habrá que añadir el código de la figura siguiente al archivo XML.

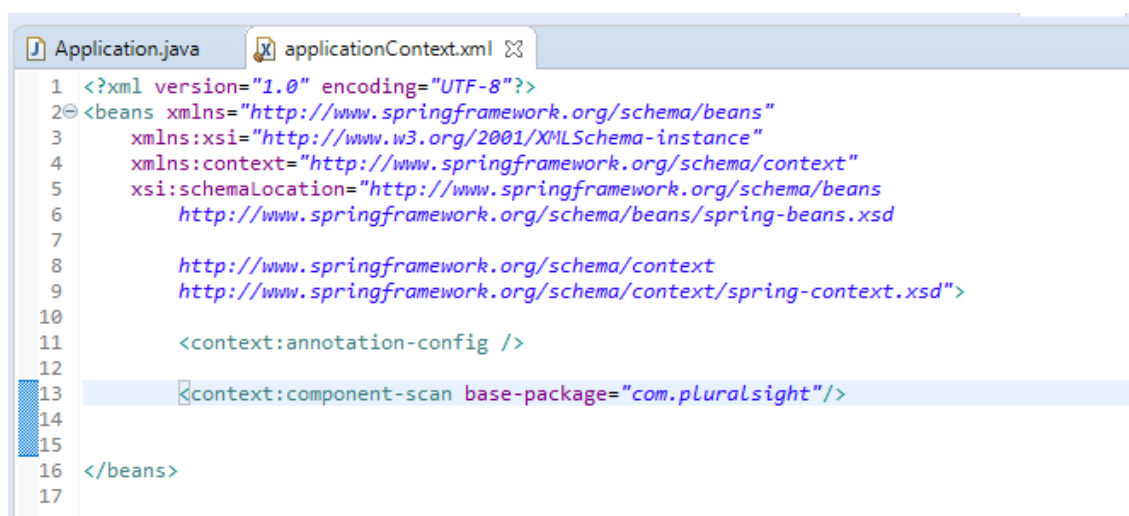


Figura 2.5.4: Configuración XML para uso de anotaciones

Este código preparará un escáner de componentes para que Spring pueda encontrar e interpretar las anotaciones que un desarrollador pueda utilizar.

Al código habitual sin este escáner, habrá que añadir las modificaciones representadas en la figura anterior. Por líneas, será necesario añadir el espacio de nombres de contexto, ubicado en la línea 4 y la localización de este esquema, en las líneas 8 y 9. Además de esta cabecera, es necesario añadir la línea 11, para hacer saber a la aplicación que se ha utilizado una configuración por anotaciones y la línea 13, para establecer una raíz bajo la que empezar a buscar las anotaciones.

El escáner de componentes es utilizado por Spring para buscar en todo el paquete especificado y sus clases para saber qué anotaciones deben ser importadas y cómo construir la aplicación.

Para incluir estas anotaciones en el código, se escriben al comienzo de una clase, añadiendo entre paréntesis el nombre correspondiente, como se muestra en la siguiente figura.

```
@Repository("customerRepository")
public class HibernateCustomerRepositoryImpl implements CustomerRepository {

@Service("customerService")
public class CustomerServiceImpl implements CustomerService { }
```

Figura 2.5.5: Ejemplos de uso de anotaciones

2.5.1.4 Archivo de propiedades

Para poder utilizar un archivo de propiedades utilizando XML, habrá que añadir el namespace context al fichero XML del proyecto, además de una línea de código que indique el fichero que se va a utilizar, esta configuración se muestra en la siguiente figura.

```
<context:property-placeholder location="app.properties"/>
```

Figura 2.5.6: archivo de propiedades con XML

Para poder utilizar este archivo de propiedades, se puede declarar el valor de una variable cualquiera en este fichero y cargarlo en un Bean desde el fichero XML a través de su nombre, marcado con \${nombreValor}.

Para que el valor pueda ser inyectado correctamente, la clase que define este Bean debe tener un atributo con el mismo nombre dado a la propiedad en la definición del Bean.

La sintaxis utilizada para el valor, con \${}, es lo que indica a Spring que debe obtenerlo del archivo de propiedades e inyectarlo en el código.

También se puede utilizar el archivo de propiedades a través de anotaciones. Para ello, solo se necesita añadir al XML las líneas que se muestran en la siguiente

imagen, la primera para indicar el uso de anotaciones y la segunda para indicar el archivo a utilizar.

```
<context:annotation-config/>  
<context:property-placeholder location="app.properties"/>
```

Figura 2.5.7: uso de anotaciones y archivo de configuración

Tras ello, se podrá definir el valor de los atributos con la anotación **@Value** y el nombre de la propiedad definida en el archivo de correspondiente.

2.5.2 Configuración mediante Java

Es uno de los métodos más nuevos para realizar las conexiones entre elementos en una aplicación Spring. Se introdujo para evitar usar XML y código fuente juntos para realizar la configuración.

En el caso de la configuración mediante Java, como es de suponer, no necesitaremos ningún fichero XML, al contrario que en el caso anterior, en el que hacíamos uso de un applicationContext.xml.

Las configuraciones más antiguas de Spring recibieron quejas por tener demasiada configuración XML. Configuraciones más recientes de Spring incluyeron namespaces para ayudar en el desarrollo de las aplicaciones, pero los desarrolladores querían ver aún menos XML o incluso nada. Así, entró la configuración Java, haciendo que casi cualquier elemento de programado en Spring pueda ser implementado completamente utilizando únicamente Java.

Para poder utilizar esta configuración mediante Java, primero es necesario incluir un fichero de configuración Java, de manera similar a como se realiza mediante XML. En este fichero se incluirán unas anotaciones que indican a Spring cómo debe tratar el fichero.

Una de estas anotaciones es **@Configuration**, utilizada a nivel de clase, indica a Spring que esta clase creada es un archivo de configuración, actuando de manera similar a un fichero de configuración XML.

Por otro lado, **@Bean**, se utiliza a nivel de método y sirve para definir un Bean en Spring y dejarlo disponible para su posterior uso en la aplicación, estos métodos podrán tener cualquier nombre, siempre que se marquen como **@Bean**.

```
6  
7 @Configuration  
8 public class AppConfig {  
9  
10     @Bean(name= "customerService")  
11     public CustomerService getCustomerService() {  
12         return new CustomerServiceImpl();  
13     }  
14 }  
15
```

Figura 2.5.8: @Configuration y @Bean

2.5.2.1 Inyección por setters

La inyección mediante setters utilizando Java, es tan sencilla como utilizar una llamada a un método.

Mientras que mediante XML hay más acciones que se acaban suponiendo, como algunas llamadas o el autowire, en la configuración Java se vuelve más transparente y visible.

La inyección por setters es básicamente el hecho de llamar al setter en el Bean. Se define un Bean utilizando las anotaciones de Bean como muestra la figura siguiente. Se pueden enlazar Beans mediante esta técnica con llamadas entre métodos, tal y como se muestra en la siguiente figura.

```
@Configuration
public class AppConfig {

    @Bean(name= "customerService")
    public CustomerService getCustomerService() {
        CustomerServiceImpl service= new CustomerServiceImpl();
        service.setCustomerRepository(getCustomerRepository());
        return service;
    }

    @Bean(name="customerRepository")
    public CustomerRepository getCustomerRepository() {
        return new HibernateCustomerRepositoryImpl();
    }
}
```

Figura 2.5.9: Inyección por setters en Java

En este caso, se crea un Bean, customerRepository, que será utilizado en customerService mediante un set para completar la instanciación de este Bean.

La diferencia en la utilización de este método con o sin Spring es que al marcarlo como Bean, Spring busca en su repositorio y su contexto para ver si hay algún Bean ya creado con ese nombre o de esa instancia y los inyecta en esa misma definición del Bean. De esta forma se evita crear una instancia del Bean cada vez.

2.5.2.2 Inyección por constructor

Es muy parecido a la inyección por setters. Al igual que en este tipo de inyección, habrá que crear una instancia de un Bean y, en lugar de llamar al setter, habrá que llamar al constructor definido para esa instancia.

2.5.2.3 Declaración de Scopes

La configuración para utilizar scopes mediante Java es sencilla. Simplemente habrá que añadir una anotación **@Scope** al código e incluir el nombre del scope en un string, en caso de no declarar este string, por defecto se instanciará como singleton. Necesita un jar AOP, pero, al utilizar Maven, ya está disponible en el proyecto.

```
13 @Scope("singleton")
14 public class CustomerServiceImpl implements CustomerService {
```

Figura 2.5.10: Ejemplo de uso de @Scope

2.5.2.4 Archivo de Propiedades

Cargar un fichero de propiedades en Java es bastante parecido a hacerlo en XML, aunque se puede ver algo más de detalle al cargarlo mediante Java.

Para cargar el fichero, el primer paso será declararlo mediante una anotación en la que se indique su nombre. A continuación, habrá que declarar un Bean en el archivo de configuración, equivalente al elemento “property-placeholder” declarado en la configuración XML, que instancie un objeto de tipo “PropertySourcesPlaceholderConfigurer”.

Lo que hace esta clase es leer todas las propiedades y valores definidos en el archivo declarado, almacenarlas y dejarlas disponibles en el contexto para utilizarlas en la aplicación. La instanciación del Bean se muestra en la siguiente figura.

```
@PropertySource("app.properties")
public class AppConfig {

    @Bean
    public static PropertySourcesPlaceholderConfigurer getPropertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }
}
```

Figura 2.5.11: declaración del archivo de propiedades

Una vez añadido esto al archivo de configuración, se podrá utilizar la anotación **@Value** para definir los valores de los atributos dentro de las clases, del mismo modo que se puede realizar con las anotaciones usando la configuración XML.

2.5.3 Autowire

En un principio, Spring tenía mala reputación por tener demasiada configuración XML y los desarrolladores tenían que realizar la conexión de cada Bean y cada referencia. Para evitarlo, se introdujo un mecanismo llamado Autowire.

Mediante el Autowire Spring puede, de forma automática, enlazar y configurar Beans dentro de la aplicación para que esta pueda ejecutarse de manera correcta sin necesidad de que el desarrollador enlace cada uno de los Beans.

Hay cuatro tipos distintos de Autowire:

- **byType:** permite que un atributo sea conectado automáticamente si hay exactamente un Bean de ese atributo contenido en el Container. Por ejemplo, si se declara un Bean de un objeto con atributo de tipo entero y declaramos un solo entero, mediante la conexión byType, se puede realizar de manera automática. Si hubiese dos beans de la misma clase con distinto nombre daría error porque Spring no puede detectar a cuál hace referencia.
- **byName:** busca elementos que coincidan por nombre. En este caso, Spring buscará un setter con el mismo nombre que el pasado por referencia. Este

método soluciona el problema de `byType`. Ambos métodos son útiles ya que `byType` nos permite tener una única instancia de la clase y `byName` nos permite tener varias y elegir un objeto específico por el nombre.

- **Por Constructor:** busca objetos de los tipos definidos por el constructor para inyectarle los argumentos a este.
- **No o none:** si se especifica como “no”, significa que no se puede aplicar Autowire para nada.

En caso de estar utilizando anotaciones con XML, se puede utilizar Autowire para inicializar los Beans mediante estas anotaciones en el código Java. Este Autowire se puede realizar en variables miembro de la clase, constructores e inyección por setter.

Para realizar un Autowire de las variables de la clase utilizando anotaciones, se puede realizar, tal como aparece en la figura siguiente, siendo `customerRepository` una variable de la clase. Utilizando el escáner de componentes, Spring buscará cualquier código marcado con anotaciones estereotipo y `autowired` para enlazar los atributos de la aplicación correctamente.

```
@Autowired  
private CustomerRepository customerRepository;
```

Figura 2.5.12:@Autowired con variables miembro

Mediante este método, se pueden llegar a instanciar Beans sin necesidad de declararlos en un fichero XML, sólo mediante anotaciones en ficheros Java.

Realizar un Autowire mediante setters y anotaciones es también sencillo y, en algunas ocasiones, es parecido a como sería desarrollar una aplicación sin Spring. Sólo hay que generar el setter y añadir la anotación sobre este para que Spring pueda detectarlo, tal como se muestra en la figura siguiente.

Tal como ocurría en el caso anterior, el escáner se encargará de buscar los elementos marcados con las anotaciones para después realizar el Autowire correspondiente.

El Autowire mediante constructor y anotaciones es bastante parecido al método anterior mediante setters. Para realizar este método, hay que crear un constructor para el tipo de objeto en el que queremos inyectar la instanciación y añadir la anotación.

En caso de querer volver a utilizar anotaciones con variables o setters habrá que tener en cuenta que se eliminará el constructor por defecto para evitar efectos inesperados.

```
@Autowired  
public CustomerServiceImpl(CustomerRepository customerRepository) {  
    System.out.println("Utilizando inyección por constructor");  
    this.customerRepository = customerRepository;  
}
```

Figura 2.5.13:@Autowired con constructor

De esta forma, se puede evitar que unas clases instancien otras, como se ha demostrado en los tres ejemplos anteriores, en los que no ha sido necesario instanciar de forma explícita en el código un objeto `customerRepository` ya que se encargaba el `Autowire` de esta tarea, con ayuda de las anotaciones.

De este modo, y con ayuda de interfaces, se logra un código muy desacoplado, lo que ayuda en tareas de desarrollo y mantenimiento de código.

Con la configuración en Java, también se puede realizar el `Autowire` mediante **@ComponentScan**. Se utiliza a nivel de clase, indicando entre paréntesis la raíz en la que debe empezar a escanear Spring los componentes para realizar el `Autowire`, muy similar al escáner de **@Component** utilizado en XML. En caso de no indicar esta raíz, se toma por defecto dónde se encuentra la clase.

De esta forma sencilla, Spring se queda preparado para escanear anotaciones e inyectarlas en aquellos lugares donde sean necesarias.

Ahora, para poder usar realmente el `Autowire`, habrá que utilizar esta anotación sobre el atributo o el setter que se quiera utilizar para realizar esta acción. De esta forma, Spring, utilizando el escáner definido, sabrá dónde debe utilizar el `Autowire` para conseguir realizar la inyección de dependencias que busca el desarrollador.

Utilizando `Autowire` podemos ir todavía un paso más allá y, sin definir ningún Bean, utilizando el escáner ya definido y las anotaciones **@Repository**, **@Service** y **@Autowired**, Spring se encargará de realizar la instanciación e inyección mediante setters.

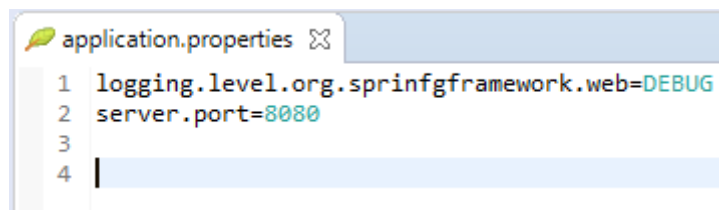
Simplemente declarando utilizando las anotaciones **@Repository**, **@Service** o **@Component**, se pueden utilizar los beans instanciados de estas clases en cualquier otra a través de un atributo marcado como `autowired` o utilizando la inyección por constructor o setter, en caso de que así se requiera.

2.5.4 Archivo de Propiedades

Son una buena forma de abstraer valores que pueden cambiar con cada entorno. En muchas ocasiones, hay URLs, contraseñas y distintas formas de conexión que necesitan ser extraídas del código fuente. Una buena forma de extraer esta información de configuración es utilizar archivos de configuración con Spring.

