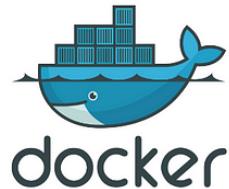




Fondo Social Europeo
El FSE invierte en tu futuro

MEMORIA FINAL DE PROYECTO

IMPLANTAR APLICACIÓN WEB CON ‘SPRING BOOT’



CICLO FORMATIVO DE GRADO SUPERIOR:
DESARROLLO DE APLICACIONES MULTIPLATAFORMA

AUTOR:

JORGE MORENO BRASERO

TUTOR:

LUIS MORENO LARA

COORDINADOR:

JOSÉ LUIS LÓPEZ ÁLVAREZ

CURSO:

2022 - 2023

I.E.S CLARA DEL REY

ÍNDICE

| | |
|---|-----------|
| 1. INTRODUCCIÓN | 4 |
| 1.1 Descripción | 4 |
| 1.2 Description | 4 |
| 2. ALCANCE DEL PROYECTO | 4 |
| 2.1 Objetivo | 5 |
| 2.2 Requisitos | 5 |
| 2.3 Planificación previa | 6 |
| 3. PLANTEAMIENTO DE POSIBLES SOLUCIONES | 8 |
| 3.1 Posibles sistemas gestores de bases de datos | 8 |
| 3.2 Posibles frameworks de desarrollo de aplicaciones web | 9 |
| 4. ANÁLISIS DE LA SOLUCIÓN ESCOGIDA | 10 |
| 4.1 ¿Qué es Spring? | 12 |
| 4.2 ¿Qué es JSP? | 14 |
| 4.3 ¿Qué es Bootstrap? | 15 |
| 4.4 ¿Qué es jQuery? | 16 |
| 4.5 ¿Qué es Ehcache? | 17 |
| 4.6 ¿Qué es i18n? | 18 |
| 4.7 ¿Qué es JUnit? | 19 |
| 4.8 ¿Qué es Selenium? | 20 |
| 4.9 ¿Qué es Docker y Docker Compose? | 21 |
| 5. DISEÑO DE LA SOLUCIÓN ESCOGIDA | 22 |
| 5.1 Base de datos | 23 |
| 5.2 Aplicación Web | 25 |
| 5.3 Herramientas | 26 |
| 6. IMPLEMENTACIÓN DEL DISEÑO | 27 |
| 6.1 Creación del proyecto de Spring | 28 |
| 6.2 Creación de los modelos | 33 |
| 6.3 Creación de la capa de repositorios | 35 |
| 6.4 Creación de la capa de servicios | 37 |
| 6.5 Creación de la capa Façades | 39 |
| 6.6 Creación de la capa de controladores | 43 |
| 6.6.1 Creación de las páginas | 44 |
| 6.6.1.1 Pagina principal | 44 |
| 6.6.1.2 Pagina por juego | 47 |
| 6.6.1.3 Pagina de registro | 49 |
| 6.6.1.4 Pagina de login | 50 |

| | |
|--|-----------|
| 6.6.1.5 Pagina de creación y edición de juegos | 51 |
| 6.6.2 Creación de los controladores | 53 |
| 7. PRUEBAS | 57 |
| 7.1 Plan de pruebas | 57 |
| 7.2 Resultado de las pruebas | 57 |
| 8. EXPLOTACIÓN | 59 |
| 8.1 Dockerización | 60 |
| 8.2 Sincronización de Dockers | 60 |
| 8.3 Creación de la imagen de Docker | 63 |
| 8.4 Implantación en un entorno real | 64 |
| 9. CONCLUSIONES | 69 |
| 9.1 Conclusiones personales | 69 |
| 9.2 Posibles aplicaciones futuras | 69 |
| GLOSARIO | 70 |
| WEBGRAFÍA | 72 |
| ANEXO | 74 |

1. INTRODUCCIÓN

1.1 Descripción

Creación de una aplicación web construida con Spring Boot que interconecta una base de datos de MySQL, desplegada con Docker, con unas páginas web hechas en JSP. La aplicación se nutre de Steam para mostrar capturas, la carátula y las últimas noticias relacionadas con los videojuegos. Tiene implementado un sistema de seguridad basado en usuarios que se consigue gracias al módulo de Spring Security.

1.2 Description

A web application built with Spring Boot that connects a MySQL database, deployed using Docker, with web pages made with JSP. This application takes from Steam the cover, the last news and some screenshots related to the videogame. It has implemented a user-based security system that is performed thanks to Spring Security’s module.

2. ALCANCE DEL PROYECTO

El proyecto contará con una aplicación web de Spring Boot, unos test unitarios y de integración hechos con JUnit 4, unos tests end-to-end hechos con la librería Selenide y una seguridad dentro de la aplicación, gracias al módulo de Spring Security, basado en usuarios.

Además de esto, tendrá unas interfaces web que están hechas en JSP y contará con una tarea de Gradle que nos permite generar una imagen de Docker de la aplicación con el objetivo de utilizar un docker-compose para orquestar el despliegue del programa en cualquier dispositivo que tenga instalado Docker simplemente ejecutando un comando.

2.1 Objetivo

Este proyecto se ha llevado a cabo debido a la necesidad de aprender las tecnologías más utilizadas en la actualidad por las empresas. Cuando buscamos ofertas de trabajo para programadores Java, en su mayoría son para construir aplicaciones web con Spring.^{1 2}

Se pretende crear una aplicación web que pueda simular una aplicación empresarial robusta, modularizada y que nos permita hacer todas las operaciones CRUD frente a una base de datos. En nuestro caso, tenemos una base de datos que almacena videojuegos, pero ésta podría ser una sobre los empleados de una empresa o una tienda de productos.

2.2 Requisitos

Para conseguir hacer una aplicación web con las características ya mencionadas necesitaremos una página principal pública donde se muestren los videojuegos contenidos en la base de datos además de un buscador por título. Tendremos otra página pública que muestre los datos relacionados con el videojuego tales como el título, la descripción, una captura de pantalla y las últimas noticias.

Para poder hacer todas las operaciones CRUD debemos tener unas páginas privadas para solo usuarios autenticados que nos permita modificar y eliminar los videojuegos ya existentes, o crear juegos nuevos.

Además, necesitaremos unas interfaces para la gestión de usuarios como el login o el registro.

2.3 Planificación previa

La planificación consta de 4 partes diferenciadas:

1. Aprendizaje y búsqueda de información sobre las tecnologías que se van a utilizar.
2. Desarrollo de la aplicación y las interfaces web.
3. Desarrollo de los test.
4. Implantación en un entorno simulado.

| Proyecto de implementación de aplicación web con Spring Boot | | | |
|---|------------------------|------------------------|--------------------|
| Lista de tareas | Fecha de inicio | Duración (días) | Fecha final |
| Aprendizaje y búsqueda de información | 23/03/23 | 10 | 14/04/23 |
| Desarrollo de la aplicación e interfaces web | 17/04/23 | 25 | 24/05/23 |
| Testeo de la aplicación | 25/05/23 | 7 | 02/06/23 |
| Implementación en entorno simulado | 05/06/23 | 5 | 09/06/23 |
| Realización de Memoria | 08/05/23 | 27 | 13/06/23 |

Figura 1.1: Planificación del proyecto.

Fuente: Propia

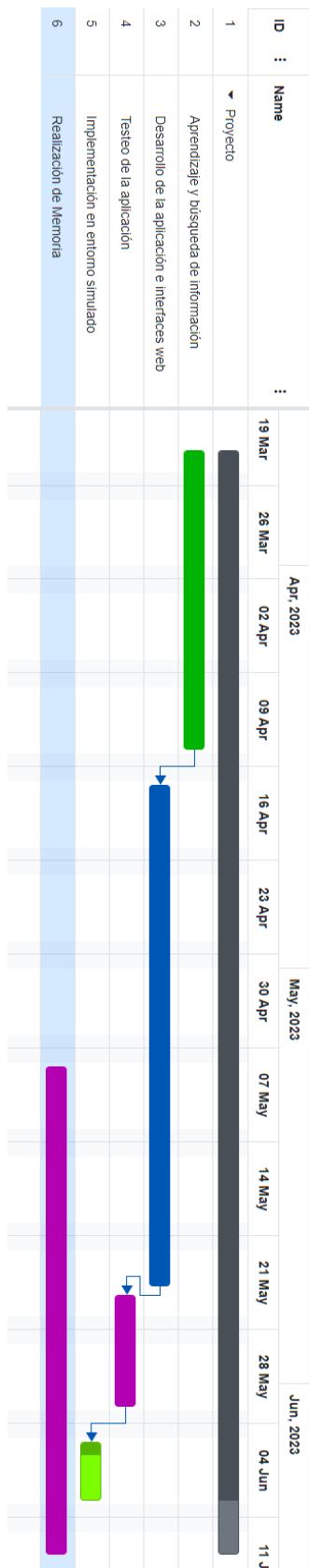


Figura 1.2: Diagrama de Gantt.

Fuente: [Gantt Online](#)

3. PLANTEAMIENTO DE POSIBLES SOLUCIONES

Exploramos las posibles soluciones para dos aspectos clave en el desarrollo de este proyecto: la gestión de la base de datos y la creación de una aplicación web. Dividiremos las soluciones en dos partes.

En primer lugar, veremos los posibles sistemas gestores de bases de datos disponibles y sus diferencias.