Utilizando Spring, se pueden añadir archivos de propiedades para facilitar el trabajo de desarrollo. Los autoconfiguradores de Spring permiten añadir muchos comportamientos con distintas características cada uno mediante propiedades simples, facilitando así la tarea del desarrollador.

A continuación, en la figura siguiente se presenta un ejemplo de propiedades comunes disponibles para añadir a un proyecto de Spring Boot a través de uno de estos archivos.

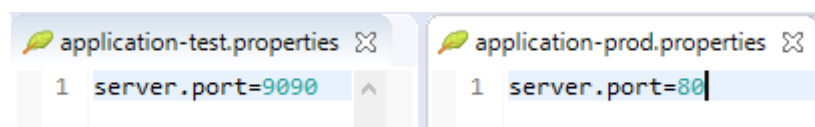


```
application.properties
1 logging.level.org.springframework.web=DEBUG
2 server.port=8080
3
4
```

Figura 2.5.14:propiedades de login y puerto

Además, se pueden añadir varios archivos de configuración, dependiendo del estado de la aplicación. De esta forma, se pueden realizar cambios de estado sin dificultad.

En este caso, a modo de ejemplo en la figura siguiente, se añaden dos archivos, que, como sus nombres indican, servirán para los estados de producción y test. Spring no utilizará las propiedades de estos archivos a menos que se le especifique el perfil correspondiente.



```
application-test.properties
1 server.port=9090

application-prod.properties
1 server.port=80
```

Figura 2.5.15:propiedades en test y producción

Al trabajar con distintos perfiles de Spring, el framework cargará primero las propiedades generales y, a continuación, sobrescribirá aquellas que sean específicas del perfil que esté seleccionado. En el caso del ejemplo, se mantendrá el nivel de log in del archivo principal del ejemplo anterior, pero se sobrescriben los puertos utilizados.

La documentación de Spring ofrece una lista de propiedades que se pueden añadir y ajustar a un proyecto. Actualmente, cuenta con una gran cantidad de propiedades, y trabaja con varias librerías y Frameworks.

2.6 Spring Boot

2.6.1 Cómo funciona Spring Boot

Primero, la aplicación arranca desde la clase principal utilizando el main estándar público y estático, al igual que cualquier otra aplicación de Java.

A continuación, Spring Boot inicializa el contexto de Spring, que comprende la aplicación Spring y acepta cualquier inicializador autoconfigurado, la configuración y anotaciones que dictan cómo inicializar y arrancar el contexto de Spring.

Por último, en caso de aplicaciones web, un servlet embebido se arranca y se autoconfigura, esto se realiza en segundo plano y evita tener que utilizar un fichero web.xml. El servlet container por defecto de Spring es Tomcat, pero hay opción de cambiarlo con Jetty o de cambiar opciones de configuración, como el puerto del servidor.

En la clase principal, se encuentra el método main. Al arrancar la aplicación, se comienza por este método main y el JVM lo inicializa y le pasa el control de la

ejecución. El contenido dentro el método es utilizado para ejecutar y cumplir con la lógica y el código de la aplicación.

Después, se añade la anotación **@SpringBootApplication** a la aplicación. Esta indica a Spring Boot cómo inicializar y arrancar Spring. Esta anotación, en realidad, ejecuta otras anotaciones en segundo plano, de este modo, se puede evitar escribir cada una de las anotaciones y utilizar esta por conveniencia.

Las principales anotaciones que trae consigo **@SpringBootApplication** son **@Configuration**, **@EnableAutoConfiguration** y **@ComponentScan**, la lista completa se detalla en la siguiente figura.

```
42 @Target(ElementType.TYPE)
43 @Retention(RetentionPolicy.RUNTIME)
44 @Documented
45 @Inherited
46 @Configuration
47 @EnableAutoConfiguration
48 @ComponentScan
49 public @interface SpringBootApplication {
50
```

Figura 2.6.1: anotaciones de SpringBootApplication

La anotación **@Configuration** clasifica la clase Java como configuración para el contexto de Spring que Spring Boot y Spring utilizarán para inicializar y configurar varios componentes y ajustes de entorno utilizados por Spring.

La siguiente, **@EnableAutoConfiguration** es una anotación específica de Spring Boot, se encarga de indicar a Spring Boot que, en caso de encontrar algún subproyecto de Spring o cualquier framework compatible con Spring en la ruta de la clase, debe intentar autoconfigurarlo, conectarlo e integrarlo automáticamente con Spring. Esta anotación hace mucha parte del trabajo de Spring al combinarlo con el BOM de Maven proporcionando por el spring-boot-starter-parent.

Por ejemplo, al añadir la dependencia spring-boot-starter-web en el archivo pom.xml, se añade el subframework Spring MVC y es esta anotación, la que indica a Spring Boot que automáticamente lo prepare para poder utilizar controladores Spring sin tener que hacer más trabajo de integración.

La última anotación, **@ComponentScan**, indica a Spring y a Spring Boot que busque cualquier componente de Spring, ya sean controladores, servicios, repositorios o cualquier otro componente, comenzando en el paquete de la clase que contiene el main y de forma recursiva busque en todos los paquetes inferiores, por tanto, para que el escáner pueda encontrar los componentes, será necesario dejar la clase principal en un nivel superior.

Por último, la clase SpringApplication es llamada en el main y su método estático run es llamado. Este código inicializa el contexto de Spring, aplica todas las anotaciones y clases de configuración, busca cualquier componente Spring y los sitúa en el contexto de Spring y, por último, prepara y configura el container embebido.

2.6.2 Propiedades y configuración de Spring Boot

Cambiando la configuración de Spring Boot, se pueden obtener comportamientos distintos a los establecidos por defecto, lo que da al desarrollador mayor libertad de trabajo.

Para ello, son importantes las propiedades de la aplicación, dentro de Spring Boot, se puede crear un archivo de propiedades para este fin. La mayoría de los ajustes y comportamientos de la aplicación se pueden modificar en este archivo.

Cuando este archivo, normalmente llamado `application.properties`, se coloca en la raíz del proyecto, Spring lo cargará y aplicará cualquier configuración a la aplicación cuando se inicie.

Teniendo en cuenta dónde se puede desplegar una aplicación, se puede observar que sus propiedades pueden cambiar de un entorno a otro. Estos cambios pueden ser en credenciales de desarrollo o usuario, por ejemplo.

Spring Boot ofrece una manera sencilla de manejar estos cambios al pasar de un entorno a otro. Además del archivo estándar `application.properties`, se pueden añadir más archivos de configuración con el nombre del entorno embebido en el nombre del archivo.

De esta forma, se podría crear un archivo `application-dev.properties`, en el que se incluya toda la configuración relativa al estado de desarrollo e incluir otra para producción, por ejemplo. Estos archivos de configuración específicos se cargarán sobre el archivo de propiedades principal y cualquier propiedad específica de entorno será sobrescrita y preparada para ese entorno concreto.

Otra pieza importante de Spring Boot es la configuración Java. Aunque se pueden realizar muchas configuraciones con las propiedades de Spring Boot en el archivo `application.properties`, se podría dar el caso de necesitar realizar alguna configuración para la aplicación que esté fuera de las opciones de autoconfiguración.

En versiones anteriores de Spring, la mayoría de la configuración estaba almacenada en ficheros XML, pero ahora se puede conseguir el mismo resultado utilizando código Java. Con esto se puede retirar el XML de una aplicación y permite manejar de forma ordenada cualquier tipo de necesidad de configuración de la aplicación que tenga que darse a su inicio.

Por ejemplo, en caso de querer utilizar un `DataSource` y una base de datos, en lugar de dejar el trabajo a la autoconfiguración de preparar y compilar el `DataSource`, se podría crear una clase de configuración y crear el `DataSource` de forma programática, de esta forma, se podrían definir varios `DataSources` para conectarse a varias bases de datos.

Cualquier clase Java de configuración que tenga métodos públicos con la anotación **@Bean**, se cargará al iniciar la aplicación y los valores que devuelvan sus métodos se guardarán como Beans dentro del contexto de Spring.

Este patrón de configuración básico permite añadir o configurar cualquier bean al contexto de Spring.

2.6.3 Gestión de dependencias con Spring Boot

Spring Boot facilita la tarea de integración de librerías y frameworks en un proyecto debido a su estructura de manejo de dependencias. Spring Boot llama a su colección inteligente de dependencias la lista de materiales o Bill of materials (BOM). Gracias a este BOM, los desarrolladores pueden utilizar frameworks con las versiones que mejor compatibilidad tienen entre ellos, gracias a una selección de frameworks y versiones hecha por los desarrolladores de Spring.

Al añadir el artefacto “spring-boot-starter-parent” al archivo de construcción de Maven, el proyecto hereda toda la administración de dependencias, además de otros plugins, la versión de soporte mínimo de Java, filtro de recursos y más por defecto. Esta declaración de herencia simple está realizando una parte pesada del proyecto.

Un ejemplo simple para comprobar cómo funciona este BOM es cambiar la versión del artefacto “spring-boot-starter-parent”, al hacerlo, se podrá comprobar cómo la versión del resto de dependencias incluidas ha variado para mejorar la compatibilidad entre ellas, tal como se aprecia en la siguiente figura.

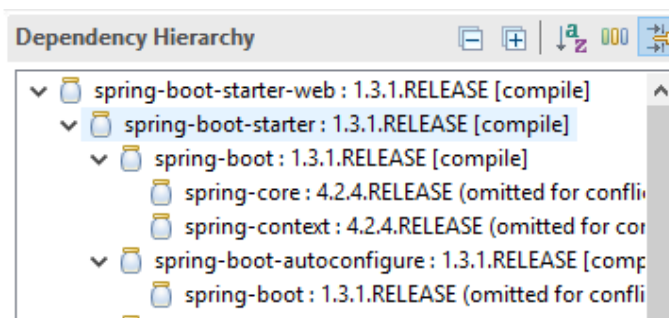


Figura 2.6.3: versión 1.3.1.RELEASE

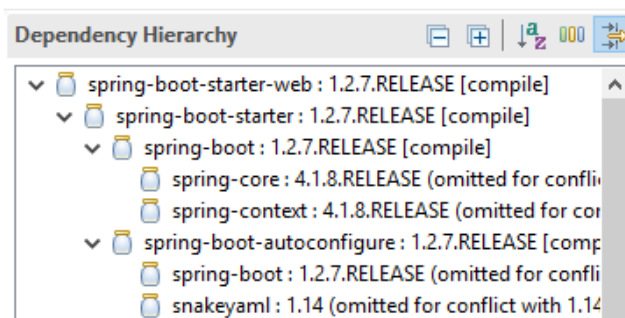
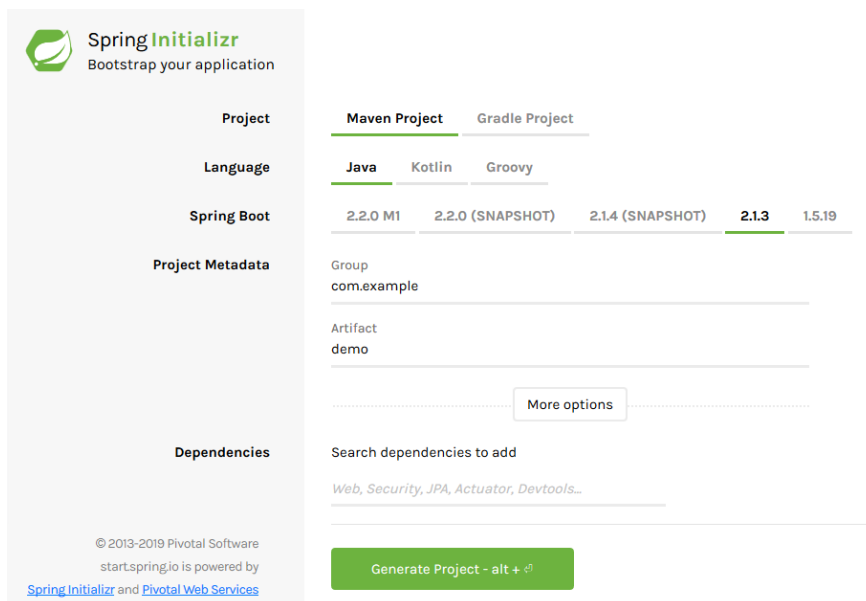


Figura 2.6.2: versión 1.2.7.RELEASE

De esta forma, Spring puede gestionar todos los detalles de las dependencias entre versiones, sólo especificando la versión del spring-boot-parent. Lo que evita la tediosa tarea de tener que definir, una por una cada una de las versiones de dependencia de los elementos añadidos al proyecto, además, asegura que su versión es la adecuada.

2.6.4 Spring Boot Initializers

Aunque crear un proyecto con Spring Boot es sencillo, Spring ofrece varias herramientas para esta tarea. Una de ellas es Spring Initializr, que permite, de una manera sencilla, seleccionar múltiples opciones para generar un proyecto y tenerlo listo para ejecutar en poco tiempo. En la siguiente figura se muestra la apariencia de la página en 2019 (start.spring.io).



The screenshot shows the Spring Initializr web interface. On the left is a sidebar with sections: Project, Language, Spring Boot, Project Metadata, and Dependencies. The main area on the right has tabs for 'Maven Project' and 'Gradle Project', with 'Maven Project' selected. Under 'Maven Project', there are tabs for 'Java', 'Kotlin', and 'Groovy', with 'Java' selected. Below these are version options for Spring Boot: '2.2.0 M1', '2.2.0 (SNAPSHOT)', '2.1.4 (SNAPSHOT)', '2.1.3' (which is highlighted), and '1.5.19'. The 'Project Metadata' section contains input fields for 'Group' (with 'com.example' entered) and 'Artifact' (with 'demo' entered). There is a 'More options' button. The 'Dependencies' section has a search bar with the text 'Search dependencies to add' and a list of suggestions: 'Web, Security, JPA, Actuator, Devtools...'. At the bottom right is a green button that says 'Generate Project - alt + ⌘'. The footer of the sidebar contains copyright information: '© 2013-2019 Pivotal Software', 'start.spring.io is powered by', and links to 'Spring Initializr' and 'Pivotal Web Services'.

Figura 2.6.4: Spring Initializr

Para crear un proyecto básico, sólo hay que rellenar la información, seleccionar la versión de Spring Boot y generar el proyecto. También permite seleccionar la integración con más frameworks y bases de datos a través del apartado de dependencias, en la que se encuentra gran cantidad de herramientas complementarias.

Por otro lado, también se puede utilizar la interfaz de línea de comandos a través del Spring Boot CLI. Esta herramienta puede inicializar proyectos al igual que la página web lo hace, de hecho, Spring Boot CLI llama a la misma API que Spring Initializr, así que es simplemente otra forma de usarlo, pero desde la línea de comandos.

Hay muchas formas de instalar Spring Boot CLI, la documentación de Spring muestra cómo usar SDKMAN, Brew y otros más. Para realizar la instalación manual, sólo habrá que descargar el zip y habrá que asegurarse que se dispone de la versión correcta del JDK en la máquina en la que se trata de instalar.

Una vez hecha de descarga y extracción de los archivos, habrá que abrir la consola en la dirección de estos archivos. Dentro debería haber una carpeta llamada bin. Una vez dentro de esta, se encuentran dos archivos ejecutables, una para sistemas basados Linux o basados en UNIX (llamado spring) y otro para Windows (llamado spring.bat).