En segundo lugar, abordaremos la creación de una aplicación web considerando el diseño intuitivo, la elección de lenguaje de programación adecuado, las arquitecturas escalables y el uso de frameworks y librerías de desarrollo.

3.1 Posibles sistemas gestores de bases de datos

Entre los distintos sistemas gestores de bases de datos encontramos los siguientes:

- **Oracle Database** es uno de los SGBD más populares y ampliamente utilizados. Es conocido por su capacidad para manejar grandes volúmenes de datos y ofrecer un alto rendimiento. El principal inconveniente es que no es gratuito.
- **MySQL** es un SGBD, que pertenece a Oracle, utilizado especialmente en aplicaciones web. Es conocido por su facilidad de uso, alto rendimiento y gran escalabilidad. MySQL, a su vez, es compatible con múltiples plataformas.
- **SQL Server** es un SGBD desarrollado por Microsoft y es popular en entornos Windows. Ofrece una amplia gama de características, incluyendo almacenamiento en memoria, seguridad avanzada y herramientas de análisis de datos. SQL Server es conocido por su integración con otras tecnologías de Microsoft, como .NET Framework. Tiene algunas versiones gratuitas como SQL Server Express pero tiene limitaciones en capacidad y rendimiento.
- **PostgreSQL** es un SGBD de código abierto y robusto, conocido por su capacidad de manejar datos complejos y su compatibilidad con estándares SQL. Proporciona características avanzadas como replicación, transacciones ACID y soporte para consultas geoespaciales. PostgreSQL es ampliamente utilizado en entornos de desarrollo y producción.

3.2 Posibles frameworks de desarrollo de aplicaciones web

Entre los distintos frameworks para crear una aplicación web encontramos los siguientes:

- **Spring** es un popular framework de desarrollo de aplicaciones Java. Ofrece características como la Inyección de Dependencias y el soporte para servicios web, lo que nos facilita la creación de aplicaciones escalables y modularizadas.
- **Django** es un framework de desarrollo web de código abierto en Python. Ofrece una estructura eficiente y funcionalidades integradas para crear aplicaciones web rápidas y seguras. Es una opción popular para desarrolladores que valoran la simplicidad y la productividad.
- **ASP.NET** es un framework de desarrollo web de Microsoft utilizado para crear aplicaciones web escalables y seguras en C# o Visual Basic.NET. Ofrece herramientas para la autenticación, la autorización y el acceso a bases de datos, y se integra fácilmente con tecnologías como Azure.
- **Laravel** es un framework de desarrollo web de código abierto basado en PHP. Entre sus principales características se encuentran el enrutamiento fácil de usar, un ORM intuitivo, autenticación y autorización integradas, y un sistema de plantillas dinámicas llamado Blade. Laravel también proporciona una sólida arquitectura MVC que promueve una estructura organizada y modular.
- **Symfony** es un framework de desarrollo de aplicaciones web basado en PHP. Reconocido por su enfoque en la modularidad y la reutilización de código. Con él se pueden crear aplicaciones web escalables y de alto rendimiento de manera eficiente. Symfony sigue los principios de diseño y desarrollo de software como el patrón de diseño MVC, lo que permite una separación clara de la lógica de negocio y la presentación visual.

4. ANÁLISIS DE LA SOLUCIÓN ESCOGIDA

Al examinar las posibilidades en el ámbito de las bases de datos, considero que MySQL es la opción más adecuada. MySQL es un sistema gratuito ampliamente utilizado en el desarrollo de aplicaciones web.

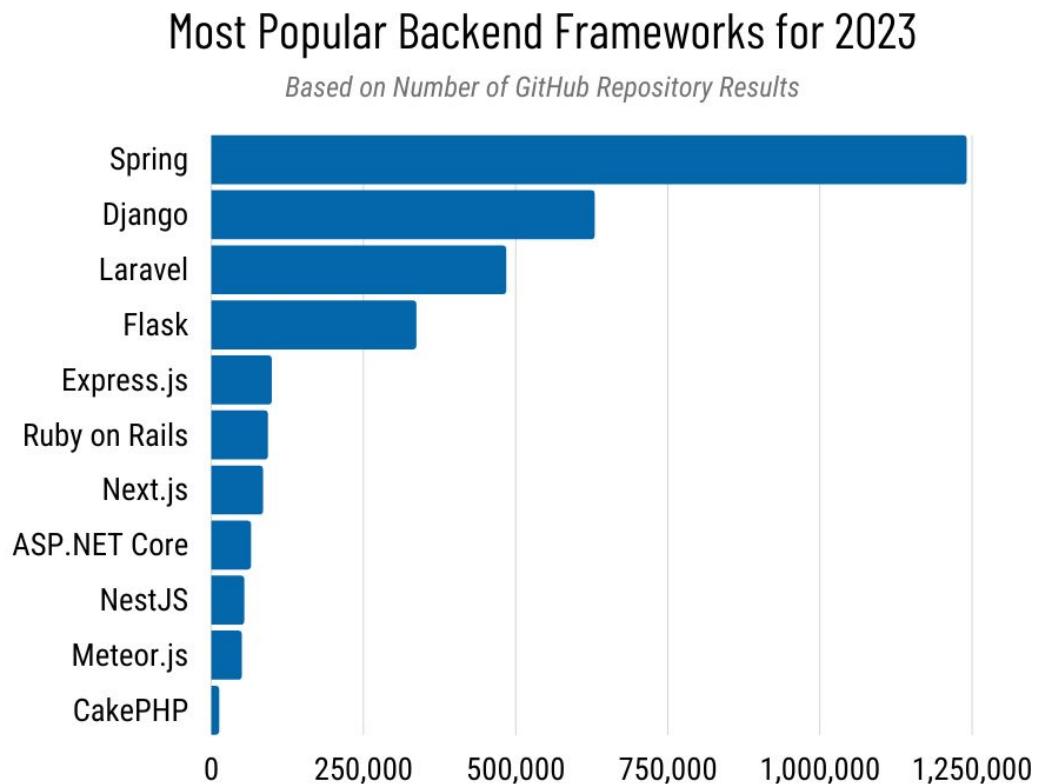


Figura 2: Comparativa frameworks web.

Fuente: [CodingNomads](#)

Teniendo en cuenta que el lenguaje de programación que domino es Java, he visto conveniente usar el framework de Spring en este proyecto.

Spring es ampliamente reconocido como el framework más utilizado en el desarrollo de aplicaciones empresariales. He decidido usar Spring debido a su poderoso ecosistema. Su elección se basa en su probada eficacia por lo que considero que es la mejor opción para desarrollar aplicaciones empresariales robustas y escalables.

Para que las páginas sean lo más profesionales posibles se utilizarán frameworks como Bootstrap para una maquetación sencilla basada en clases de CSS y jQuery para poder manejar los contenedores que existan en la página y consigamos que las interfaces sean lo más intuitivas posible.

Al estar haciendo peticiones a fuentes externas, estas puede que no respondan todo lo rápido que deseamos o, en ocasiones tenemos un límite de peticiones y podemos superar ese límite fácilmente. Para ello se ha implementado una caché en memoria. En Spring la caché se puede conseguir de forma nativa pero para configurarla, de una forma más sencilla, se usan los llamados *cache providers*. En mi caso, he utilizado Ehcache debido a que es un proveedor de caché popular dentro del ecosistema de Spring.



Figura 3.1: Spring Boot Framework.

Fuente: [Spring Boot](#)

4.1 ¿Qué es Spring?

Spring es un framework Java de código abierto que nos permite crear herramientas y soluciones para aplicaciones empresariales robustas. Spring en sí mismo es un framework muy potente pero con una alta complejidad en su configuración e inclusión de las librerías necesarias, para solucionar este problema se recomienda el uso de Spring Boot.

Con **Spring Boot** la configuración se simplifica, permitiendo crear aplicaciones listas para producción de forma rápida. La ventaja que nos ofrece es muy considerable por lo que se suele utilizar Spring Boot antes que Spring nativo para crear aplicaciones web.

Spring Security se encarga de la seguridad, proporcionando características avanzadas de autenticación y autorización. Además nos permite conocer ciertos datos sobre la sesión y nos facilita la gestión de los usuarios de forma notable.

Spring MVC es un componente clave de Spring para desarrollar aplicaciones web escalables utilizando el patrón Modelo-Vista-Controlador. Con Spring MVC, se logra una separación clara entre la lógica de negocio, la presentación de datos y el control de la interacción. Esto facilita el desarrollo eficiente de aplicaciones web y su mantenimiento a largo plazo.

Spring MVC permite usar varios tipos de plantillas siendo los más utilizados Thymeleaf y JSP. JSP ofrece una sintaxis familiar y similar a HTML, lo que nos permite trabajar con facilidad y rapidez. Además, tiene la disponibilidad de una amplia variedad de etiquetas personalizadas y bibliotecas de etiquetas, lo que facilita la creación de componentes reutilizables y el desarrollo de aplicaciones web complejas. Por eso, considero que JSP es superior a Thymeleaf.

En conjunto, Spring proporciona un entorno de desarrollo poderoso y flexible. Con gran cantidad de características y un enfoque modular.



Figura 3.2: Java Server Pages.

Fuente: [JSP](#)

4.2 ¿Qué es JSP?

JSP (Java Server Pages) es una tecnología de Java que permite crear páginas web dinámicas y basadas en componentes. Permite combinar código Java y HTML para generar contenido dinámico en el lado del servidor.