En este punto, ya se puede realizar un nuevo proyecto de spring haciendo una llamada al archivo necesario con el método "init", indicando las dependencias y dándole un nombre a la aplicación. Al ejecutar el comando, se creará una nueva carpeta que contendrá el proyecto, ubicada en la ruta actual. La utilización y ejecución de estos comandos se muestran en la figura siguiente

```
C:\Windows\System32\cmd.exe

E:\Usuario\Downloads\spring-2.1.3.RELEASE\bin>spring.bat --version
Spring CLI v2.1.3.RELEASE
E:\Usuario\Downloads\spring-2.1.3.RELEASE\bin>spring.bat init --dependencies=web nuevaApp
Using service at https://start.spring.io
Project extracted to 'E:\Usuario\Downloads\spring-2.1.3.RELEASE\bin\nuevaApp'
E:\Usuario\Downloads\spring-2.1.3.RELEASE\bin>
```

Figura 2.6.5: nuevo proyecto Spring mediante consola

2.7 Modulo Web

Para crear aplicaciones web, normalmente, la aplicación en Spring Boot se aloja una API REST, proporcionada por Spring MVC. Es un punto de entrada y salida al servidor para el cliente web.

A modo de ejemplo, para comprender mejor los conceptos y poder utilizar código orientativo, se creará una aplicación muy sencilla. En ella, se creará un controlador MVC de Spring para manejar los dispositivos remotos o endpoints.

Para poder crear este controlador se necesitarán las anotaciones de la imagen siguiente. Con ellas, se identifica al controlador y se indica, mediante un string, la base de la URL que todos los endpoints van a contener para esta clase.

```
6 @RestController
7 @RequestMapping("api/v1/")
8 public class ShipwreckController {
9
```

Figura 2.7.1: anotaciones del controlador

A continuación, se crea el método de la imagen siguiente, para listar elementos. Una parte importante, es el mapeado de la petición y la definición del método. El mapeo de la petición, definido en la anotación con `method` y `value`, indica que acepta una petición GET al endpoint "api/v1/shipwrecks". Esta anotación, a nivel de método, se añade a la definida a nivel de clase.

```
@RequestMapping(value = "shipwrecks", method=RequestMethod.GET)
public List<Shipwreck> list(){
    return ShipwreckStub.list();
}
```

Figura 2.7.2: Ejemplo de listado de objetos

A continuación, para completar la funcionalidad de la clase, se añaden otros métodos, para crear, obtener, actualizar o borrar los elementos manejados, llamados shipwreck, en este caso.

```
@RequestMapping(value = "shipwrecks", method=RequestMethod.POST)
public Shipwreck create(@RequestBody Shipwreck shipwreck){
    return ShipwreckStub.create(shipwreck);
}

@RequestMapping(value = "shipwrecks/{id}", method=RequestMethod.GET)
public Shipwreck get(@PathVariable Long id){
    return ShipwreckStub.get(id);
}

@RequestMapping(value = "shipwrecks/{id}", method=RequestMethod.PUT)
public Shipwreck update(@PathVariable Long id, @RequestBody Shipwreck shipwreck){
    return ShipwreckStub.update(id, shipwreck);
}

@RequestMapping(value = "shipwrecks/{id}", method=RequestMethod.DELETE)
public Shipwreck delete(@PathVariable Long id){
    return ShipwreckStub.delete(id);
}
```

Figura 2.7.3: Ejemplo del resto del CRUD de shipwrecks

En estos métodos se añaden nuevas anotaciones como el **@RequestBody** y **@PathVariable**. Estos métodos son código de Spring MVC, también hay una parte de la que se encarga Spring Boot, aunque no se puede apreciar directamente en el código.

Spring Boot maneja la integración con Spring MVC y después prepara la biblioteca Jackson de JSON, así, al mandar la información del shipwreck a través de una conexión HTTP, Spring Boot y Spring MVC están, automáticamente, presentando la información JSON dentro y fuera del objeto Java shipwreck. De esta forma al desarrollador le quedará el trabajo de codificar la lógica de la aplicación.

Para integrar Spring MVC, Spring Boot ha realizado un trabajo por debajo que puede que no sea advertido, pero que sí que es importante.

Al añadir en el fichero pom.xml de Maven la dependencia spring-boot-starter-web Spring Boot hace más que simplemente incluir el jar en la ruta del proyecto. Al tener también activada la autoconfiguración con la anotación **@SpringBootApplication**, Spring Boot va un paso más allá y prepara algunas características de Spring MVC.

Lo primero que hace es preparar algunos ViewResolvers de Spring MVC de forma automática. Spring Boot indica a Spring MVC que prepare el contenido obtenido por el ViewResolver que determina cómo responder basado en el tipo de contenido. Como se trabaja con payload de JSON, Spring Boot y Spring MVC dejan disponible la librería Jackson de JSON para manejar contenido de las vistas de negocio para los tipos JSON de la aplicación.

Después, Spring Boot configura e indica a Spring MVC que debería servir recursos estáticos localizados en la raíz del directorio de recursos. Así es como una aplicación Angular se sirve al navegador ya que el contenido está basado en JavaScript, HTML y CSS.

Spring Boot después prepara algunos HTTPMessageConverters estándar para que pueda usar algunos métodos por defecto JSON a Java y viceversa. La codificación básica de strings está establecida con UTF-8 por Spring Boot y Spring MVC.

Por último, Spring Boot y Spring MVC dejan un modo para sobrescribir cualquiera de los comportamientos por defecto preparados con la autoconfiguración, dejando al desarrollador algunos hooks personalizables y programables.

Estos hooks dejan control pleno sobre cómo se integra Spring MVC y generalmente se puede programar en ellos cualquier tipo de comportamiento que se pueda necesitar. Los hooks tienen gran potencial, pero requieren un conocimiento mayor sobre las integraciones del framework y, generalmente, acaban siendo más complicados que los elementos por defecto.

2.8 Modulo Datos

En este punto, se estudiará cómo utilizar Spring Boot para integrar en una aplicación una fuente de datos en un driver JDBC apropiado.

Esto incluye trabajar con los iniciadores de Spring Boot y la autoconfiguración al tener que trabajar con el framework Spring Data JPA, entity models y las migraciones de bases de datos.

Mediante este punto, se pretende comprender, además de cómo trabajar con fuentes de datos (DataSources) en Spring Boot, cómo integrar Spring con frameworks de terceros utilizando los inicializadores de Spring.

2.8.1 Integración de H2

Mediante el uso de H2 se puede manejar y almacenar una base de datos para gestionar la información de la aplicación.

Para realizar esta tarea, principalmente, habrá que añadir las dependencias al archivo POM de Maven. Esto incluirá la base de datos H2 embebida y el controlador de soporte JDBC para esta.

La primera dependencia que añadir es el inicializador de Spring Data JPA, la segunda es la base de datos h2 efectiva. Ambas dependencias se muestran en las siguientes figuras.

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

Figura 2.8.1:dependencia JPA

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

Figura 2.8.2:Dependencia H2

Al guardar los cambios en el fichero, Maven se encargará de obtener las dependencias. Una vez obtenidas estas dependencias, a través de la pestaña de

jerarquía de dependencias, se puede observar las dependencias que ha añadido Maven de forma transitiva al proyecto.

Gracias al BOM (Bill Of Materials) de Spring y el inicializador de dependencias, la integración se obtiene sin problemas ni esfuerzo, simplemente añadiendo la dependencia del archivo POM.

Además, mediante el uso de propiedades de Spring se puede obtener de manera muy sencilla la consola de h2, lo que permite visualizar los cambios realizados al proyecto.

Mediante estas propiedades de la siguiente imagen, se activa la consola de h2 y se accede a ella a través de la ruta indicada, en este caso, añadiendo /h2 a la URL base.

```
spring.h2.console.enabled=true  
spring.h2.console.path=/h2
```

Figura 2.8.3: propiedades de consola de h2

Spring Boot autoconfigura la base de datos, así que se puede acceder directamente con los valores por defecto. Aunque en este punto aún no hay añadidos datos ni tablas en la base de datos, el proyecto ya cuenta con la base de datos y ha sido integrada en la aplicación, tal como muestra la consola h2.

2.8.2 Configuración del DataSource

En la mayoría de las aplicaciones reales, no se utiliza una base de datos embebida como en el caso de H2. Lo más probable es que sea más parecida a MySQL o Oracle, en ese caso, la base de datos estará localizada en su propio servidor y será necesario proporcionar detalles de conexión de JDBC. Con Spring Boot, se puede configurar el DataSource con el archivo application.properties.

Además de gestionar la conexión al DataSource, Spring también intenta realizar el pooling de la aplicación de forma automática. El pooling de la aplicación gestiona las conexiones a la base de datos, permitiendo que varios usuarios se conecten a la misma base de datos desde conexiones preconfiguradas. Para las aplicaciones empresariales, un buen pool puede mejorar el rendimiento y uso de la base de datos.

Si Spring detecta cualquiera de las librerías comunes de pooling en un proyecto, la autoconfiguración integrará el pool y lo dejará unido a la fuente de datos.

Al añadir la dependencia spring-boot-starter-data-jpa, se añade de forma transitiva la librería de pooling de tomcat-jdbc. Spring Boot utiliza este pool por defecto debido al rendimiento y concurrencia que ofrece.

Gracias a esta integración de la base de datos, se podrán introducir cambios persistentes, en lugar de usar la memoria dinámica.

Además, se puede añadir en el archivo de configuración propiedades de pooling comunes como número de conexiones activas y en espera, cuánto deben esperar, lo que se tarda en expulsarlas, entre otras. Por supuesto, como ya ha sido comentado, esta configuración puede ser especificada para cada uno de los entornos del proyecto, ya sea desarrollo, producción, test...

2.8.3 Integración de Flyway

Mediante Flyway, se pueden gestionar las migraciones de las bases de datos de un proyecto.

Para integrarlo al proyecto, habrá que incluir algunas dependencias al pom.xml de Maven. Spring Boot se encargará de detectar y autoconfigurar las librerías necesarias.

A continuación, hay que asegurar que Spring Boot tiene disponible el DataSource para Flyway. Spring Boot, normalmente, utilizará el DataSource por defecto que se establezca, pero también se dispone de la opción de un DataSource alternativo si se decide no utilizar el DataSource primario por defecto.

Una vez establecidas las dependencias y el DataSource, se puede comenzar a crear scripts de migración para crear y modificar la estructura de la base de datos.

Al iniciar la aplicación, Flyway ejecutará cualquier migración contra la base de datos. Flyway se integra con el core de las aplicaciones Spring Boot, por tanto, las migraciones son simplemente ejecutadas y no será necesario realizar esta tarea en un paso separado.

Tras este paso, se puede utilizar un modelo para basarse en algo ya creado para crear la primera migración de la base de datos, o simplemente utilizar una definición de la misma, en este caso, se ha utilizado un script, que se debe ubicar en la carpeta de migraciones.

Tras completar estos pasos, Flyway estará prácticamente integrado y el proyecto cuenta con un script de migración que puede ser ejecutado en el startup para crear una tabla de la base de datos.

Spring Boot configurará Flyway con el DataSource proporcionado. De este modo se puede crear migraciones para las aplicaciones. Los cambios creados se podrán comprobar desde la consola de H2.

2.8.4 Configuración de la persistencia con Java

Para realizar la configuración que permite administrar la parte de gestión de datos, se puede utilizar la configuración basada en clases Java y anotaciones de Spring. Para ello, es recomendable crear un paquete con las clases de configuración Java y utilizar sufijos para el nombre de las clases que indique que son de configuración.

```
@Bean
@ConfigurationProperties(prefix="datasource.flyway")
@FlywayDataSource
public DataSource flywayDataSource() {
    return (DataSource) DataSourceBuilder.create().build();
}
```

Figura 2.8.4: Bean de flyway de configuración de persistencia

El primer método incluido en la clase de configuración es el de la imagen superior. La anotación **@ConfigurationProperties** indica al DataSourceBuilder que utilice la conexión y propiedades de pooling ubicadas en el archivo application.properties.

Se pueden mantener todas las propiedades de definición de conexión existentes que han sido usadas con el setup de la autoconfiguración y el Bean del DataSource de Spring se encargará de reutilizarlas.

En caso de necesitar añadir otro DataSource a la aplicación, Spring necesitará saber cuál es el que debe utilizar por defecto si no se ha definido. Esto se logra simplemente añadiendo una anotación **@Primary** a nivel del método que genera el Bean de DataSource.

Ahora, tras seguir estos pasos, la aplicación debería contar con los DataSources definidos.

2.8.5 JPA y Spring Data

Para terminar de conectar los componentes relativos a la persistencia de una aplicación, se puede utilizar JPA y Spring Data.

Para convertir una clase de datos, como un modelo, en una entidad JPA, simplemente habrá que añadir, a nivel de clase la anotación **@Entity**, que indica a Spring que deberá tratarla como tal.

A continuación, para esa misma clase, hay que indicar a JPA cuál es la clave primaria. Esto se logra mediante la anotación **@Id** sobre el atributo que formará la clave primaria, a lo que se pueden añadir más anotaciones para generar valores incrementales, por ejemplo.

También, con JPA se pueden usar muchas otras anotaciones, como es el caso de **@column**, que indica que un atributo formará una columna en la tabla de la base de datos, pero, en este caso, se utiliza la base de datos de los puntos anteriores, por lo que no es necesario.

A continuación, al proyecto se le añade un repositorio. Para ello, primero se crea una interfaz que extienda la clase JpaRepository ya que se busca que sea un repositorio de Spring Data, pasando por parámetro la entidad y el id.

2.9 Modulo Test

En muchos proyectos, muchas veces no se le da la importancia que se debería dar a los test, en algunos casos por falta de tiempo o por no considerarlos tan

importantes como en realidad son. Afortunadamente, Spring cuenta con un módulo de test unitarios, que se pueden integrar a un proyecto y poder facilitar y agilizar así esta tarea.

Utilizando Spring Boot, se necesita incluir la dependencia de test spring-boot-starter-test. Esta única dependencia preparará la mayoría de las necesidades de pruebas del proyecto.

El primer framework que va a integrar la dependencia es JUnit. JUnit ya lleva en el mercado bastante tiempo y es uno de los frameworks para tests en Java más comunes. Al hacer test unitarios básicos, JUnit y Spring se complementan el uno al otro.

Aunque JUnit soporta assertions para ayudar a analizar los resultados de las pruebas, Spring Boot también incorpora Hamcrest. Este framework proporciona formas de coincidencia en los resultados de los test y assertions mejoradas que, combinadas con JUnit, permiten automatizar las pruebas de principio a fin.

El siguiente framework integrado es Mockito. A veces, al hacer las pruebas, el código tiene dependencias con otros objetos o código que son difíciles de aislar para un test unitario. En ese caso, Mockito puede ayudar a realizar el Mock o Stub de esos objetos y poder así continuar con las pruebas.

Por último, la dependencia integra las herramientas de test de Spring. Estas incluyen anotaciones, utilidades para pruebas y ayudas para la integración de pruebas con JUnit, Hamcrest y Mockito en un entorno de Spring.

2.9.1 Primeros pasos con pruebas

El primer paso será añadir al pom.xml la dependencia spring-boot-starter-test, del mismo modo que se ha realizado con dependencias anteriores. Como esta dependencia ya añade JUnit, se puede retirar la que añade Spring al proyecto por defecto de este framework de pruebas.

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

Figura 2.9.1: dependencia de spring starter test

Como se aprecia en la imagen anterior, se ha añadido el scope de test. Esto significa que cuando la aplicación es preparada para el despliegue cualquier dependencia declarada con el scope de test será ignorada, estos scopes están disponibles únicamente en fase de desarrollo y modos de test de Maven.

Para crear el primer test, se puede reutilizar la clase AppTest incluida en la creación del proyecto. Tras limpiarla de métodos y comentarios que no se vayan a utilizar, se puede retirar también la herencia incluida, ya que ahora se pueden utilizar las anotaciones **@Test** para marcar cada uno de los métodos como pruebas unitarias.

```
public class AppTest {  
  
    @Test  
    public void testApp()  
    {  
        HomeController hc=new HomeController();  
        String result=hc.home();  
        assertEquals(result, "Das Boot, reporting for duty!");  
    }  
}
```

Figura 2.9.2: Ejemplo de test unitario

En el método creado en la imagen anterior, se instancia un objeto, se obtiene el resultado de este objeto y, por último, mediante una assertion, se comprueba que el resultado es el esperado.

Para ejecutar los test, simplemente se seleccionan y se ejecutan como test de JUnit. Tras ello, se mostrará una lista de test del proyecto que, al finalizar ejecución, se indicará mediante una barra de color verde o rojo si han pasado o no los test, respectivamente. Además, se pueden ejecutar mediante el plugin de Maven y la consola de comandos.

2.9.2 Mockito

La prueba anterior era bastante simple, al sólo devolver texto, era bastante fácil de probar. Pero, cuando las aplicaciones aumentan de complejidad, no se puede testear de forma unitaria de este modo porque el código tendrá dependencias en otro código manejado por el framework o es difícil de construir mediante una prueba. En este caso, se puede utilizar Mockito para realizar las pruebas.

```
public class ShipwreckControllerTest {  
  
    @Test  
    public void testShipwreckGet() {  
        ShipwreckController sc = new ShipwreckController();  
        Shipwreck wreck = sc.get(1L);  
        assertEquals(1L, wreck.getId().longValue());  
    }  
}
```

Figura 2.9.3: Ejemplo de test con dependencias

En la prueba de la imagen anterior, se obtiene un objeto mediante su id y, a continuación, se comprueba que el resultado del id es el esperado.

El problema será que, si en el controlador se está utilizando un repositorio, como es común, Spring no lo instanciará mediante la inyección de dependencias al estar ejecutando un test, y este devolverá una excepción de objeto nulo. Afortunadamente, para estos casos se utilizan los framework de mocking.

Utilizando Mockito se puede realizar un mock del repositorio para obtener un test útil para probar este caso. En el test, en lugar de instanciar un objeto de la clase

real, se crea una instancia mock de este. Mockito proporciona una anotación que crea este objeto y la inyecta en el test.

Una vez hecho esto, será necesario crear un mock del repositorio para que lo pueda utilizar este mock del controlador, para ello se puede utilizar otra anotación de Mockito.

Estos dos objetos ahora son manejados por el framework Mockito. Para que Mockito pueda preparar el mock del repositorio automáticamente en el controlador, hay que añadir un método que inicialice todos los mock de los objetos. Este método se marca con una anotación que indica que se debe ejecutar antes de iniciar los test.

Dentro de este método se añade una llamada a otro método estático de Mockito que se encarga de buscar las anotaciones **@InjectMocks** y **@Mocks** y comprueba si deben ser relacionadas.

Por último, queda especificar el comportamiento del mock en el repositorio cuando se ejecuta el test. Como no se accede realmente a la base de datos, habrá que especificar un comportamiento en el test. Esta acción se puede realizar gracias a otro método estático proporcionado por mockito.

Esto permite probar el código del controlador en un test unitario real, sin tener que depender de la base de datos o Spring para conseguir que el test pase.

```
public class ShipwreckControllerTest {
    @InjectMocks
    private ShipwreckController sc;

    @Mock
    private ShipwreckRepository shipwreckRepository;

    @Before
    public void init() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testShipwreckGet() {
        Shipwreck sw=new Shipwreck();
        sw.setId(11);
        when(shipwreckRepository.findOne(11)).thenReturn(sw);

        Shipwreck wreck = sc.get(11);
        assertEquals(11, wreck.getId().longValue());
    }
}
```

Figura 2.9.4: Resultado final del test

Además, Mockito ofrece más funcionalidades, como comprobar el número de veces que se han realizado llamadas a un método. De esta forma, se puede mostrar el comportamiento de una aplicación, y corregirlo si es necesario.

2.9.3 Hamcrest

Aunque se pueden utilizar las assertions que ofrece JUnit, Hamcrest presenta una versión más declarativa y fácil de leer de estas comprobaciones de resultados.

Normalmente, una vez que se ha pasado tiempo realizando test, con variedad de casos de test y resultados amplios, muchos desarrolladores comienzan a preferir la sintaxis de Hamcrest sobre otros métodos de assertions.

```
assertEquals(11, wreck.getId().longValue());    assertThat(wreck.getId(), is(11));
```

Figura 2.9.6: assertion con JUnit

Figura 2.9.5: assertion con Hamcrest

De esta forma, sin tener que cambiar el código del test, se pueden intercambiar entre sí las líneas de assertion de las figuras anteriores, ya que son lógicamente equivalentes. Sin embargo, el método de Hamcrest es más cómodo para su lectura y se puede entender fácilmente su función.

Aunque este ejemplo es una simple comparación, Hamcrest ofrece una gran variedad de opciones y conectores. Para poder conocerlos, se puede consultar su documentación disponible para cualquier desarrollador que necesite utilizarlos.

2.9.4 Test de integración

Un test de integración trata de probar todas las piezas de una aplicación trabajando juntas como lo harían en un entorno real o de producción. Esto significa que la aplicación debe estar en ejecución para probarla.

Por la naturaleza de un test de integración, traen unos retos a la hora de construirlos y ejecutarlos. Antes de Spring Boot, algunos de los retos que las aplicaciones Spring encontraban eran los siguientes.

Primero, los Containers. Cualquier código, parte de la lógica de la aplicación, que utiliza el container o el servlet de especificación es difícil de probar porque, o se necesita probar el startup del container y ejecutar los test contra él, o se necesita realizar un mock del container y emularlo de algún modo.

Como el Spring Core, los Spring Beans y la inyección de dependencias requieren que Spring se esté ejecutando y manejando esas piezas en el contexto de la aplicación, todos los test de integración necesitan asegurar que el contexto de la aplicación Spring se está ejecutando.

Arrancar el contexto de Spring o ejecutar o emular el container, puede tomar mucho tiempo en aplicaciones más grandes. Los test de integración, naturalmente, se ejecutan más despacio que los test unitarios normales. Es lógico pensar que según se añaden más y más test de integración, el tiempo para ejecutar todas las pruebas puede incrementar mucho.

Por último, si los test de integración modifican la base de datos o esperan ciertos datos en esta, se puede llegar a tener problemas si no se consigue que la base de datos se mantenga consistente cada vez que se ejecutan los test.

Ahora, la forma de trabajar de Spring Boot con estos problemas es la siguiente.

Como las aplicaciones de Spring Boot puede ejecutarse como una aplicación Java más, la complejidad de tratar con un container y desplegar la aplicación desaparece. Spring Boot mantiene un container embebido, pero Spring Boot simplemente hace arrancar y tratar la aplicación mucho más fácilmente.

Los test de integración en Spring Boot, siguen necesitando un contexto de Spring. La diferencia principal entre Spring Boot y las aplicaciones de Spring es el uso de inicializadores y la autoconfiguración. Esto hace que preparar el contenedor de Spring con Spring Boot sea algo más fácil.

El tiempo requerido para el startup de los test de integración y su ejecución siguen siendo problemas en el entorno de Spring Boot. Cuanto más grande sea la aplicación y más componentes Spring tenga, más tiempo tardará en arrancar.

La consistencia en bases de datos sigue siendo también un problema con los test en Spring Boot.

Pese a todos estos retos, los test de integración son una de las mejores formas para probar que una aplicación completa trabaja como se esperaba.

Para convertir cualquier prueba de JUnit en un test de integración, sólo hay que añadir dos cambios básicos. El primero, es una anotación, **@RunWith**, con la que se marcan los test. El siguiente es añadir la anotación **@SpringApplicationConfiguration** y proporcionar la clase principal de la aplicación de Spring Boot, ambas incluidas en la figura siguiente.

La primera anotación indica que el test de integración debería ejecutarse con el `SpringJUnit4ClassRunner`. Este elemento para ejecutar clases de test, basado en Spring, es parte de las herramientas de test de Spring que el inicializador de test incluye de forma transitiva en el proyecto.

La segunda anotación pertenece a Spring Boot y se le proporciona la clase principal como parámetro de la anotación. Esto indica a Spring Boot cómo configurar y arrancar la aplicación. Esto es parecido a llamar al método estático principal al arrancar la aplicación, pero embebiéndolo dentro del contexto de un test.

```
17 @RunWith(SpringJUnit4ClassRunner.class)
18 @SpringApplicationConfiguration(App.class)
19 public class ShipwreckRepositoryIntegrationTest {
20
21     @Autowired
22     private ShipwreckRepository shipwreckRepository;
23
24     @Test
25     public void testFindAll() {
26         List<Shipwreck> wrecks = shipwreckRepository.findAll();
27         assertThat(wrecks.size(), is(greaterThanOrEqualTo(0)));
28     }
29
30 }
```

Figura 2.9.7: clase de pruebas de integración con métodos implementados

Para inyectar el repositorio en el ejemplo anterior, al tratarse de un test de integración, sólo hay que añadir un autowire del atributo, como muestra la imagen anterior, así cuando el test se inicie, el contexto de Spring se cargará y Spring inyectará el repositorio en el test, tal como haría si se estuviese ejecutando en una aplicación estándar.

Además, Spring ofrece la anotación **@WebIntegrationTest**, útil para probar aplicaciones que utilizan API mediante una simple anotación. Se utiliza como una anotación de clase y los test se realizan de la misma forma que los anteriores, con la diferencia de la anotación añadida.

3 Aplicación práctica

3.1 Descripción de la aplicación

Mediante el desarrollo de una aplicación, se busca comprender, en mayor profundidad, y gracias a los conocimientos teóricos expuestos, el funcionamiento de Spring y cómo aplicarlo a un ejemplo práctico algo más complejo.

La aplicación, statGambler, se trata de una aplicación web que tratará de aconsejar al usuario la mejor jugada posible en distintos juegos de azar, basándose en cálculos estadísticos y la recolección de datos. Además, utilizando estos datos, en aquellos juegos que sea posible, también se aconsejará la cantidad óptima de apuesta.

La aplicación permitirá guardar datos de alguno de los juegos disponibles, asociando estos datos a un solo usuario, con la posibilidad de compartirlos de forma pública o a un grupo concreto de usuarios.

Además, los distintos cálculos podrán ser compartidos entre usuarios, para que puedan realizar estudios conjuntos, pudiendo recolectar datos entre varios usuarios.

En un principio, la aplicación se centrará en un solo juego, desarrollándose de manera modular para que se puedan añadir todos aquellos juegos que se requieran en un posible futuro.

La interfaz constará de una pantalla de login y otra de creación de usuarios para poder gestionar el registro de nuevos usuarios y la entrada al sistema. Además, se incluirá una vista de bienvenida, en la que se explica brevemente cada uno de los juegos incluidos.

Por otro lado, para cada uno de los juegos, se incluye una pantalla de estadísticas, en la que se mostrarán cada una de las estadísticas calculadas para cada juego y una pantalla de muestra de datos utilizados para los cálculos y una más para la edición o añadido de datos.

Esta última pantalla de edición, creado y borrado de datos estará disponible para el usuario sólo en el caso de la ruleta, al ser un juego a escala local y tener cada usuario sus datos propios. En el caso de la Primitiva y el Euromillones, estas vistas sólo se mostrarán para los administradores, al ser datos compartidos para todos los usuarios.

3.2 Análisis de requisitos y casos de uso

En este apartado se indican las principales especificaciones del software en función de los distintos objetivos que tiene con el fin de darles solución.

Estos son los requisitos que debe cumplir la aplicación:

- La aplicación deberá mostrar la esperanza matemática, el número de apariciones de cada resultado, su promedio y probabilidad de obtener cada uno de los premios de cada juego al usuario.
- La aplicación debe mostrar, para cada usuario, sus estadísticas personales, siendo estas el dinero ganado, el dinero gastado, beneficios, número de veces que ha jugado y beneficio medio por jugada.
- La aplicación deberá permitir iniciar sesión a los usuarios de forma segura.
- La aplicación permitirá a los usuarios añadir y actualizar sus estadísticas personales para cada uno de los juegos.
- La aplicación podrá mostrar los resultados almacenados para cada juego.
- La aplicación deberá permitir, sólo a los administradores, actualizar los datos de primitivas y euromillones.
- La aplicación permitirá añadir y actualizar los resultados de la ruleta de cada uno de los usuarios.
- La aplicación permitirá crear nuevos usuarios.

3.2.1 Diagrama de casos de uso

Mediante el siguiente diagrama de casos de uso, se pretende mostrar las acciones que puede realizar el usuario con el uso de esta aplicación.

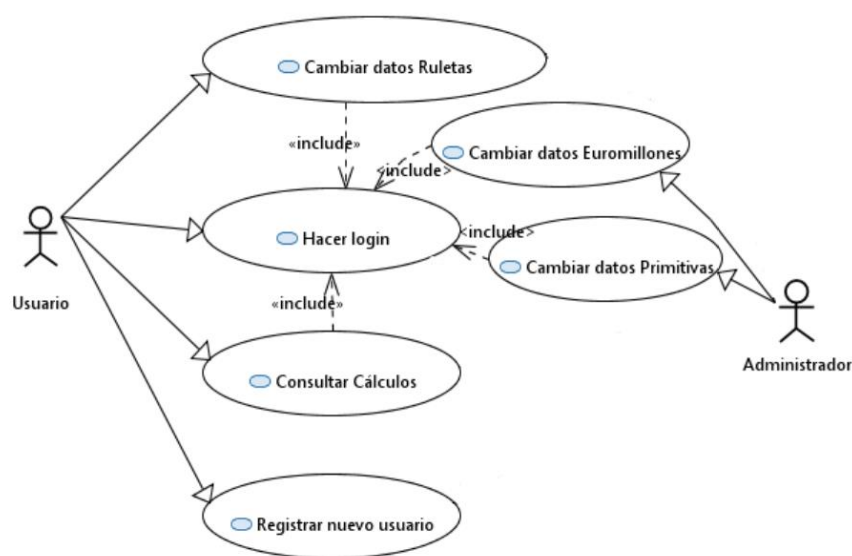


Figura 3.2.1: Diagrama de casos de uso

La descripción de los casos de uso viene detallada a continuación:

Nombre	Registrar nuevo usuario
Autor	Fernando Barcala



Actores	Usuario
Descripción	
El usuario podrá hacerse una cuenta al iniciar la aplicación. De esta forma, podrá guardar sus datos en su cuenta personal y comenzar a realizar cálculos con ellos. En esta acción, el usuario debe proporcionar sus datos para que la aplicación le pueda registrar en su base de datos.	
Precondiciones	
Flujo Normal	
<ol style="list-style-type: none">1. El usuario accede a la aplicación2. El usuario accede a la pantalla de creación de usuario3. El usuario ingresa sus datos y completa su registro	
Flujo Alternativo	
<ol style="list-style-type: none">1. El usuario accede a la aplicación2. El usuario accede a la pantalla de creación de usuario3. El usuario introduce datos incorrectos4. Se muestra la misma pantalla de registro	
Postcondiciones	
Se guarda tanto los datos del usuario como la sesión del usuario.	