Las ventajas de usar JSP incluyen su integración con Java, lo que permite utilizar todas las funcionalidades y bibliotecas de Java en las páginas web generadas. Además, JSP es altamente escalable y permite reutilizar componentes, lo que acelera el desarrollo y mejora la productividad.

En resumen, JSP es una tecnología poderosa para crear páginas web dinámicas en Spring, ofreciendo ventajas en términos de integración, separación de preocupaciones y escalabilidad.



Figura 3.3: Bootstrap.

Fuente: [Bootstrap](#)

4.3 ¿Qué es Bootstrap?

Bootstrap es un framework de maquetación que proporciona herramientas y componentes pre estilizados para la creación rápida y fácil de interfaces web **responsive** y atractivas. Se destaca por su enfoque en la simplicidad y la compatibilidad multiplataforma.

En comparación con **Materialize** y **Tailwind**, Bootstrap se ha establecido como un estándar de la industria. Su documentación exhaustiva, amplia gama de componentes y su gran cantidad de temas y plantillas disponibles hacen que sea más fácil y rápido comenzar a construir aplicaciones web.

Además, su estructura **CSS** bien organizada y su sistema de cuadrícula flexible ofrecen una mayor consistencia y eficiencia en el desarrollo. Usar Bootstrap proporciona una base sólida y confiable para el desarrollo de interfaces web, con una gran cantidad de recursos disponibles.



Figura 3.4: jQuery.

Fuente: [jQuery](#)

4.4 ¿Qué es jQuery?

jQuery es una biblioteca de JavaScript rápida, liviana y de fácil uso que simplifica la manipulación y navegación del documento **HTML**, manejo de eventos, animaciones y llamadas **AJAX**.

Su objetivo principal es facilitar el desarrollo web al proporcionar una sintaxis sencilla y eficiente para interactuar con los elementos de una página web.

jQuery es compatible con una amplia gama de navegadores, lo que la convierte en una herramienta común y poderosa para mejorar la experiencia del usuario en aplicaciones web.



Figura 3.5: Ehcache.

Fuente: [Ehcach](#)

4.5 ¿Qué es Ehcache?

Ehcache es un proveedor de caché de código abierto y altamente escalable que proporciona una solución eficiente para almacenar en caché datos en aplicaciones Java.

Permite almacenar y recuperar datos en memoria, lo que mejora significativamente el rendimiento y reduce la carga en bases de datos u otros sistemas de almacenamiento.

Ehcache es fácil de configurar y se integra sin problemas con diferentes frameworks y tecnologías, como Spring, Hibernate y Java EE.

Las ventajas del uso de Ehcache incluyen una respuesta más rápida de la aplicación, un menor uso de recursos y una mayor escalabilidad al reducir las consultas a sistemas de almacenamiento externos. También mejora la experiencia del usuario al presentar datos almacenados en caché de forma rápida y consistente.



Figura 3.6: i18n internationalization.

Fuente: [i18n](#)

4.6 ¿Qué es i18n?

En el contexto de una aplicación Spring, la internacionalización (**i18n**) se refiere a la capacidad de adaptar y localizar la aplicación para diferentes idiomas y regiones. Permite que una aplicación pueda ser utilizada por usuarios de diferentes culturas, presentando textos, mensajes y recursos en el idioma adecuado según la configuración del usuario.

Spring proporciona un soporte completo para la internacionalización, permitiendo la externalización de textos y mensajes en archivos de propiedades específicas para cada idioma. Algunas de las ventajas de utilizar la internacionalización son la capacidad de llegar a un público más amplio, la mejora de la experiencia del usuario, una mayor facilidad de mantenimiento y actualización y una mayor adaptabilidad a diferentes culturas.



Figura 3.7: JUnit tests.

Fuente: [JUnit](#)

4.7 ¿Qué es JUnit?

JUnit es una tecnología para pruebas unitarias en Java que facilita la creación y ejecución de pruebas automatizadas. Permite a los desarrolladores escribir y ejecutar casos de prueba de forma rápida y sencilla, lo que mejora la calidad y confiabilidad del código.

La principal ventaja de usar JUnit es que proporciona una estructura clara y organizada para escribir pruebas, lo que facilita el mantenimiento y la comprensión del código de prueba.

En resumen, JUnit es una herramienta imprescindible para el desarrollo de software en Java, ofreciendo ventajas en términos de detección temprana de errores, agilidad en la depuración y estructura organizada de las pruebas.



Figura 3.8: Selenium Java.

Fuente: [Selenium](#)

4.8 ¿Qué es Selenium?

Selenium es una librería que se usa para la automatización de pruebas de software y se utiliza para probar la funcionalidad de aplicaciones web.

Se pueden escribir scripts de prueba en Java para interactuar con elementos de la interfaz de usuario, simular acciones de usuario y verificar el comportamiento de la aplicación web.

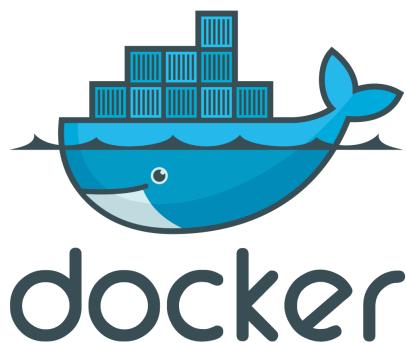


Figura 3.9: Docker containers.

Fuente: [Docker](#)

4.9 ¿Qué es Docker y Docker Compose?

Docker es una plataforma de código abierto que permite la creación, distribución y ejecución de aplicaciones en contenedores.

Los contenedores son entornos ligeros y aislados que encapsulan todas las dependencias y configuraciones necesarias para que una aplicación funcione de manera consistente en cualquier entorno.

La principal ventaja de Docker es su portabilidad, ya que los contenedores pueden ejecutarse en cualquier sistema operativo y en diferentes infraestructuras. Además, Docker facilita la gestión y el despliegue de aplicaciones al permitir la creación de imágenes reutilizables, lo que simplifica el proceso de desarrollo, pruebas y puesta en producción.

Docker Compose es una herramienta que permite definir y gestionar aplicaciones multi-contenedor de manera sencilla. Proporciona un enfoque declarativo para definir la configuración de múltiples servicios en un archivo YAML.

Con Docker Compose, es posible describir la estructura de la aplicación, especificar las dependencias entre los diferentes servicios y definir las variables de entorno. Una vez que se define el archivo de configuración, Docker Compose se encarga de crear y gestionar los contenedores de forma coherente.

5. DISEÑO DE LA SOLUCIÓN ESCOGIDA

La estructura general que tiene la aplicación es la siguiente:

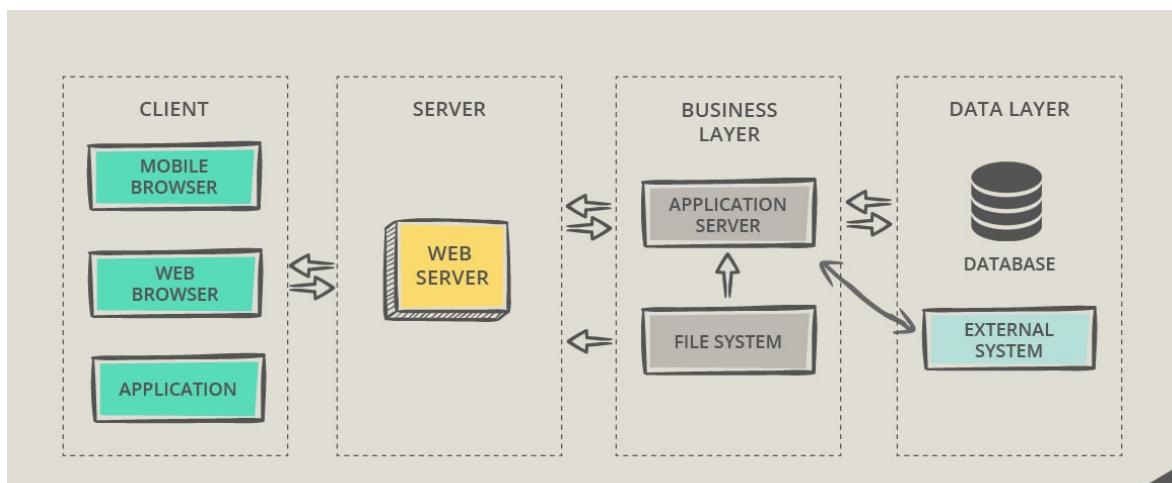


Figura 4: Diagrama de una aplicación web.

Fuente: [Existek](#)

Como vemos en la imagen de arriba, la petición se manda desde una aplicación como Postman o un navegador, el servidor web la recibe, la aplicación web lo maneja y accede a los recursos de la base de datos o a la API de Steam para recorrer el camino a la inversa hasta presentar la interfaz con los datos al usuario. Para su acceso se utiliza la librería de **Ibasco**.

Toda la aplicación web con el servidor estaría contenido en una imagen de Docker y la base de datos estaría contenido en otro Docker los cuales están interconectados y sincronizados gracias a un Docker Compose.

5.1 Base de datos

El diseño de la base de datos para almacenar juegos y usuarios se basa en dos tablas principales: una tabla para juegos y otra para usuarios. Esta estructura permite organizar y gestionar eficientemente la información relacionada con los juegos y tener la posibilidad de almacenar varios usuarios para ejecutar las operaciones CRUD de los juegos.