Nombre	Hacer login
Autor	Fernando Barcala
Actores	Usuario
Descripción	
Para poder acceder a la aplicación, el usuario deberá hacer login. Para comprobar que el usuario está registrado, este deberá proporcionar su nombre de usuario y su contraseña.	
Precondiciones	
Flujo Normal	
<ol style="list-style-type: none">1. El usuario accede a la aplicación2. El usuario accede a la pantalla de login3. El usuario ingresa sus datos y accede	



Flujo Alternativo
<ol style="list-style-type: none">1. El usuario accede a la aplicación2. El usuario accede a la pantalla de login3. El usuario introduce datos incorrectos4. Se muestra la misma pantalla de login
Postcondiciones
Se guarda tanto los datos del usuario como la sesión del usuario.

Nombre	Consultar cálculos
Autor	Fernando Barcala
Actores	Usuario
Descripción	
El usuario podrá consultar las estadísticas, tanto personales, como del juego que está visualizando.	
Precondiciones	
Haber hecho login	
Flujo Normal	
<ol style="list-style-type: none">1. Acceder a la pantalla de estadísticas del juego	
Flujo Alternativo	
Postcondiciones	

Nombre	Cambiar datos Ruletas
Autor	Fernando Barcala
Actores	Usuario
Descripción	
El actor, a través de esta acción, podrá acceder a la pantalla de edición de datos de ruletas, donde podrá ver, añadir, editar y borrar los datos relacionados con su cuenta.	
Precondiciones	
Haber hecho login en la aplicación	
Flujo Normal	



<ol style="list-style-type: none"> 1. El actor accede a la pantalla de estadísticas 2. El actor accede a la pantalla de visualización de datos 3. El usuario realiza una operación CRUD sobre los datos <ol style="list-style-type: none"> 4. Se validan los datos introducidos 5. Se completa la operación
Flujo Alternativo
<ol style="list-style-type: none"> 1. El actor accede a la pantalla de estadísticas 2. El actor accede a la pantalla de visualización de datos 3. El usuario realiza una operación errónea CRUD sobre los datos <ol style="list-style-type: none"> 4. Se muestra la misma pantalla
Postcondiciones
Se guardan los cambios en la base de datos

Nombre	Cambiar datos Euromillones
Autor	Fernando Barcala
Actores	Administrador
Descripción	
El actor, a través de esta acción, podrá acceder a la pantalla de edición de datos de euromillones, donde podrá ver, añadir, editar y borrar los resultados que hayan aparecido.	
Precondiciones	
Haber hecho login en la aplicación como administrador	
Flujo Normal	
<ol style="list-style-type: none"> 1. El actor accede a la pantalla de estadísticas 2. El actor accede a la pantalla de visualización de datos 3. El usuario realiza una operación CRUD sobre los datos <ol style="list-style-type: none"> 4. Se validan los datos introducidos 5. Se completa la operación 	
Flujo Alternativo	
<ol style="list-style-type: none"> 1. El actor accede a la pantalla de estadísticas 2. El actor accede a la pantalla de visualización de datos 3. El usuario realiza una operación errónea CRUD sobre los datos <ol style="list-style-type: none"> 4. Se muestra la misma pantalla 	
Postcondiciones	

Se guardan los cambios en la base de datos

Nombre	Cambiar datos Primitivas
Autor	Fernando Barcala
Actores	Administrador
Descripción	
El actor, a través de esta acción, podrá acceder a la pantalla de edición de datos de primitivas, donde podrá ver, añadir, editar y borrar los resultados.	
Precondiciones	
Haber hecho login en la aplicación como administrador	
Flujo Normal	
<ol style="list-style-type: none"> 1. El actor accede a la pantalla de estadísticas 2. El actor accede a la pantalla de visualización de datos 3. El usuario realiza una operación CRUD sobre los datos <ol style="list-style-type: none"> 4. Se validan los datos introducidos 5. Se completa la operación 	
Flujo Alternativo	
<ol style="list-style-type: none"> 1. El actor accede a la pantalla de estadísticas 2. El actor accede a la pantalla de visualización de datos 3. El usuario realiza una operación errónea CRUD sobre los datos <ol style="list-style-type: none"> 4. Se muestra la misma pantalla 	
Postcondiciones	
Se guardan los cambios en la base de datos	

Cabe aclarar que este diagrama actuará como una guía, pudiendo sufrir modificaciones durante el proyecto.

3.3 Arquitectura de la aplicación

Para el desarrollo de la aplicación se ha decidido seguir un diseño MVC ya que, al ser una aplicación web, es el que mejor encaja, además, trae consigo una gran mantenibilidad y claridad de código.

Con el apoyo de Spring, se intentará organizar el código siguiendo dicha arquitectura, añadiendo una capa de datos y otra de servicios, de forma que la aplicación quede bien organizada.

Siguiendo esta idea, la estructura del proyecto quedará definida con los siguientes elementos:

- **Capa de presentación:** Será, en definitiva, la interfaz de la aplicación, ofreciendo al usuario vistas usables y sencillas, que permitan realizar las funciones de la aplicación de manera intuitiva.
- **Capa de negocio:** contendrá los controladores y servicios que doten a la aplicación de funcionalidad. Gracias a los controladores, la aplicación podrá recibir y atender peticiones del usuario, y, además, los servicios se encargarán de realizar las funciones correspondientes, sirviendo como apoyo a los controladores.
- **Capa de datos:** En esta capa estarán contenidos los repositorios y modelos, por un lado, los modelos serán las clases que definan los objetos a guardar en la base de datos. Por otro, los repositorios, se encargarán de realizar las conexiones con la base de datos y sus correspondientes operaciones de actualización, guardado, borrado y lectura de objetos.

3.3.1 Capa presentación

La base de esta capa será el código html implementado para cada una de las vistas implementadas, sobre estas vistas, se han utilizado, junto al framework Spring, y MVC, distintos elementos que ayudarán a simplificar las tareas de desarrollo de las vistas web. Estos elementos son:

Thymeleaf, que ayuda a la creación de vistas web, dotando al código html de distintas funcionalidades ya implementadas, como la iteración en listas o la comprobación de la autenticación para mostrar ciertos elementos. Además, permite implementar partes de vistas compartidas, de forma que sólo haya que codificarlas una vez.

Con Bootstrap, se han conseguido implementar vistas más agradables para el usuario de manera sencilla y rápida, gracias a la cantidad de clases que trae consigo implementado Bootstrap.

3.3.2 Capa de negocio

En esta capa se implementa la funcionalidad real de la aplicación, con la que se redirigirá al usuario según la petición que haya realizado, realizando antes las tareas correspondientes.

En esta capa de negocio, se implementan los controladores de la aplicación, que serán los encargados de recibir la petición establecida, y realizar la llamada correspondiente al servicio, en caso de que sea necesario.

Una vez el servicio haya devuelto la respuesta correspondiente, se guardará este resultado en el modelo, para que, de esta forma, la vista pueda mostrar los datos que hayan sido calculados.

Además, los servicios se encargarán de ofrecer distintas funcionalidades para evitar duplicidades de código y permitir que los controladores tengan un código más limpio y reducido.

Algunas de estas funciones incluyen el cálculo de estadísticas de cada uno de los juegos, el tratamiento del usuario registrado o la validación de datos.

El cálculo de datos se realiza en clases exclusivas para cada juego, de forma que cada uno tiene unos cálculos adaptados para ese juego en concreto. Además, se realizan conversiones a string y redondeos, de forma que la vista se muestre lo más limpia posible.

El tratamiento del usuario que ha sido registrado realiza a través de Spring Security y de un request Http, de esta forma y gracias a la inyección de dependencias, se puede recuperar, en cualquier punto del código, los datos del usuario, como su nombre, roles u otros datos.

La validación, por último, se realiza tanto para juegos como para usuarios, y trata de comprobar que el usuario no ha introducido errores al intentar introducir datos nuevos a la base de datos. De esta forma, se evitan posibles errores, tanto de datos como de código, al limitar al usuario los datos que puede introducir.

En esta capa, además, se utiliza tomcat para que la aplicación pueda disponer de un servidor. Esta tecnología permite utilizar de manera sencilla un servidor que de las funcionalidades necesarias a la aplicación para poder ejecutarse sin problemas y sin tener que desarrollar código. Simplemente se tendrán que definir unas pocas configuraciones para que pueda funcionar completamente.

Por supuesto, también se utilizará Spring, que será el que se encargue de conectar la mayoría de los componentes de la aplicación, además de utilizar el container para poder utilizar el autowire en las distintas clases de la aplicación.

3.3.3 Capa de datos

La capa de datos será la encargada de dotar a la aplicación de la capacidad de almacenar datos, de forma que el usuario pueda realizar distintas acciones y poder guardar estos resultados en la base de datos. Esta capa estará compuesta de los repositorios y los modelos.

Los repositorios heredan de CrudRepository, una interfaz de Spring data. Gracias a esta interfaz, Spring podrá realizar todas las operaciones necesarias para que la aplicación pueda funcionar correctamente y almacenar todos los datos necesarios. Simplemente habrá que indicar a la interfaz el Modelo del que se debe encargar el repositorio y el tipo de datos del identificador del modelo, Long en este caso.

Por otro lado, los modelos serán los encargados de definir los datos de cada una de las tablas de la aplicación. Estos modelos, son marcados con la anotación **@Entity**, de forma que Spring puede identificarlos junto a los repositorios para su posterior manipulación como entidades de la base de datos.

Estos modelos, serán POJOs, ya que simplemente definirán su id, marcado con la anotación correspondiente y sus atributos, además de sus getters y setters.

3.3.4 Estructura final

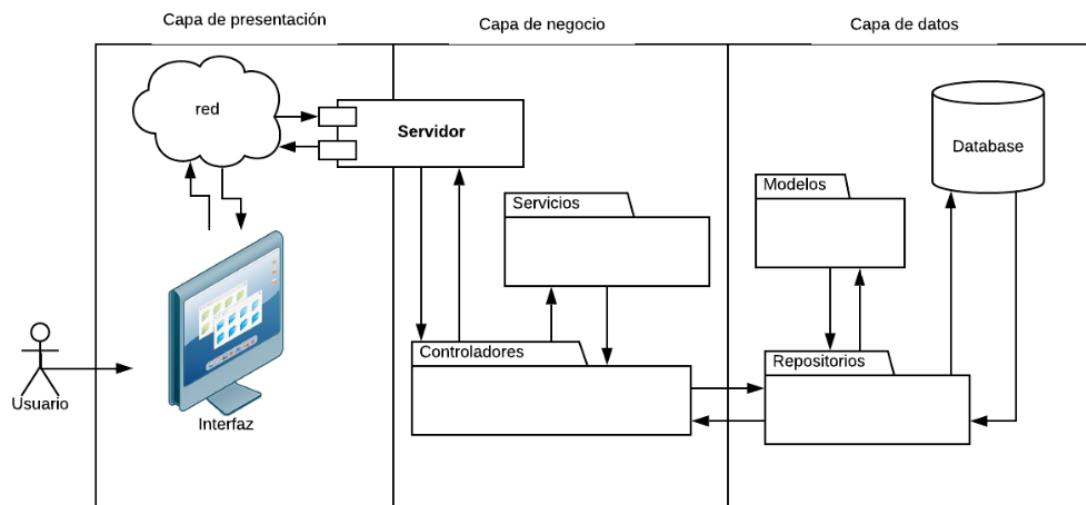


Figura 3.3.1: Estructura final

El resultado final está mostrado en la figura anterior. En esta figura se muestran cada una de las capas incluidas en la arquitectura, mostrando, para cada una, sus componentes y las conexiones entre estos. De esta forma queda ilustrado cómo se comunican los componentes en la aplicación.

De esta forma, se busca dotar a la aplicación de mantenibilidad y claridad en el código. A través de estas características, en definitiva, lo que se busca es obtener una aplicación de calidad, que dé al usuario una experiencia agradable y facilite el trabajo del desarrollador.

3.4 Diseño de la interfaz

A través de una interfaz sencilla e intuitiva, se espera que el usuario sea capaz de utilizar la aplicación sin dificultades.

Para ello, se han diseñado las siguientes vistas, como prototipo de las finales.

Las vistas implementadas para cada uno de los juegos son utilizadas para añadir, mostrar, y actualizar los juegos, además de una vista más para mostrar las estadísticas de cada juego, basadas en los datos introducidos.

Además, se añade una vista index, que ofrece una breve descripción de cada uno de los juegos y una vista común a todas, que implementa una barra de navegación y añade los scripts necesarios, mostradas ambas en la siguiente figura.

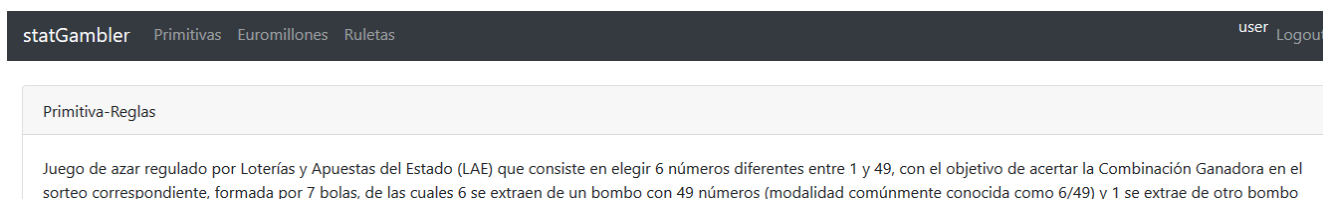


Figura 3.4.1: vista index y común



La vista de estadísticas de cada juego consta de dos partes principales, la primera muestra las estadísticas personales de cada usuario, dependiendo del juego que esté visualizando, la segunda parte, consta de unas estadísticas relativas al juego correspondiente, teniendo en cuenta los datos guardados.

Navbar Primitivas Euromillones Ruletas						
Datos generales de Euromillones						
Media de Resultados	Media de Estrellas	Bote	Esperanza			
18.333333333333332	4.0	1000.0	0.0			
probabilidad5y2	probabilidad5y1	probabilidad5	probabilidad4y2	probabilidad4y1	probabilidad4	
1.311039371036728E-8	1.835455119451419E-7	2.753182679177129E-7	2.949838584832638E-6	4.129774018765693E-5	6.19466102814854E-5	
probabilidad3y2	probabilidad3y1	probabilidad3	probabilidad2y2	probabilidad2y1	probabilidad2	
1.2979289773263608E-4	0.001817100568256905	0.0027256508523853576	0.001860364867501117	0.02604510814501564	0.03906766221752346	

Estadísticas de Euromillones					
numero	Promedio numeros	Apariciones numeros	numero	Promedio numeros	Apariciones numeros
1	0.0	0	24	33.33333333333333	2
2	0.0	0	25	33.33333333333333	2

Figura 3.4.2: vista de estadísticas del juego

statGambler Primitivas Euromillones Ruletas					user Logout						
Estadísticas personales - Primitivas											
Dinero Gastado	Dinero Ganado	Beneficio	Apuestas Jugadas	Beneficio Medio							
0.0	0.0	0.0	0	0.0							
Dinero Gastado	Dinero Ganado	Apuestas Jugadas									
<input type="text" value="0.0"/>	<input type="text" value="0.0"/>	<input type="text" value="0"/>									
<button>Actualiza estadísticas</button>											
Dinero Gastado	Dinero Ganado										
<input type="text" value="0.0"/>	<input type="text" value="0.0"/>										
<button>Nueva Apuesta</button>											

Figura 3: vista estadísticas personales

La vista que muestra los datos de los juegos se trata de una tabla que muestra cada uno de los datos del juego correspondiente. En caso de ser administrador, además, se mostrarán unos botones para poder editar, borrar y añadir resultados en los juegos de la primitiva y euromillones. En el caso de las ruletas, estos botones se muestran para todos los roles.

Euromillones									
Id	Resultado 0	Resultado 1	Resultado 2	Resultado 3	Resultado 4	Estrella 0	Estrella 1	editar	borrar
1	12	24	25	24	25	3	5	Edit	Delete

Figura 3.4.4: lista de resultados

Las vistas de editar y añadir juegos se muestran mediante el mismo formulario, en él, se muestran al usuario cada uno de los datos que debe introducir o actualizar para poder guardar un nuevo resultado o editar uno existente.

Resultado 1

Resultado 2

Resultado 3

Resultado 4

Estrella 0

Estrella 1

Figura 3.4.5: formulario de datos

Por último, la vista de login y creación de usuarios consta de un simple

Login

Username

Nombre

Contraseña

Contraseña

Login

No tienes cuenta?[Click aqui](#)

Nuevo usuario

Nombre

Nombre

Username

Username

Contraseña

Password

Repetir Contraseña

PasswordConfirm

Nuevo Usuario

formulario en el que el usuario deberá introducir los datos que se requieran para poder entrar a la aplicación o registrar un nuevo usuario.

Figura 6: vista de login y nuevo usuario

Al acceder a la aplicación, al usuario se le presentará la vista de login, para que pueda ingresar sus datos, a través de esta vista podrá también acceder a la creación de usuarios.

Una vez haya ingresado con un usuario válido, se le llevará a la vista de índice de la aplicación, y a través de la barra de navegación podrá acceder a cada una de las estadísticas de la aplicación.

Una vez se encuentre visualizando las estadísticas de uno de los juegos, podrá acceder a sus datos y, si su rol lo permite, a desde ahí podrá acceder a la vista de añadido y edición de datos.

3.5 Diagrama de clases

Mediante el diagrama de clases, se busca mostrar la relación a implementar entre las clases de la aplicación, además del tipo de arquitectura que se ha decidido utilizar.

En este caso la arquitectura elegida es MVC, que es la que mejor parece encajar con el tipo de aplicación. Gracias a esta arquitectura, se podrá realizar un cambio más sencillo entre vistas, y una separación en ámbitos más clara, pudiendo separar fácilmente los datos de las vistas.

Se ha añadido en este caso además una capa de servicios, para separar las operaciones y cálculos de cada uno de los juegos de las acciones del controlador, buscando que el código quede más ordenado y simple.

Además, se ha añadido un contexto, que define, junto con los modelos, la base de datos de la aplicación y será el encargado de realizar las conexiones con esta y un repositorio, que será el que opere sobre la base de datos.

En el diagrama aparecen marcadas, por colores cada una de las carpetas que formarán la aplicación. Estas carpetas son servicios, marcada en azul, vistas, marcada en rojo, controladores, marcado en naranja y modelos, marcado en verde.

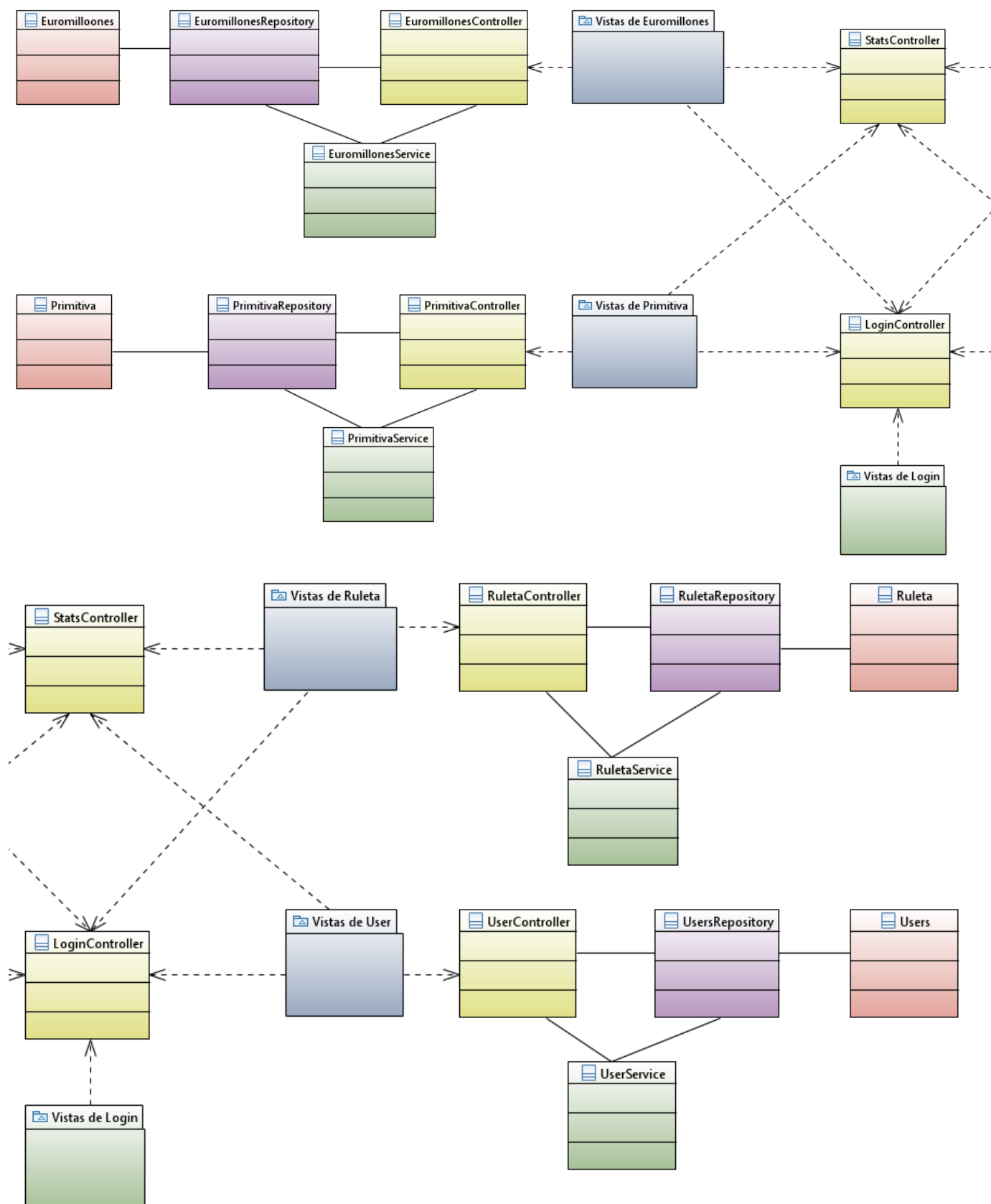


Figura 3.5.1:diagrama de clases

Para cada uno de los juegos y los usuarios se mantendrá una estructura similar, en la que se hará uso, para cada juego, de unas vistas, un controlador, un repositorio, un modelo y un servicio. Además, para el caso del login, se incluirá un controlador y unas vistas propias.

Para los juegos, se han incluido una vista de estadísticas, que muestra todas las estadísticas relacionadas con el juego en concreto, tanto personales como del juego, una vista de datos, en las que se muestran todos los datos que hay guardados para ese juego y, por último, una vista de edición y otra de añadido.

Para cada uno de los juegos y los usuarios, se ha añadido un controlador, que será el encargado de recibir las peticiones enviadas a través de las vistas y dirigirlas y procesarlas como corresponda.

También se ha añadido para este mismo grupo un modelo, que será el que define los datos que se guardarán y un repositorio, encargado de realizar las operaciones con los modelos y la base de datos.

Por último, se han añadido unos servicios, que son los encargados de realizar las operaciones sobre los datos y realizar los cálculos estadísticos con los datos guardados.

Además, el controlador de login, será el encargado de procesar las peticiones de las vistas de login y registro de usuarios. Este controlador se encargará de llamar y redirigir al usuario desde las pantallas hacia la aplicación y de realizar la llamada correspondiente al repositorio al añadir un nuevo usuario.

En todos los controladores, se hace un control de seguridad, a través de Spring Security, con el que se asegura que ningún usuario puede acceder donde no debe, y realizar, por ejemplo, cambios en resultados de primitivas o euromillones.

Además, desde los controladores, se realizarán llamadas de validación de datos para comprobar que el usuario no introduce errores al registrar datos nuevos en la aplicación.

3.6 Diagrama de actividades

Mediante el diagrama de actividades, se pretende ilustrar el flujo de ejecución típico de la aplicación, para poder comprender de forma más clara el comportamiento de la misma.

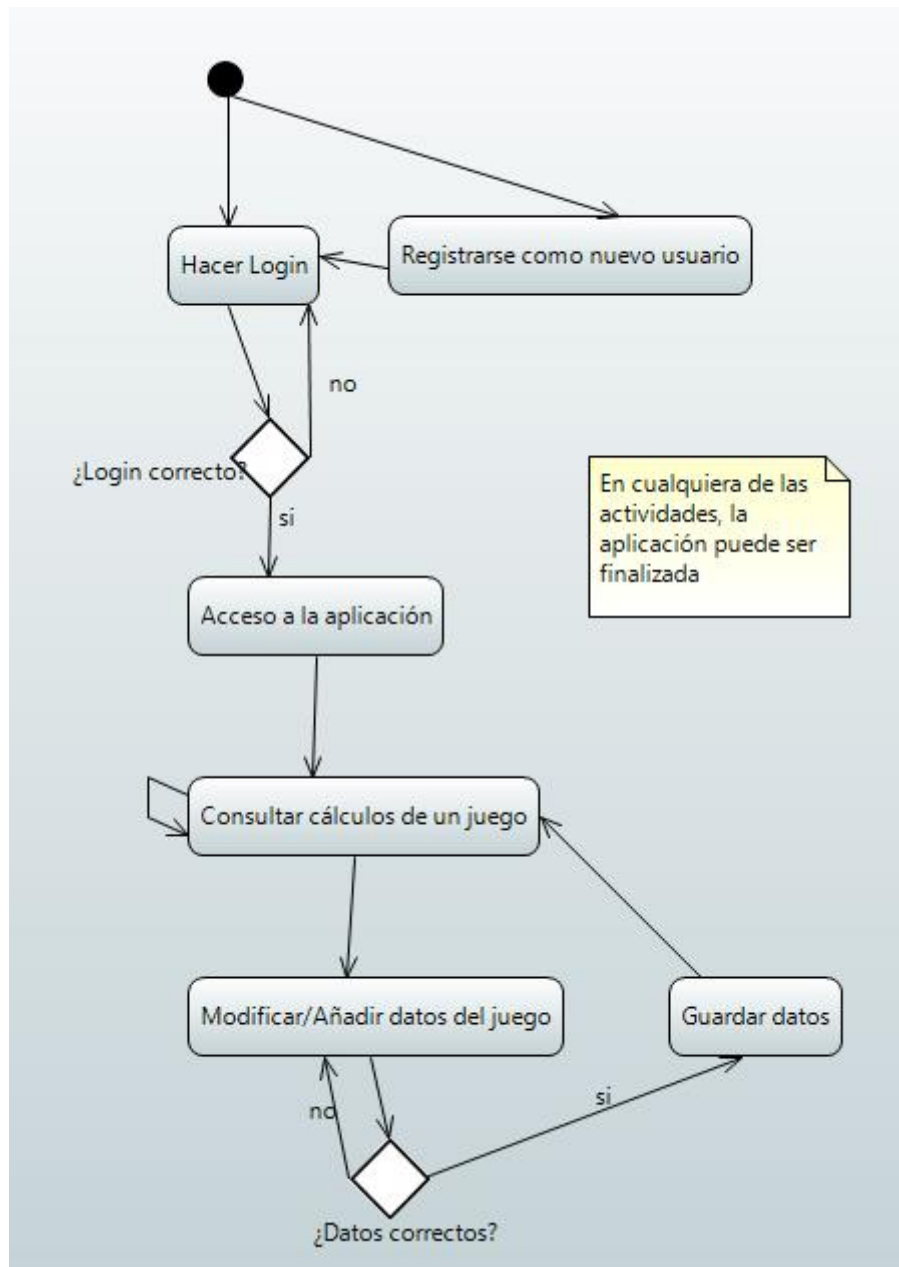


Figura 3.6.1: Diagrama de actividades

Cuando se comienza a utilizar la aplicación, el usuario cuenta con dos opciones posibles, registrarse o hacer login. Tras registrarse, en caso de no disponer de una cuenta, podrá hacer el login para acceder a la aplicación.

Si el login no es correcto se le devolverá a la pantalla de login, advirtiéndolo del error, en caso de ser correcto se le dará acceso a la aplicación y podrá comenzar a utilizarla.

Una vez dentro de la aplicación, el usuario podrá consultar tantos datos de tantos juegos como necesite. Dentro de cada juego, podrá modificar los datos con los que trabaja.

En caso de modificar o añadir datos, se comprobará si estos son correctos, en caso de no serlo, se devolverá al usuario a la modificación de datos, en caso de ser

correctos, estos datos se guardarán y se volverá a la consulta de los datos disponibles.

Por último, cabe destacar que, en cualquier momento, el usuario podrá finalizar la aplicación, sin ser necesario que siga una ruta específica para ello.

Cabe aclarar que este diagrama actuará como una guía, al igual que en los dos diagramas anteriores, pudiendo sufrir modificaciones durante el proyecto.

3.7 Diagrama de datos

A través del siguiente diagrama de la base de datos, se muestran las relaciones entre las distintas tablas de la aplicación, así como los datos que almacenan cada una de ellas.