Este sería el diagrama correspondiente a la tabla de juegos:

| games | |
|-------------|--------------|
| id | bigint |
| description | varchar(350) |
| steam_id | int |
| title | varchar(255) |

Figura 5.1: Diagrama tabla ‘games’

Fuente: Propia

Este sería el diagrama correspondiente a la tabla de usuarios:

| users | |
|-----------|--------------|
| id | bigint |
| username | varchar(255) |
| password | varchar(255) |
| enabled | bit(1) |
| role | varchar(255) |

Figura 5.2: Diagrama tabla ‘users’

Fuente: Propia

No he requerido de una complejidad en la base de datos superior, ya que, cierta información como las noticias o las imágenes de los juegos se almacena temporalmente en caché y no en la base de datos.

5.2 Aplicación Web

La aplicación web está construida gracias a Spring. Esta aplicación tiene las siguientes capas:

- **Repositorio:** El repositorio es la capa de acceso a los datos. Es la capa que se va a comunicar con la BBDD y la que hará todas las operaciones CRUD que se requieran hacer.
- **Servicio:** Es la capa donde tendríamos toda la lógica de negocio. Si estuviésemos hablando de una aplicación donde gestionamos los empleados de una empresa y las horas que llevan trabajadas, se implementaría toda la lógica de los empleados de la empresa en esta capa.
- **Façade:** La façade actúa como un intermediario entre el cliente y la aplicación, proporcionando una interfaz para acceder a las funcionalidades de la aplicación. Esta capa oculta la complejidad interna y los detalles de implementación del sistema, presentando solo los datos relevantes para el usuario. Además, la capa de Façade puede realizar tareas adicionales, como la validación de datos o la gestión de errores. En nuestra Façade se accede a la API de Steam y se convierte lo que nos devuelve a **DTOs** propios que encapsulan la información que queremos pasar a los controladores.
- **Controladores:** La capa de controladores se encarga de manejar las solicitudes entrantes y coordinar las acciones necesarias para procesarlas. Su función principal es recibir las peticiones, extraer los datos necesarios y llamar a los componentes correspondientes para procesar la lógica de la aplicación. Además, los controladores pueden realizar validaciones y enviar las respuestas correspondientes al cliente.

Esta sería una descripción gráfica del flujo de una petición a la aplicación:



Figura 6: Flujo de peticiones y respuestas.

Fuente: Propia

5.3 Herramientas

IntelliJ IDEA es una herramienta de desarrollo de software de **JetBrains** con características y funcionalidades avanzadas. Su principal ventaja para desarrollar aplicaciones web profesionales radica en su amplio soporte para tecnologías populares como Spring. Ofrece herramientas de refactorización, depuración y navegación inteligente de código, lo que agiliza el proceso de desarrollo. Además, cuenta con una variedad de complementos y extensiones para adaptarse a las necesidades de cada proyecto.

Para estudiantes tenemos la versión Ultimate de forma gratuita durante 1 año, que proporciona por defecto extensiones y utilidades para el desarrollo de una aplicación web con Spring y sus interfaces asociadas.

Con todo lo mencionado anteriormente, he visto conveniente que la mejor herramienta de desarrollo web es IntelliJ IDEA Ultimate.

6. IMPLEMENTACIÓN DEL DISEÑO

En el desarrollo de una aplicación web, la buena codificación y la implementación de patrones de diseño son aspectos fundamentales para asegurar una solución eficiente y mantenible. Una codificación limpia y bien estructurada facilita la comprensión y el mantenimiento del código a lo largo del tiempo, el uso de patrones de diseño como el patrón MVC o el patrón de repositorio ayuda a separar las responsabilidades y promover la reutilización de código. Estos enfoques permiten construir una API escalable, robusta y fácil de mantener.

En las aplicaciones de Spring unas de las características más importantes son la Inyección de Dependencias, el IoC y los Beans.

Las **beans de Java** son componentes de software reutilizables. Prácticamente, son clases escritas en Java que siguen cierta convención particular. Estos objetos pueden pasarse como un objeto en vez de tener múltiples instancias del mismo.

El **Principio de inyección de dependencia** no es más que poder pasar, inyectar, las dependencias cuando sea necesario en lugar de inicializar las dependencias dentro de la clase receptora. En Spring, la inyección de dependencias se puede hacer de la forma nativa de Spring, con la etiqueta `@Autowired` que se recomienda usar en el constructor de la clase o con la inyección nativa de Java, con la etiqueta `@Resource`. Ambas a efectos prácticos hacen lo mismo, pero si usamos la anotación de Java, no tendremos tanta dependencia de Spring en el proyecto.

La **Inversión de Control** (IoC) es un principio fundamental que permite gestionar las dependencias entre los componentes de manera más flexible y desacoplada. En lugar de que los objetos creen y administren sus propias dependencias, la IoC de Spring se encarga de crear y proporcionar las dependencias necesarias.

Esto implica que los objetos dependen de interfaces en lugar de implementaciones concretas, lo que facilita la sustitución y la prueba de componentes.

6.1 Creación del proyecto de Spring

Para la creación de un proyecto de Spring podemos utilizar herramientas que nos faciliten el trabajo incluyendo las librerías necesarias o de mantener la estructura correcta de ficheros y directorios de una aplicación de este tipo.

La plataforma más famosa para poder llevarlo a cabo es la de **Spring Initializr**, una herramienta en línea que proporciona una interfaz fácil de usar para generar proyectos de Spring con las dependencias y configuraciones necesarias, agilizando el proceso de configuración inicial.

IntelliJ IDEA Ultimate la tiene integrada en su entorno para poder incluir directamente el proyecto generado a nuestro workspace:

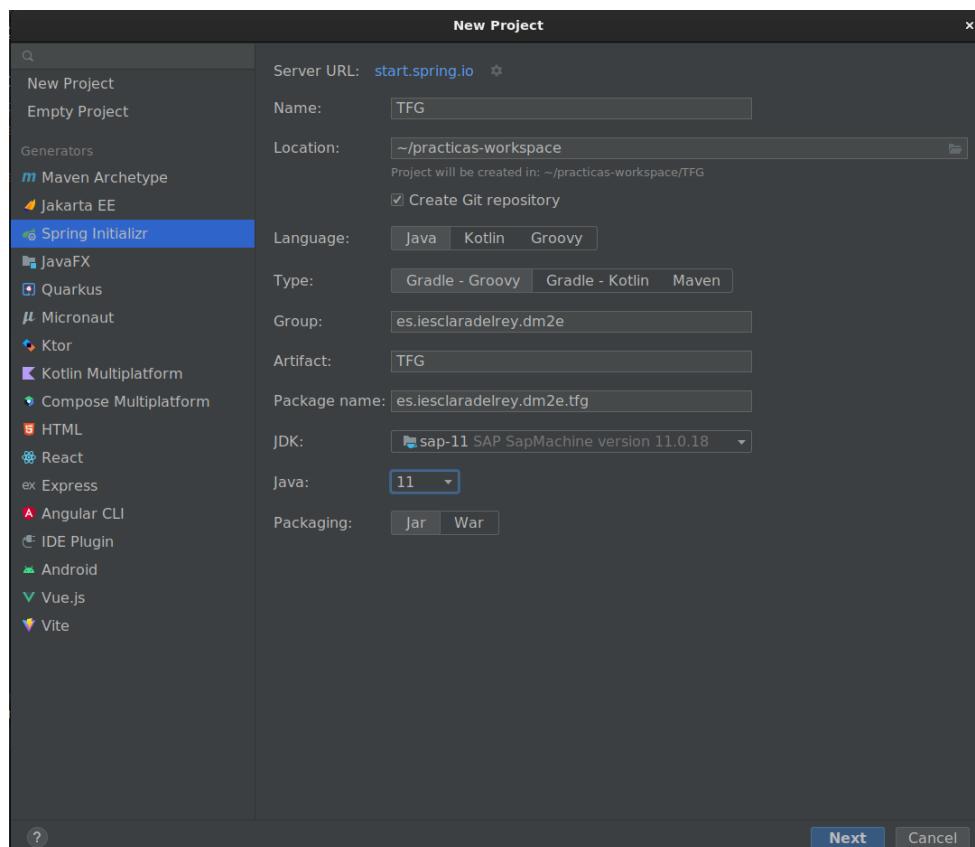


Figura 7.1: Creación del proyecto.

Fuente: Propia

En nuestro caso va a ser un proyecto que usa Gradle como gestor de dependencias ya que voy a crear una tarea para generar posteriormente la imagen de Docker de la aplicación,

va a ser un repositorio de Git para tener un control de versiones y simular cómo sería el trabajo en equipo real y se utilizará la versión de **Java 11 LTS** de **SAP machine**.