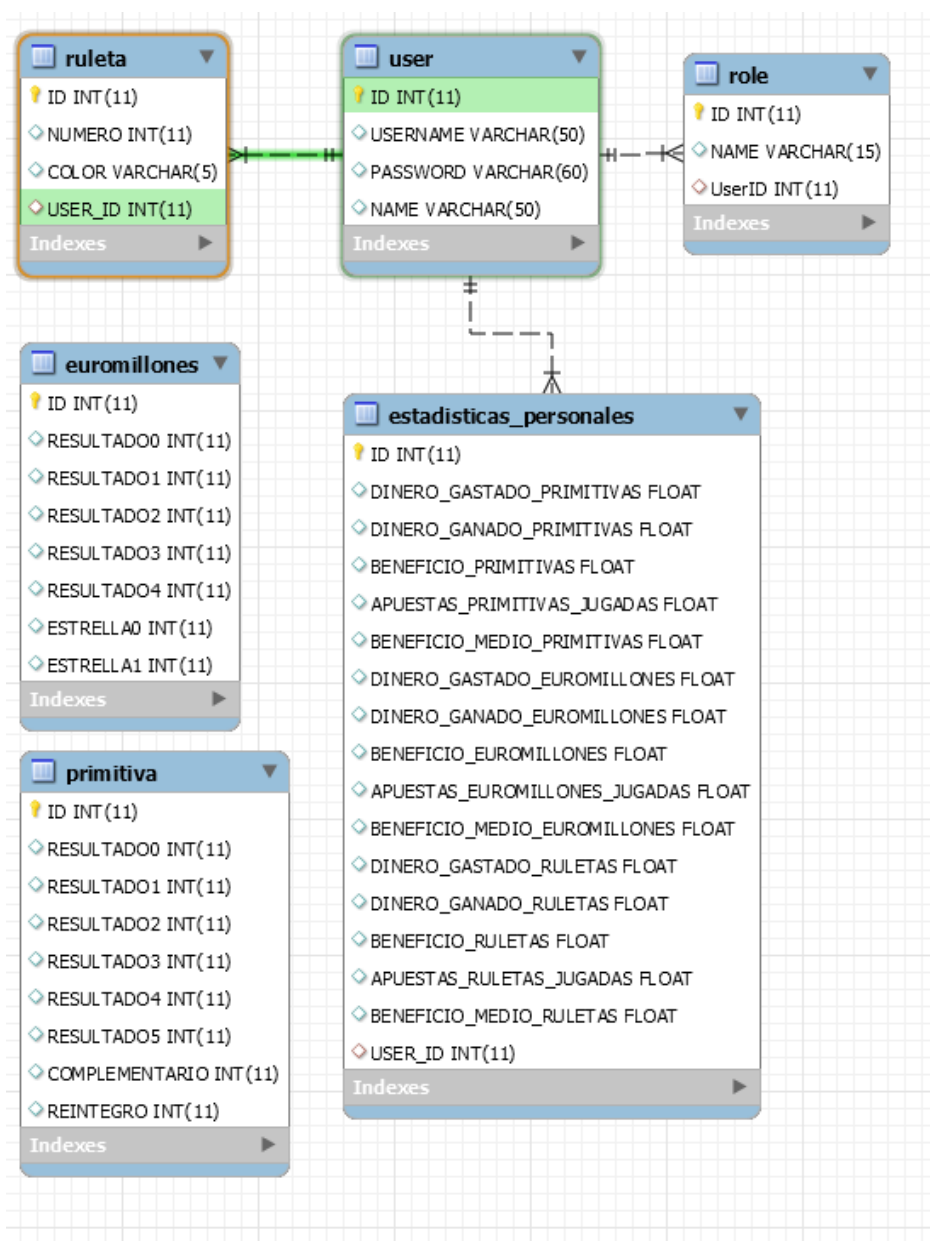


Figura 3.7.1:diagrama de base de datos

La base de datos cuenta con las siguientes tablas:

- **Euromillones:** Se encargará de almacenar todos los resultados de euromillones, constando estos de cada uno de los números extraídos en el sorteo.
- **Primitiva:** Al igual que la tabla euromillones, primitivas, se encargará de almacenar cada uno de los resultados de la primitiva, en los que se incluyen cada uno de los números extraídos.
- **Ruleta:** En la tabla ruleta, se almacenará cada uno de los números obtenidos. Cada ruleta estará relacionada con un único usuario, para que cada usuario pueda mantener unos datos de las ruletas en las que haya jugado.
- **Estadísticas personales:** Guarda cada una de las estadísticas personales para un único jugador, siendo estas características el dinero gastado, el dinero ganado, el beneficio total, el número de apuestas totales y el beneficio medio por apuesta obtenido, para cada uno de los juegos disponibles.
- **Role:** En esta tabla, se almacenan cada uno de los roles que puede tener un usuario, en este caso serán “USER” y “ADMIN”, que otorgarán al usuario privilegios distintos.
- **User:** En esta tabla, se almacenan cada uno de los usuarios de la aplicación. Para cada uno, se guarda el nombre, el nombre de usuario y la contraseña, almacenada tras la aplicación de un hash. Además, esta tabla está relacionada con uno o varios roles y con una sola fila de estadísticas personales.

Gracias a esta base de datos se podrá mantener una estructura de datos sostenible y que permita a la aplicación realizar las operaciones pertinentes.

3.8 Estructura del proyecto

Teniendo esto en cuenta y siguiendo los diseños realizados en el punto anterior, la estructura del proyecto ha quedado de la siguiente forma:

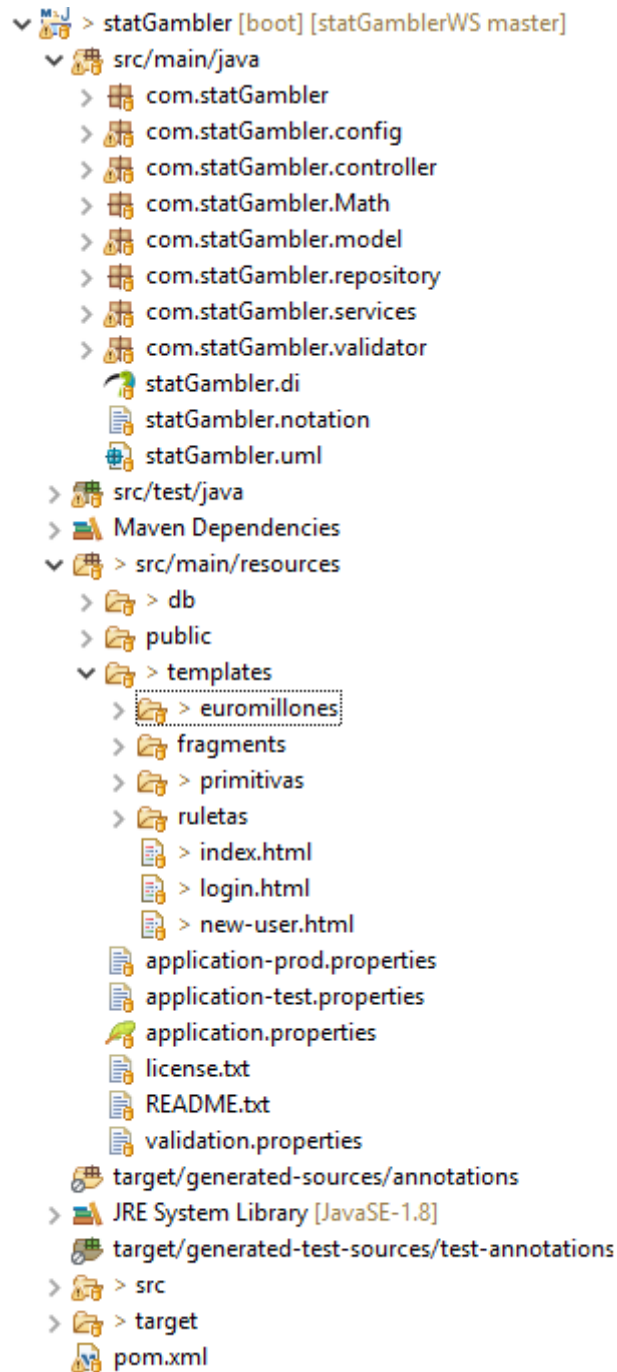


Figura 2: estructura final del proyecto

- **Main.java.com.statGambler:** contiene la clase App, que es la clase principal de la aplicación, donde se define el escáner de componentes y desde donde arrancará la aplicación para su manejo por Spring.
 - **config:** contiene la configuración necesaria para que Spring pueda hacer que la aplicación funcione como se espera. En este caso, se ha incluido una clase de configuración de seguridad, de esta forma, este apartado queda definido por el desarrollador

para que, posteriormente, Spring Security pueda utilizarla correctamente.

- **controller:** En esta carpeta se incluyen todos los controladores que necesita la aplicación, cada uno de sus métodos atenderá un post o un get para poder realizar la acción correspondiente.
- **Math:** Esta es una carpeta de apoyo, que incluye una clase que implementa algunos métodos estáticos para poder realizar distintas operaciones matemáticas para apoyo a los servicios de cada uno de los juegos implementados. De esta forma, sólo habrá que hacer una llamada a estos métodos para obtener los resultados, evitando duplicidades.
- **model:** En este paquete se incluyen las clases necesarias para definir los modelos necesarios para guardar los datos de la aplicación. Cada modelo corresponde con una tabla de la base de datos.
- **repository:** En esta carpeta se incluye, para cada uno de los modelos implementados, un repositorio. Estos repositorios se encargan de realizar todas las operaciones necesarias sobre la base de datos y realizar las conexiones con esta.
- **services:** Se trata de un paquete de apoyo a los controladores, se encargan de realizar los cálculos de las estadísticas, basándose en los datos guardados y el usuario al que correspondan.
- **validator:** Es una carpeta de validadores que comprueban, para cada juego y para la creación de usuarios, una serie de condiciones, buscando errores en los datos introducidos, en caso de haber introducido algún error, no se producirá la acción iniciada.
- **Main.resources:** En esta carpeta se incluyen las propiedades del proyecto, entre estas propiedades, se incluyen la definición del puerto a usar, las propiedades de h2 y las de flyway.
 - **Db.migration:** Aquí se incluyen las migraciones realizadas sobre la base de datos. Estas migraciones son: creación de las tablas, inserción de primitivas, inserción de euromillones, inserción de ruletas e inserción de los usuarios “admin” y “user”.
 - **Public:** incluye varios archivos de diseño de las vistas.
 - **Templates:** En esta carpeta se incluyen las distintas vistas de la aplicación. Para cada juego se incluyen 4 vistas: añadir datos, actualizar datos, mostrar resultados, y las estadísticas del juego. Además, se incluyen el login y el registro de nuevos usuarios y una vista común que incluye una barra de navegación y distintos scripts de diseño.

Para acceder a cada una de las vistas anteriores, será necesario estar registrado en la aplicación, salvo en el caso de las vistas de login y registro de usuario. Además, en el caso de las vistas de edición o añadido de datos de primitivas y euromillones, el usuario registrado deberá contar con el rol de administrador.

3.8.1 Autenticación

Para implementar la autenticación de la aplicación, se ha utilizado Spring Security, ya que implementa por el desarrollador muchas de las partes del código, ahorrando tiempo y esfuerzo.

En primer lugar, se ha definido una nueva configuración de seguridad. Esta clase extiende la interfaz `WebSecurityConfigurerAdapter`, que será la que utilice el framework para poder configurar las distintas partes de la aplicación basándose en la configuración personalizada de esta nueva clase.

En esta clase se definen las rutas a las que puede acceder un usuario según su rol o si está autenticado o no y cuál será la vista de login, además, se define el hash que utilizan las contraseñas de los usuarios para guardarlas de forma segura y algunas otras características.

Por otro lado, se implementa la interfaz `UserDetails`, de Spring Security. En esta clase se definen los principales métodos de la interfaz y será la que luego utilice Spring para saber si el usuario es válido o no y cuál es su rol.

Además, implementa otros métodos para realizar el guardado de usuarios u obtener el usuario principal. Este usuario principal se obtiene a través de la request http de la aplicación, inyectada por Spring a la clase, a través de la cual se obtienen los detalles del usuario registrado en la aplicación.

Por último, en las vistas, usando Thymeleaf, se limitan los elementos que se pueden visualizar según el rol, como botones de edición de primitivas o euromillones para que sólo se puedan visualizar desde una cuenta de administrador.

Para poder comprobar estas características y observar diferencias entre roles, se han añadido dos usuarios, uno cuyo nombre de usuario es “admin” y su contraseña es “123” y otro cuyo nombre de usuario es “user” y su contraseña es “123”. Estos usuarios tienen rol de administrador y usuario, respectivamente.

3.9 Pruebas realizadas

En este apartado de pruebas se ha decidido optar por realizar test unitarios y de integración que demuestren que las principales funciones de la aplicación funcionan correctamente y, sobre todo, se ha buscado comprender cómo funcionan de forma más práctica este tipo de pruebas.

3.9.1 Pruebas unitarias

En el apartado de pruebas unitarias se han realizado pruebas sobre los controladores de la aplicación.

En estas pruebas, se ha comprobado el funcionamiento de los métodos de creación, actualización y borrado de cada uno de los juegos incluidos en la aplicación.

De esta forma, se pueden detectar de manera rápida posibles errores que existan en la aplicación, o nuevos errores que se puedan introducir al modificar la aplicación durante un posible mantenimiento futuro.

Para cada uno de los juegos se han creado métodos similares, siguiendo los mismos pasos y cambiando únicamente los repositorios y controladores correspondientes.

Cada una de las clases de pruebas unitarias creadas comienza con la declaración de un controlador y un repositorio, además de un BindingResult, que se encarga de comprobar los valores introducidos y un Model, que es la clase que recibe el modelo de la vista. Mediante anotaciones se hace el mock correspondiente para que la clase de test unitarios se ejecute correctamente.

Por un lado, **@Mock** se encarga de marcar y declarar los atributos como mocks de forma que durante los test se puedan utilizar sin necesidad de hacer llamadas a Mockito.mock de forma manual, además de poder definir su comportamiento en estos.

Por otro lado, **@InjectMocks** inyecta las dependencias que necesita la clase utilizando los mocks declarados en la misma y definidos en sus test.

En el siguiente test, primero se instancia un juego, que será el que se le pasará más adelante al controlador. A continuación, se define el comportamiento que se necesite de los Mocks declarados, en este caso se define el método de búsqueda repositorio.

```
@Test
public void testEuromillonesAdd() {
    Euromillones euromillones = new Euromillones();
    euromillones.setId(11);
    when(model.addAttribute("euromillones", euromillonesRepository.findAll())).thenReturn(model);

    String resul=euromillonesController.addGame(euromillones, bindingResult, model);

    assertThat(resul, is("euromillones/euromillones"));
}
```

Figura 3.9.1:test add euromillones

Este método debe ser definido en mediante el mock, ya que, al realizar la llamada al método correspondiente del controlador, se hace una llamada al método y, al tratarse de pruebas unitarias, no se puede incluir el funcionamiento real del método ya que se estaría incluyendo su código y pasaría a ser otro tipo de prueba.

A continuación, se hace la llamada al método y por último se comprueba que el resultado es el esperado.

De igual manera, teniendo en cuenta aquellos métodos que necesiten un mock, se realizan las pruebas de los métodos de actualización y borrado de cada uno de los juegos de la aplicación.

3.9.2 Pruebas de integración

Las pruebas de integración se han realizado sobre los repositorios de los juegos de la aplicación, incluyendo para cada uno, pruebas sobre los métodos de búsqueda, guardado y borrado.

Para realizar estas pruebas, se ha creado, para cada uno de los juegos, una clase de pruebas de integración, en la que se han incluido los métodos de prueba correspondientes.

Cada una de estas clases cuenta con atributo, que será el repositorio correspondiente en cada caso. En este caso, el repositorio será inyectado por Spring en lugar de inyectarlo mediante mocks, como en el caso anterior. De esta forma, se puede comprobar su funcionamiento con el resto de la aplicación y sus dependencias.

A continuación, se añaden las pruebas que comprueban la correcta integración de la clase en la aplicación, en estas clases todo lo que se tendrá que hacer es preparar unos parámetros, realizar la prueba y comprobar el resultado devuelto.

Como muestra de uno de los test de integración, se muestra el siguiente método de prueba de borrado de elementos.

```
@Test
public void testDelete() {
    Primitiva p=new Primitiva();
    p.setId(11);
    primitivaRepository.save(p);

    primitivaRepository.delete(p);
    long primitivasDespues = primitivaRepository.count();

    assertThat(11, equalTo(primitivasDespues));
}
```

Figura 3.9.2:prueba integración de borrado de juegos

Como se aprecia en la imagen, primero se preparan los datos de prueba, instanciando y guardando un juego, después se realiza una llamada al método que se quiere probar, delete en este caso, guardando el resultado obtenido, es decir, el número de elementos tras la llamada. Por último, se comprueba que este número de elementos coincide con el esperado.

3.10 Despliegue de la aplicación

Para realizar el despliegue de la aplicación, sólo habrá que seguir una serie de pasos sencillos expuestos a continuación. En este caso, se utiliza Tomcat para realizar el despliegue.

En primer lugar, se debe definir el modo de compilación del proyecto en eclipse, para ello, habrá que acceder al menú correspondiente y seleccionarlo, en este caso jdk 1.8. Además, se añade un nuevo bloque en el pom.xml de la aplicación, tal como se muestra en la siguiente figura. [24]

```
<build>
  <finalName>${project.artifactId}</finalName>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <fork>true</fork>
        <executable>"C:/Program Files (x86)/Java/jdk1.8.0_101/bin/javac.exe"</executable>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Figura 3.10.1: bloque build

En este nuevo bloque se indica a Maven el nombre que deberá tener el proyecto, además del compilador a utilizar, de forma que no se produzcan errores a la hora de realizar el despliegue.

A continuación, se ha generado el archivo WAR. Para ello, en la consola del sistema, ubicado en la carpeta del proyecto, simplemente se debe ejecutar el comando de la siguiente figura.

```
E:\Usuario\Documents\OneDrive\OneDrive - Universidad Politécnica de Madrid\UNI\TFG\statGamblerWS\statGambler>mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.statGambler:statGambler >-----
[INFO] Building statGambler 0.0.1-SNAPSHOT
```

Figura 3.10.2: comando compilación Maven

Una vez ejecutado, si todo ha salido correctamente, Maven genera un archivo WAR, listo para utilizar con Tomcat en este caso.

Para poder utilizar la aplicación con tomcat, simplemente se debe llevar este fichero WAR generado a la ruta /webapps de la carpeta de tomcat y una vez ubicado, con una consola de comandos abierta en la carpeta /bin de tomcat, se deberá ejecutar el comando de la siguiente figura si el comando se ejecuta en windows.

```
E:\Usuario\Downloads\apache-tomcat-8.5.41\bin>catalina.bat run
Using CATALINA_BASE:   "E:\Usuario\Downloads\apache-tomcat-8.5.41"
```

Figura 3.10.3: catalina.bat run

Una vez seguidos estos pasos, se podrá acceder a la aplicación a través de la dirección localhost:8080/statGambler, en este caso.

En el caso del proyecto, se ha seguido un paso adicional para evitar la raíz añadida a la dirección web, ya que la aplicación ha sido desarrollada sin esta raíz y, si no se realiza este cambio, la aplicación no es capaz de encontrar los recursos al hacer click en los enlaces, ya que la dirección será errónea.

El cambio realizado es simplemente borrar la carpeta ROOT que trae por defecto Tomcat y renombrar el WAR del proyecto de statGambler.war a ROOT.war. De esta forma, se evita esa raíz y la aplicación puede funcionar correctamente.

Como alternativa, también se puede declarar en el fichero pom.xml el packaging como jar y, al ejecutar en la carpeta del proyecto el comando “mvn clean package”, Maven generará una nueva carpeta target en la que se incluye el nuevo fichero .jar.

Una vez generado el fichero, sólo habrá que ejecutarlo mediante el comando `java -jar <nombre.jar>`, tal como muestra la figura. Mientras se encuentre en ejecución, se podrá acceder a la app a través de localhost:8080 en este caso.

```
java -jar statGambler.jar
```

Figura 3.10.4: comando ejecución proyecto

4 Conclusiones

Como conclusión del proyecto, se puede decir que ha servido para realizar una buena toma de contacto con Spring Boot, además de con muchos otros elementos útiles a la hora de crear una aplicación, como es el caso de hibernate o JUnit.

A través del proyecto, se ha podido comprender, desde un punto de vista teórico y práctico, cómo se desarrolla un proyecto Spring, incluyendo tanto las facilidades como los inconvenientes que presenta.

Entre las características más destacables aprendidas de los frameworks, una vez realizado el proyecto, se pueden incluir la inyección de dependencias, realmente útil para evitar dependencias fuertes y facilitar el desarrollo de aplicaciones o la facilidad que dan algunas de las dependencias incluidas en el proyecto, como flyway e hibernate a la hora de realizar migraciones o gestionar la base de datos.

Algunos de los inconvenientes encontrados podrían ser la falta de información en algunos casos o la dificultad para poder continuar al encontrar un error, durante el proceso de creación de un nuevo proyecto.

En definitiva, gracias a este proyecto, se ha adquirido tanto nuevos conocimientos con el desarrollo de aplicaciones con Spring, como una visión global más práctica del ciclo de vida de creación de una nueva aplicación.

4.1 Problemas afrontados

En primer lugar, cuando se comenzó a crear la aplicación, se utilizó Spring Initializr, pero, al encontrar problemas con la base de datos y teniendo en cuenta que ya se disponía de un proyecto de ejemplo que tenía una base de datos configurada de la parte de teoría, se decidió reutilizar este proyecto refactorizando nombres y realizando cambios en la configuración.

En un principio, al reutilizar el proyecto utilizado como ejemplo durante la parte teórica, se afrontaba el problema de desarrollar js. Dado que no se había utilizado nunca, y sumando su aparente complejidad, se optó por buscar otras solución más sencilla y rápida para el desarrollo, llegando a la solución de la implementación de thymeleaf, que permite añadir la funcionalidad de la aplicación sin necesidad de programar este tipo de ficheros.

Además, se afrontó el problema de las migraciones, al no comprender bien su funcionamiento y utilidad, simplemente se modificaban los scripts existentes cada vez que se quería modificar la estructura de la base de datos de alguna forma. Esto llevaba a varios problemas durante la ejecución con flyway, ya que no se habían realizado las migraciones pertinentes.

Afortunadamente, estos errores se producían durante fases tempranas del desarrollo, por lo que se podía eliminar la base de datos completamente con un comando de flyway, clean, y continuar con la base de datos vacía.

Durante el desarrollo de la aplicación, en un principio, se optó por afrontar primero las clases padre más abstractas de la aplicación, para luego bajar a clases concretas.

Debido a la complejidad que parecía tener de esta forma, se optó finalmente deshacer estas clases padre y comenzar con un par de clases hijas para luego abstraer todo lo que se pudiese de estas clases concretas y simplificar en un futuro añadir funcionalidades a la aplicación.

En un principio, se intentó plantear una solución que utilizase una interfaz Game, para abstraer la utilización de clases, permitiendo mejorar la calidad del código y la facilidad a la hora de componentes, pero jpa no contempla la utilización de interfaces en los modelos utilizados para el repositorio.

Tras intentar algunas soluciones, intentando usar los hooks proporcionados por Spring, con los que se alteraba la creación de los repositorios, no se llegó a una solución del problema. Debido al tiempo que estaba consumiendo el problema, se optó por volver a utilizar clases concretas para poder seguir avanzando.

Aunque el código deba depender de esta forma de clases concretas, Spring facilita el desacoplamiento entre clases, por tanto, las dependencias no son fuertes y, aunque el código sea de calidad algo peor y la cantidad de trabajo de desarrollo haya aumentado, no supone un problema mayor, por tanto, la solución se considera aceptable, aunque no la mejor posible.

A la hora de utilizar Thymeleaf, a veces, la documentación podía llegar a resultar confusa, haciendo que una simple iteración por índice se pudiese llegar a complicar por no saber cómo realizar esta implementación y no llegar a encontrar documentación que diese solución rápida al problema.

Aun así, utilizando dicha documentación, además de distintos foros, se podía llegar a dar solución a los problemas afrontados tras varios intentos y ejemplos encontrados.

En algunos casos Bootstrap podía resultar confuso también, tal vez por falta de experiencia, debido a la cantidad de etiquetas anidadas que existían en algunas vistas de la aplicación. Pero gracias a su popularidad y cantidad de ejemplos y contenido disponible en internet, estos problemas afrontados se podían solucionar de manera sencilla.

4.2 Posibles mejoras

El desarrollo de este prototipo tenía como objetivo una mera toma de contacto con Spring y algunos frameworks de apoyo, aun así, algunas características que podrían ayudar a mejorar la aplicación son las siguientes:

- Agregar una funcionalidad real para usuarios, ya que actualmente, no se relaciona a los usuarios con los datos introducidos, y todos comparten los mismos datos.

- Agregar unos datos de prueba reales del juego de la ruleta para poder realizar un estudio real, ya que actualmente, son simplemente datos generados al azar.
- Implementar un sistema de recomendación de jugadas, en el que se indique al jugador cuál es la mejor jugada que tiene disponible, teniendo en cuenta su dinero disponible y los datos estadísticos del juego en concreto.
- Para mejorar la claridad de los datos expuestos, se podrían añadir a las vistas unos gráficos que muestren los datos calculados de una forma mucho más visual.

5 Bibliografía

- [1] Lista de reproducción Spring Framework. Youtube, subido por el usuario MitoCode, 17 Mayo 2018 [fecha de consulta Febrero 2019]. Disponible en: https://www.youtube.com/playlist?list=PLvimn1Ins-40ClmsffjCkv_TrKzYiB1gb
- [2] Imagen Spring Framework Runtime. Spring, autor desconocido, fecha de subida desconocida [fecha de consulta Febrero 2019]: <https://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/overview.html>
- [4] MVC (Model, View, Controller) explicado.CodigoFacilito, autor desconocido, fecha desconocida [Fecha de consulta Febrero 2019]. Disponible en: <https://codigofacilito.com/articulos/mvc-model-view-controller-explicado>
- [6] Curso Spring Fundamentals. Pluralsight, presentado por Bryan Hansen, 9 Diciembre 2016 [fecha de consulta marzo 2019]. Disponible en: <https://app.pluralsight.com/library/courses/spring-fundamentals/table-of-contents>
- [7] Programación orientada a aspectos (AOP) en Spring. Depto. jTech, Ciencia de la computación e IA, Universidad de Alicante, 26 Junio 2014 [fecha de consulta Marzo 2019]. Disponible en: http://www.itech.ua.es/j2ee/publico/spring-2012-13/apendice_AOP-apuntes.html
- [8] Design Pattern - Abstract Factory Pattern. Tutorialspoint, autor desconocido, fecha desconocida [Fecha de consulta 3 Marzo 2019]. Disponible en: https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm
- [9] Spring Tool Suite IDE. Spring.io, autor desconocido, fecha desconocida [Fecha de consulta Febrero 2019]. Disponible en: <https://spring.io/tools>
- [11] Parte del curso Introduction to Spring MVC. Pluralsight, presentado por Bryan Hansen, 14 Enero de 2013 [fecha de consulta Febrero 2019]. Disponible en: <https://app.pluralsight.com/library/courses/springmvc-intro/table-of-contents>
- [12] Parte AOP del curso Aspect Oriented Programming (AOP) using Spring AOP and AspectJ. Pluralsight, presentado por Eberhard Wolff, 14 Octubre 2013 [fecha de consulta Febrero 2019]. Disponible en: <https://app.pluralsight.com/player?course=aspect-oriented-programming-spring-aspectj&author=eberhard-wolff&name=aspect-oriented-programming-spring-aspectj-m3-why-aop&clip=0>
- [13] Spring: The Big Picture. Pluralsight, presentado por Dustin Schultz, 15 Mayo 2018 [fecha de consulta Febrero 2019]. Disponible en: <https://app.pluralsight.com/library/courses/spring-big-picture/table-of-contents>
- [14] Developer Survey Results 2018, Most Loved, Dreaded, and Wanted Frameworks, Libraries, and Tools. Stackoverflow, Diversos autores, 2018, [fecha de consulta Febrero 2019]. Disponible en: <https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted>
- [15] Inversión de Control - Principio de Hollywood: “Don’t call us, we’ll call you”. Medium, Miguel Ángel Sánchez, 26 Septiembre 2017 [fecha de consulta Mayo 2019]. Disponible en: <https://medium.com/all-you-need-is-clean->



[code/inversi%C3%B3n-de-control-principio-de-hollywood-dont-call-us-we-ll-call-you-179e9c70e3d0](#)

- [16] El patrón de inyección de dependencia y su utilidad. Arquitectura Java, Cecilio Álvarez Caules, 13 agosto 2018 [fecha de consulta Mayo 2019]. Disponible en: <https://www.arquitecturajava.com/el-patron-de-inyeccion-de-dependencia/>
- [17] Inversión de Dependencias (DIP). Medium, Miguel Ángel Sánchez, 21 septiembre 2017 [fecha de consulta Mayo 2019]. Disponible en: <https://medium.com/all-you-need-is-clean-code/inversi%C3%B3n-de-dependencias-dip-b8a07b42f99e>
- [18] Principio de Inversión de Dependencias (SOLID 5ª parte). Devexperto, Antonio Leiva, enero 2016 [fecha de consulta Mayo 2019]. Disponible en: <https://devexperto.com/principio-de-inversion-de-dependencias/>
- [19] Spring bean scopes. HowToDoInJava, Lokesh Gupta, 2013 [fecha de consulta Mayo 2019]. Disponible en: <https://howtodoinjava.com/spring-core/spring-bean-scopes/>
- [20] Spring Bean Scopes: Guía para comprender los distintos scopes (ámbitos) de un Spring Bean. Blog Marcnuri, Marc Nuri, 19 Noviembre 2017 [fecha de consulta Mayo 2019]. Disponible en: <https://blog.marcnuri.com/spring-bean-scopes-guia-rapida/>
- [21] Spring Boot CRUD Application with Thymeleaf. Baeldung, Alejandro Ugarte, 6 Abril 2019, [fecha de consulta Mayo 2019]. Disponible en: <https://www.baeldung.com/spring-boot-crud-thymeleaf>
- [22] ESPERANZA MATEMÁTICA DE LAS LOTERÍAS. Estadística para todos, autor desconocido, fecha desconocida [fecha de consulta Junio 2019]. Disponible en: <http://www.estadisticaparatodos.es/taller/loterias/esperanza.html>
- [23] Probabilidades en La Primitiva. Combinación ganadora, autor desconocido, fecha desconocida [fecha de consulta Junio 2019]. Disponible en: <https://www.combinacionganadora.com/primitiva/probabilidades/>
- [24] Deploy a Spring Boot WAR into a Tomcat Server. Baeldung, baeldung, 21 Febrero 2019 [fecha de consulta Junio 2019]. Disponible en: <https://www.baeldung.com/spring-boot-war-tomcat-deploy>
- [25] Retrieve User Information in Spring Security. Baeldung, Eugen Paraschiv, 26 Mayo 2019, [Fecha de consulta Junio 2019]. Disponible en: <https://www.baeldung.com/get-user-in-spring-security>
- [26] Hidden Field Value Blank Thymeleaf. StackOverflow, Govardhana Rao Ganji, 1 marzo 2016 [Fecha de consulta Junio 2019]. Disponible en: <https://stackoverflow.com/questions/25808433/hidden-field-value-blank-thymeleaf>
- [27] Thymeleaf - Spring Security integration modules. Github, [danielfernandez](#), Mayo 2016 [Fecha de consulta Junio 2019] <https://github.com/thymeleaf/thymeleaf-extras-springsecurity>
- [28] Thymeleaf + Spring Security integration basics. Thymeleaf, José Miguel Samper, fecha desconocida, [Fecha de consulta Junio 2019] <https://www.thymeleaf.org/doc/articles/springsecurity.html>