A continuación añadiremos las dependencias que usaremos inicialmente:

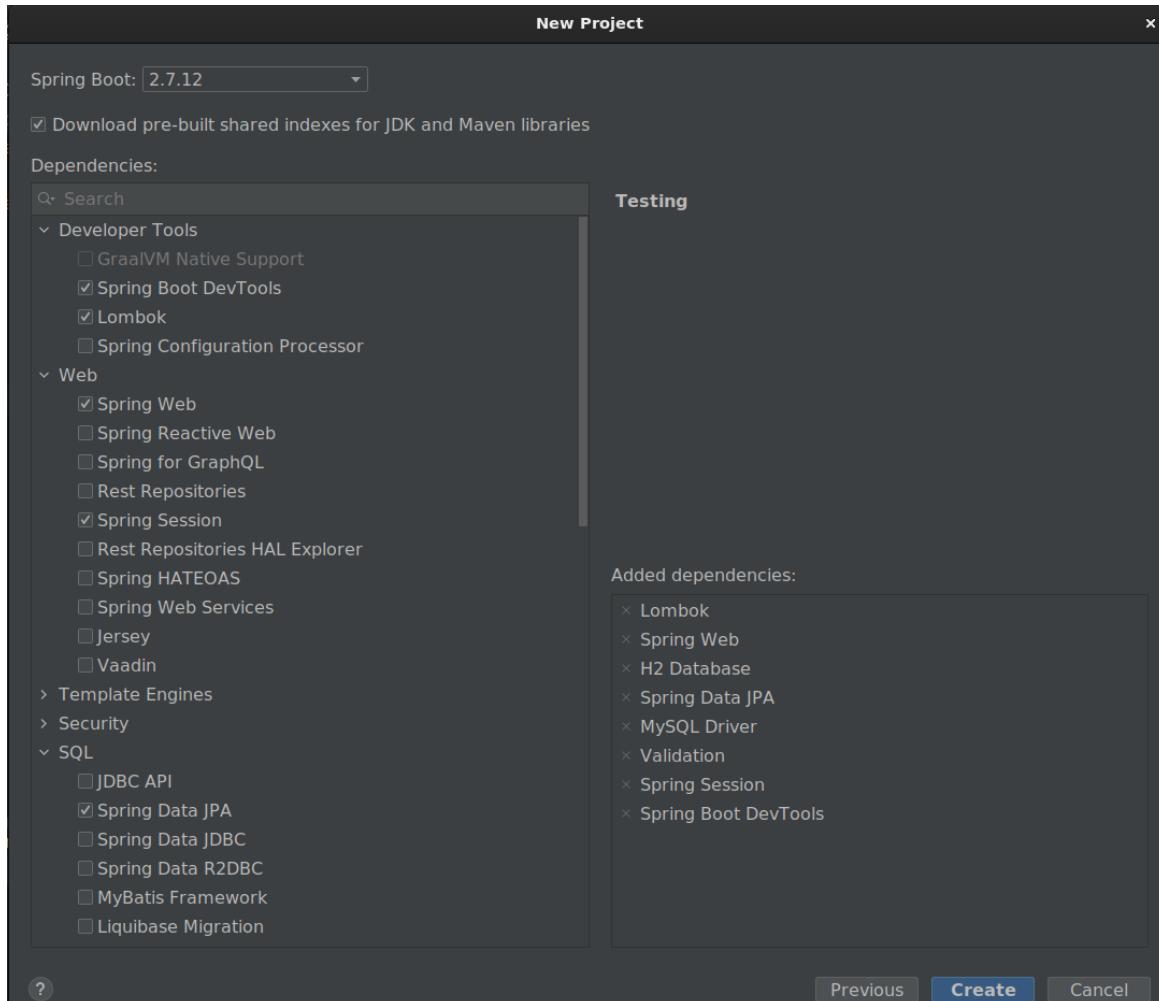


Figura 7.2: Dependencias iniciales.

Fuente: Propia

Estas son las dependencias que tendremos inicialmente y que nos permite el selector del entorno elegir.

Lombok es una biblioteca de Java que reduce la necesidad de escribir código repetitivo al generar automáticamente métodos getter, setter, constructores y otros métodos comunes a través de anotaciones, mejorando así la productividad del desarrollo.

Spring Web es un módulo de Spring Framework que incluye soporte para controladores, enrutamiento, manejo de solicitudes HTTP y vistas, facilitando así la creación de APIs RESTful y aplicaciones web.

H2 Database es un sistema de gestión de bases de datos relacional escrito en Java, que ofrece un rendimiento rápido y ligero y es una opción popular para pruebas de aplicaciones.

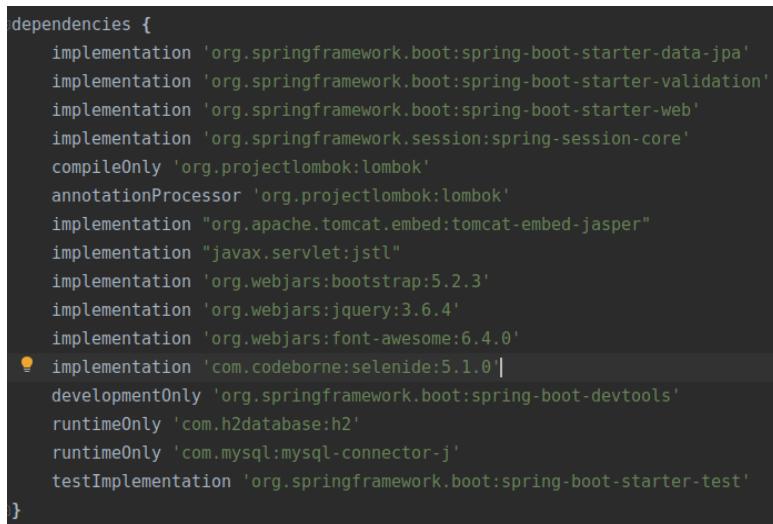
JPA (Jakarta Persistence API) proporciona un conjunto de interfaces y métodos para el mapeo objeto-relacional (ORM), permitiendo interactuar con bases de datos relacionales utilizando objetos Java, simplificando así el acceso y manipulación de datos en aplicaciones.

MySQL Driver es un controlador **JDBC** que permite establecer una conexión entre una aplicación Java y una base de datos MySQL, proporcionando las funcionalidades para enviar consultas y recibir resultados. En el proyecto trabajará en conjunto con JPA.

Spring Validation es un módulo de Spring Framework que ofrece un conjunto de herramientas y anotaciones para validar datos de entrada, permitiendo la validación de formularios y objetos.

Spring Session es un módulo de Spring que proporciona soporte para gestionar y mantener sesiones de usuarios de manera transparente en aplicaciones web, facilitando la gestión y persistencia de la información de sesión de los usuarios.

Para poder tener una aplicación web completa necesitaremos añadir algunas librerías como el servidor Tomcat Embebido o las librerías de JSTL, Bootstrap, JQuery entre otras:



```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-validation'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.session:spring-session-core'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    implementation "org.apache.tomcat.embed:tomcat-embed-jasper"
    implementation "javax.servlet:jstl"
    implementation 'org.webjars:bootstrap:5.2.3'
    implementation 'org.webjars:jquery:3.6.4'
    implementation 'org.webjars:font-awesome:6.4.0'
    implementation 'com.codeborne:selenide:5.1.0' |<-- This line is highlighted
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    runtimeOnly 'com.h2database:h2'
    runtimeOnly 'com.mysql:mysql-connector-j'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

Figura 7.3: Dependencias extra.

Fuente: Propia

Entre las librerías incluidas están las siguientes:

1. **Tomcat Embed** es una biblioteca de Tomcat que permite embeber un servidor web Tomcat dentro de una aplicación Java, lo que facilita la implementación y el despliegue de aplicaciones web sin necesidad de una instalación separada del servidor Tomcat.
2. **Bootstrap** es un framework de desarrollo web que ofrece un conjunto de estilos predefinidos y componentes reutilizables, facilitando la creación de interfaces responsive para aplicaciones web y dispositivos móviles.
3. **jQuery** es una biblioteca de JavaScript que simplifica la manipulación del DOM, permitiendo desarrollar de manera más rápida y sencilla aplicaciones web dinámicas.
4. **Font Awesome** es una biblioteca de iconos que ofrece una amplia variedad de iconos vectoriales listos para usar, permitiendo mejorar la estética y la usabilidad al añadir iconos de alta calidad de manera sencilla y personalizable.
5. **Selenide** es una biblioteca de automatización de pruebas de interfaz de usuario basada en Selenium WebDriver, que simplifica y agiliza el desarrollo de pruebas funcionales en aplicaciones web.
6. **JSTL (Jakarta Standard Tag Library)** es una biblioteca de etiquetas estándar para **JSP (Java Server Pages)** que proporciona una sintaxis simplificada y

reutilizable para el desarrollo de aplicaciones web Java, facilitando la separación de la lógica de presentación en las páginas JSP.

Una vez tenemos las dependencias necesarias podemos empezar a codificar.

6.2 Creación de los modelos

Para poder almacenar, organizar y manipular la información de la aplicación necesitamos representar las estructuras de tablas de la base de datos en el lenguaje de programación que estemos utilizando. Para eso tenemos que crear modelos de datos.

Los modelos en Java se corresponden a los POJOs, son objetos de Java que simplemente tienen propiedades, los getters y setters correspondientes, los constructores y posibles métodos como ‘*equals()*’ y ‘*hashCode()*’ para poder compararlos con otros u obtener algunas funcionalidades extra.

Para poder utilizar estos modelos de datos para acceder a la base de datos, vamos a usar anotaciones de JPA que, entre otras cosas, le permitirán traducir las instrucciones de Java al lenguaje SQL o las propiedades en atributos de la tabla. Estas estructuras a partir de este punto empezarán a llamarse entidades.

Esta sería la entidad ‘game’:

```
# Jorge Moreno Brasero
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@EqualsAndHashCode
@Entity(name = "games")
public class GameModel implements Serializable {

    private static final int DESCRIPTION_CHARACTERS_LENGTH = 350;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "title", nullable = false)
    private String title;

    @Column(name = "description", length = DESCRIPTION_CHARACTERS_LENGTH)
    private String description;

    @Column(name = "steam_id")
    private Integer steamId;

}
```

Figura 8.1: Modelo de datos ‘*GameModel*’.

Fuente: Propia

Las anotaciones que encontramos en la parte superior corresponden a la librería de Lombok. Esta librería nos genera código como getters y setters automáticamente sin tener que declararlo.

La anotación `@Entity` corresponde a la declaración de JPA de que esta clase es una entidad.

La anotación `@Column` le indica a JPA que cada atributo es una columna de la tabla ‘games’ con la estructura que está declarada como el nombre o la longitud del campo.

La anotación `@Id` le indica a JPA que ese campo va a ser la ‘primary key’ de la tabla. La anotación `@GeneratedValue` indica que este campo se generará automáticamente y la estrategia que va a seguir en este caso es el ‘`auto_increment`’ de MySQL.

Este es el modelo de usuarios:

```
30 usages  ± Jorge Moreno Brasero
-@Entity(name = "users")
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class UserModel {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "username", nullable = false)
    private String username;
    @Column(name = "password", nullable = false)
    private String password;
    @Column(name = "role", nullable = false)
    private String role;
    @Column(name = "enabled")
    private Boolean enabled;

}
```

Figura 8.2: Modelo de datos ‘UserModel’.

Fuente: Propia

6.3 Creación de la capa de repositorios

Para poder hacer cualquier operación con la base de datos debemos de crear una clase que acceda a ella. Para ello podemos utilizar el **patrón de Repositorio**. Esto nos permitirá abstraer la lógica de acceso y la manipulación de los datos, proporcionándonos un punto centralizado de interacción con la base de datos. Esto facilita la gestión y mantenimiento de su código.

En nuestro caso vamos a usar el repositorio de JPA que viene con Spring:

```
2 sages - Jorge Moreno Brasero
public interface GameDao extends JpaRepository<GameModel, Long> {
    List<GameModel> findByTitleContainingIgnoreCase(String title);
}
```

Figura 9.1: Repositorio de juegos.

Fuente: Propia

En los repositorios si necesitáramos funcionalidad extra que no traiga el Repositorio de JPA podemos crear un repositorio propio en el que tengamos el acceso específico. Este repositorio podría utilizar la clase *CriteriaBuilder* para construir la consulta y usar la clase *EntityManager* para poder ejecutar la sentencia SQL contra el SGBD.

En nuestro caso, necesitaré una consulta que tenga la condición de buscar por título. Para conseguir esta consulta he utilizado un mecanismo del *JpaRepository* que se llama **naming convention**. Esto nos permite que mediante el uso de una convención de Spring a la hora de declarar el método en la interfaz, la implementación de éste se encargará Spring de hacerla por ti.

Este es el repositorio de los usuarios:

```
public interface UserDao extends JpaRepository<UserModel, Long> {  
  
    List<UserModel> findByUsernameContainingIgnoreCase(String username);  
  
    Optional<UserModel> findByUsernameIgnoreCase(String username);  
}
```

Figura 9.2: Repositorio de usuarios.

Fuente: Propia

6.4 Creación de la capa de servicios

Tras tener ya el acceso a la BBDD, podemos hacer la capa de la lógica de negocio.

En este proyecto no hay ninguna lógica de negocio implementada, pero si se requiriese implementar podríamos hacerla en esta capa.

Para la creación de la capa de servicio tenemos implementado el **patrón de Abstracción**. Este patrón nos permite separar cada implementación específica de un mismo servicio sin afectar la interfaz pública de la API, lo que facilita el mantenimiento y la evolución del sistema a largo plazo. Esto brinda flexibilidad y modularidad a la aplicación.

Esta es la interfaz de la capa de servicio que tienen los juegos:

```
6 usages 1 implementation ✎ Jorge Moreno Brasero
public interface UserService {

    1 implementation ✎ Jorge Moreno Brasero
    UserModel save(UserModel user);

    1 implementation ✎ Jorge Moreno Brasero
    Optional<UserModel> findById(Long id);

    1 implementation ✎ Jorge Moreno Brasero
    List<UserModel> findAll();

    1 implementation ✎ Jorge Moreno Brasero
    void delete(UserModel user);

    1 usage 1 implementation ✎ Jorge Moreno Brasero
    List<UserModel> findByUsernameContaining(String username);

    2 usages 1 implementation ✎ Jorge Moreno Brasero
    Optional<UserModel> findByUsername(String username);
}
```

Figura 10.1: Interfaz del servicio.

Fuente: Propia

Como he explicado antes he utilizado el patrón de Abstracción, esta es la interfaz donde se declara la funcionalidad que tendrá esta capa.

Esta es la implementación de la funcionalidad:

```
@Service
public class GameServiceImpl implements GameService {

    5 usages
    @Resource
    private GameDao gameDao;

    ▲ Jorge Moreno Brasero
    @Override
    public List<GameModel> findAll() { return gameDao.findAll(); }

    ▲ Jorge Moreno Brasero
    @Override
    public Optional<GameModel> findById(Long id) {
        return gameDao.findById(id);
    }

    ▲ Jorge Moreno Brasero
    @Override
    public void delete(Long id) { gameDao.deleteById(id); }

    ▲ Jorge Moreno Brasero
    @Override
    public GameModel save(GameModel game) { return gameDao.save(game); }

    4 usages ▲ Jorge Moreno Brasero
    @Override
    public List<GameModel> findByTitle(String title) {
        return gameDao.findByTitleContainingIgnoreCase(title);
    }

}
```

Fuente 10.2: Implementación del *GameService*.

Fuente: Propia

6.5 Creación de la capa Façades

Cuando tenemos nuestra lógica de negocio programada, debemos encapsular los datos de los modelos en objetos llamados *DTOs*. Esta encapsulación se debe a que en ocasiones no queremos mostrar ciertos datos al usuario como el id de un juego o la contraseña de un usuario.

La capa Façades es la encargada de convertir los datos ‘brutos’ en los ‘procesados’ que deseemos. Además podemos hacer procesamiento de errores como veremos con los errores de validación de los usuarios.

En esta capa se utiliza para hacer las acciones que he mencionado unas clases llamadas *Converters* estas clases son las encargadas de transformar un objeto del tipo modelo a un objeto del tipo DTO y a la inversa.

Aquí tenemos un ejemplo de *Converter*:

```
@Component
public class GameModelToGameDtoConverter implements Converter<GameModel, GameDto> {

    @Override
    public GameDto convert(final GameModel source) {
        Assert.notNull(source, "The source object must not be null");
        return GameDto.builder()
            .id(source.getId())
            .title(source.getTitle())
            .description(source.getDescription())
            .steamId(source.getSteamId())
            .build();
    }
}
```

Figura 11: Converter de *GameModel* a *GameDto*.

Fuente: Propia

Aquí tenemos una implementación de Façade:

```
▲ Jorge Moreno Brasero
@Override
public void delete(Long id) { gameService.delete(id); }

▲ Jorge Moreno Brasero
@Override
public GameDto save(GameDto game) {
    return modelToGameDtoConverter.convert(gameService.save(dtoToGameModelConverter.convert(game)));
}

1 usage ▲ Jorge Moreno Brasero +1
@Override
@Cacheable(value = "imageCache", key = "#steamId")
public GameSteamData getGameDetails(Integer steamId, Locale locale) {
    return appDetailsToGameSteamData.convert(steamStorefront.getAppDetails(steamId,
        locale.getDisplayCountry().toLowerCase(),
        locale.getDisplayLanguage()
            .toLowerCase())
    .join());
}
```

Figura 12: Fragmento de la implementación de *GameFaçade*.

Fuente: Propia

La Façade en nuestra implementación utiliza la API de Steam, por lo que los modelos que nos suministran los métodos de cada servicio de la librería de Ibasco los convertimos en nuestros propios DTOs.

Como se puede observar en la imagen, para obtener los detalles del juego por el ID está cacheado, esto se debe a que tenemos que hacer una petición a la API de Steam mencionada anteriormente. Este proceso puede tardar bastante tiempo y para asegurarnos de no tener una página lenta y tampoco hacer demasiadas peticiones y tener el riesgo a que nos banee, tenemos la caché implementada.

La caché la he configurado con Ehcache. El fichero de configuración es un XML donde declaras ciertas directivas como el tamaño máximo de la caché o el tiempo que va a estar cacheados esos datos:

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.ehcache.org/v3"
    xsi:schemaLocation="http://www.ehcache.org/v3 https://www.ehcache.org/schema/ehcache-core-3.0.xsd">

    <cache alias="newsCache">
        <key-type>java.lang.Integer</key-type>
        <value-type>java.util.ArrayList</value-type>
        <expiry>
            <ttl unit="hours">2</ttl>
        </expiry>

        <resources>
            <offheap unit="MB">10</offheap>
        </resources>
    </cache>
```

Figura 13: Configuración de Ehcache.

Fuente: Propia

En esta configuración podemos tener distintas directrices dependiendo de las necesidades que tenga para cada dato que esté almacenado en la caché.

Aquí un ejemplo de DTO que contiene datos de Steam:

```
@Getter
@Setter
@Builder
public class GameSteamNewsData implements Serializable {

    private String title;
    private String url;
    private String author;
    private String contents;

}
```

Figura 14: DTO de las noticias de Steam.

Fuente: Propia

Aquí un ejemplo de DTO propio de nuestra aplicación web:

```
@Builder  
@Getter  
@Setter  
@NoArgsConstructor  
@AllArgsConstructor  
@UniqueUsername(message = "...")  
public class UserDto {  
  
    private Long id;  
    @NotEmpty(message = "...")  
    private String username;  
    @NotEmpty(message = "...")  
    private String password;  
    private String role;  
  
}
```

Figura 15: ‘UserDto’.

Fuente: Propia

Los DTOs de los datos propios de la aplicación web tienen las anotaciones `@NotEmpty` o `@UniqueUsername` para la validación de formularios. Esta evaluación corresponde a la validación **JSR 303**.

La validación JSR 303, también conocida como Bean Validation, proporciona una forma estandarizada de validar datos en objetos Java. Define un conjunto de anotaciones y reglas para la validación de campos y propiedades en clases Java, lo que permite garantizar la integridad de los datos ingresados en una aplicación. Algunas de las ventajas del uso de la validación JSR 303 son la simplicidad de implementación, ya que se basa en anotaciones fáciles de entender y aplicar en las clases, y la reutilización de código, ya que la validación se puede aplicar de manera uniforme en diferentes capas de la aplicación.

Para el procesamiento de los errores de esta validación se hace en esta capa. Aquí una implementación:

```
@Component
public class AuthenticationErrorFacadeImpl implements ValidationErrorFacade {

    @usage  ↗ Jorge Moreno
    @Override
    public Map<String, Set<String>> getErrorsByField(BindingResult result) {
        return Stream.concat(result.getAllErrors() .List<ObjectError>
            .stream() .Stream<ObjectError>
            .filter(FieldError.class::isInstance)
            .map(FieldError.class::cast) .Stream<FieldError>
            .collect(Collectors.toMap(FieldError::getField,
                fieldError -> Set.of(Objects.requireNonNull(fieldError.getDefaultMessage())))) .Map<String, Set<String>>
            .entrySet() .Set<Entry<...,>>
            .parallelStream(),
        result.getAllErrors() .List<ObjectError>
            .stream() .Stream<ObjectError>
            .filter(objectError -> !(objectError instanceof FieldError))
            .filter(objectError -> Objects.equals(objectError.getCode(), "UniqueUsername"))
            .collect(Collectors.toMap(objectError -> "username",
                objectError -> Set.of(Objects.requireNonNull(objectError.getDefaultMessage()))))
            .entrySet() .Set<Entry<...,>>
            .stream())
        .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
    }
}
```

Figura 16: Implementación del procesamiento de errores del formulario de registro.

Fuente: Propia

6.6 Creación de la capa de controladores

Tras tener los datos de los objetos encapsulados y la gestión de errores hecha correctamente, debemos crear los puntos donde nuestro servidor web va a responder dependiendo de la URL y el método HTTP utilizado en la petición.

Esta capa depende directamente de las interfaces, por lo que la creación de la misma se corresponde en 2 fases. La creación de los propios controladores y la generación de las páginas asociadas.

6.6.1 Creación de las páginas

Las páginas son las encargadas de mostrar los datos que le pasemos desde los controladores con la estructura que definamos en ellas.

6.6.1.1 Pagina principal

En mi caso para la página principal he cogido una **plantilla de Bootstrap** y la he modificado para que no tenga que hacer todo el código desde 0 y quede lo más profesional posible.

En esta plantilla tenemos un menú de navegación donde encontramos los botones de login y signup además de una barra de búsqueda para poder buscar por título. Tenemos una sección donde se ve una imagen difuminada con el título de la página y una pequeña descripción. A continuación, tenemos unas tarjetas donde se muestra la carátula del juego con su título.

Tengo implementada paginación para mostrar los videojuegos en la aplicación, esto nos permite no tener que cargar todos los datos de golpe y si tenemos muchos datos en el futuro esto mejorará el rendimiento. La paginación viene implementada en el repositorio de JPA por defecto, simplemente tenemos que pasar como parámetros de la URL el número de página en el que estamos y, si queremos una cantidad distinta a la por defecto, la cantidad de elementos por página. Veremos más adelante cómo se maneja en el controlador.

Si el usuario está autenticado tendrá un botón del tipo dropdown donde se muestran las opciones de editar y borrar. La opción de borrar nos mostrará una ventana emergente de confirmación de borrado. Tanto el botón de dropdown como la ventana modal están hechas con Bootstrap.

Para poder eliminar el juego que queremos en la ventana modal se ha utilizado jQuery para poner la URL correcta en el botón de confirmación. En él extraemos datos del juego gracias a los **data de las modales** de Bootstrap.

Este es el script:

```
$(document).ready(function () {
    $('#confirm-delete').on('show.bs.modal', function (event) {
        let gameId = $(event.relatedTarget).data('bs-game-id');
        let gameTitle = $(event.relatedTarget).data('bs-game-title');
        $(this).find('.modal-body p span').text(gameTitle);
        $('#confirm-button').attr('href', '/game/delete/' + gameId);
    });
});
```

Figura 17: Modal de Bootstrap dinámica.

Fuente: Propia

Para finalizar la interfaz tenemos unos botones que corresponden al avance o retroceso de las páginas y debajo de estos botones un footer sencillo.

La página principal se ve tal que así:

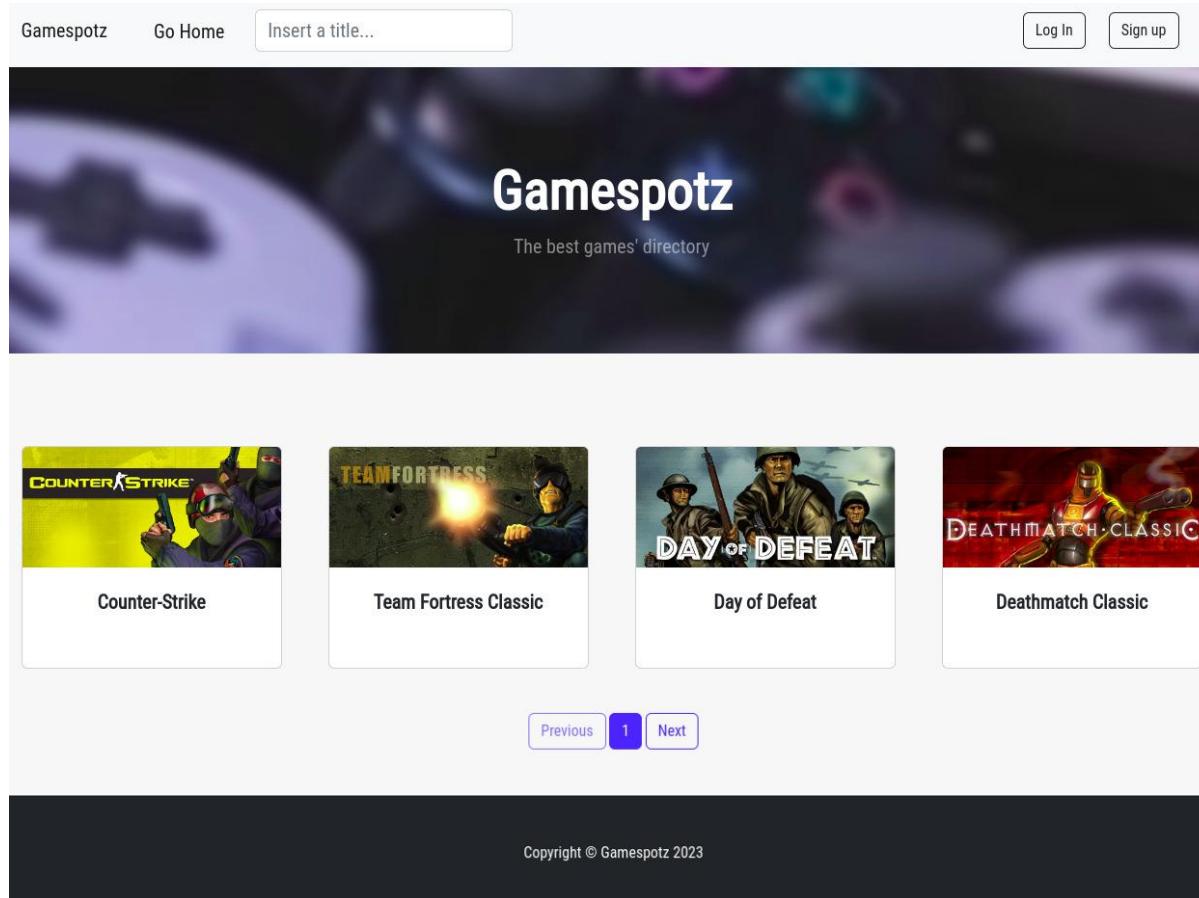


Figura 18.1: Pantalla principal.

Fuente: Propia

La tarjeta de un juego estando autenticado se vería de esta forma:

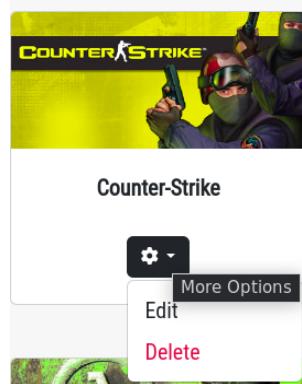


Figura 18.2: Juego estando autenticado

Fuente: Propia

Para conseguirlo, al ser una página dinámica, le pasamos desde el controlador la lista de juegos que deseamos mostrar y la recorremos en la interfaz. Así se ve el JSP:

```
<c:forEach var="game" items="${games.content}">
    <div class="col mb-5">
        <div class="card h-100">
            <img class="card-img-top img cover" src="" alt="${game.title}'s cover"/>
            <span id="game-id" hidden="hidden">${game.id}</span>
            <span class="steam-id" hidden="hidden">${game.steamId}</span>
            <div class="card-body p-4">
                <div class="text-center">
                    <h5 class="fw-bolder">${game.title}</h5>
                </div>
            </div>
            <div class="card-footer p-4 pt-0 border-top-0 bg-transparent text-center">
                <sec:authorize access="hasAuthority('REGISTERED')">
                    <div class="dropdown">
                        <button type="button" class="btn btn-outline-dark dropdown-toggle"
                            data-bs-toggle="dropdown" aria-expanded="false" title="More Options"
                            id="options-dropdown">
```

Figura 19: Fragmento del código de la página principal.

Fuente: Propia

En este fragmento encontramos la etiqueta `<c:forEach>` que lo que hace es recorrer la lista que le pasamos desde el controlador. Obtenemos la lista con el atributo `'items=" ${games.content} "'` y generamos una variable `'var="game"'` que corresponderá a cada juego. Con esta variable podemos obtener los atributos a los que tenemos acceso concatenando `.atributo`.

En la parte inferior de la foto encontramos la etiqueta `<sec:authorize>`, es una etiqueta de Spring Security que nos permite ocultar ciertas partes de la página en ciertas ocasiones. En este caso con el atributo `access="hasAuthority('REGISTERED')"` le limitamos el botón de dropdown. Este botón se mostrará únicamente cuando estemos autenticados.

Para que cada juego tenga su carátula, tengo un script de JavaScript que usa jQuery para llenar las tarjetas y que ejecuta una petición AJAX para obtener las carátulas desde la aplicación:

```
$document).ready(function () : void {
    let steamIds = []
    $('.steam-id').each(function () : void {
        steamIds.push(parseInt($(this).text()))
    })
    $.ajax({
        url: "http://localhost:8080/game/steam/covers",
        type: "GET",
        data: {steamIds: decodeURI(steamIds)},
        success: function (response) : void {
            let covers = []
            response.forEach(element => {
                covers.push(element.coverUrl)
            })
            for (let indexCover = 0; indexCover < covers.length; indexCover++) {
                $('#card').get(indexCover).find('.cover').attr('src', covers.at(indexCover)).show()
            }
        }
    });
});
```

Figura 20: Script para insertar las carátulas.

Fuente: Propia

6.6.1.2 Pagina por juego

Si pinchamos en una tarjeta podemos ir a la información detallada del juego. Si este está enlazado con un ID de Steam válido se consultará la API de Steam y se mostrará una captura del juego además de sus últimas noticias.

Así es como se vería un juego con un ID de Steam válido:

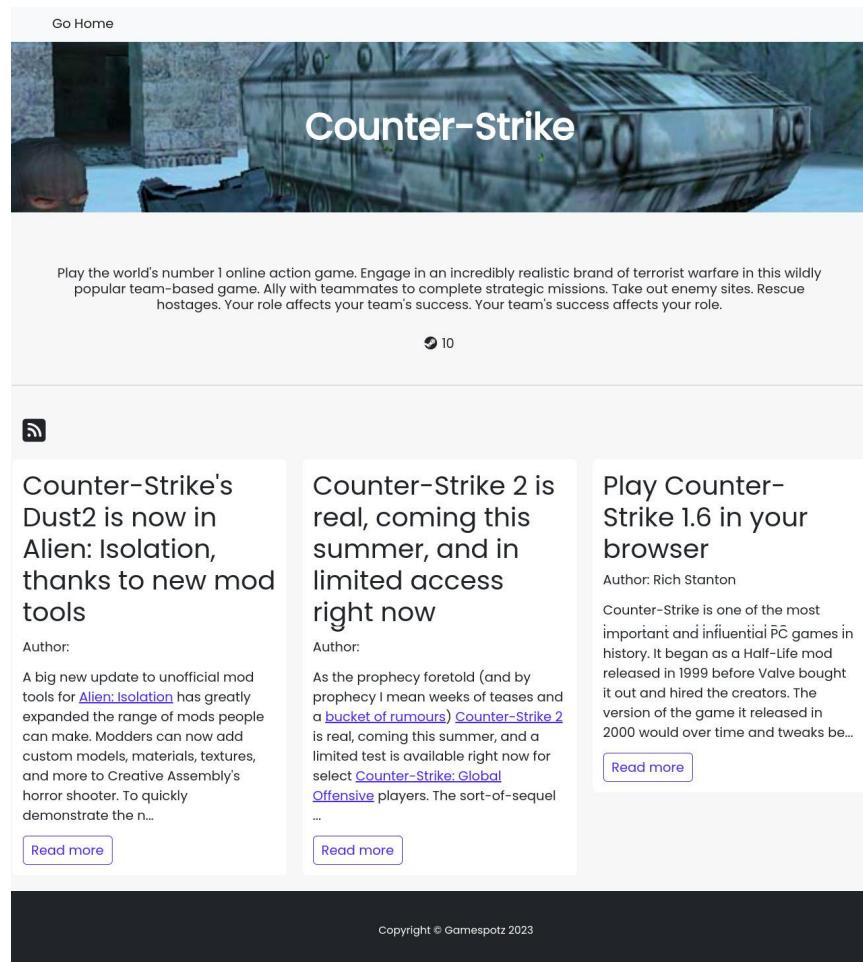


Figura 21: Vista por juego.

Fuente: Propia

En esta página tengo un control de si el juego existe en la base de datos. Esto se puede conseguir de esta forma:

```
<header class="bg-dark py-5" id="image-top">
  <div class="container px-4 px-lg-5 my-5">
    <div class="text-center text-white">
      <c:choose>
        <c:when test="${empty game}">
          <h1 class="display-4 fw-bolder"><spring:message code="game.not.found.error"/></h1>
        </c:when>
        <c:otherwise>
          <h1 class="display-4 fw-bolder">${game.title}</h1>
        </c:otherwise>
      </c:choose>
    </div>
  </div>
</header>
```

Figura 22: Fragmento de la página por juegos.

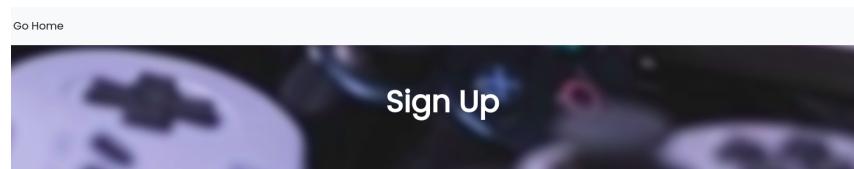
Fuente: Propia

Encontramos la etiqueta `<c:choose>` la cual nos permitirá hacer un if-else o un if-else-if-else. Esto nos da la posibilidad de mostrar una información u otra en función de una condición. En este caso si el juego no existe se muestra un mensaje de error como título de la interfaz, sino se muestra el título del juego. La etiqueta `<c:when>` es la que valora la condición, en este caso, si está vacío entra en ese bloque de código y sino entra en el `<c:otherwise>` u otra etiqueta `<c:when>` que esté debajo.

La etiqueta `<spring:message>` hace que el mensaje que se muestra esté internacionalizado, es decir, cogerá el valor de ese código en función del lenguaje en el que esté configurado el cliente. Esto es posible gracias a la librería de i18n.

6.6.1.3 Pagina de registro

Para poder tener usuarios necesitamos poder darlos de alta, para ello necesitamos una interfaz específica que sea un formulario de registro:



The screenshot shows a registration form titled "Sign Up". At the top left is a "Go Home" link. Below the title are two input fields: "Username:" and "Password:", each with a corresponding text input box. At the bottom center is a blue "Register now" button. A small copyright notice "Copyright © Gamespotz 2023" is visible at the very bottom of the page.

Figura 23: Página de registro.

Fuente: Propia

En esta interfaz utilizamos el siguiente fragmento para el formulario:

```
<c:url var="register" value="/sign-up"/>
<form:form method="POST" action="${register}" modelAttribute="user">
    <div class="form-group mt-4">
        <form:label cssClass="display-6" path="username"><spring:message
            code="signup.username.label"/></form:label>
        <form:input path="username" cssClass="form-control form-control-lg" id="username"/>
        <c:if test="${not empty usernameError}">
            <c:forEach items="${usernameError}" var="error">
                <span class="error">${error}</span>
            </c:forEach>
        </c:if>
    </div>
    <div class="form-group mt-4">
        <form:label path="password" cssClass="display-6"><spring:message
            code="signup.password.label"/></form:label>
        <form:password path="password" class="form-control form-control-lg" id="password"/>
        <c:if test="${not empty passwordError}">
            <c:forEach items="${passwordError}" var="error">
                <span class="error">${error}</span>
            </c:forEach>
        </c:if>
```

Figura 24: Fragmento de la página de registro.

Fuente: Propia

Utilizo la etiqueta de JSTL de formularios para poder pasar los datos como si fuese un objeto de la clase *UserModel*. Las etiquetas *label* e *input* son similares a las etiquetas de formularios de HTML.

Deabajo de cada *input* tenemos un *span* que mostrará los errores que se hayan encontrado con el procesamiento de errores custom de la façade.

6.6.1.4 Pagina de login

La página de login es similar a la de registro pero tiene una singularidad:

```
<c:url var="login" value="/login"/>
<form:form method="post" action="${login}">
    <sec:csrfInput/>
    <div class="form-group mt-4">
        <label for="username" class="display-6"><spring:message code="login.username.label"/></label>
        <input type="text" class="form-control form-control-lg" id="username"
            placeholder=<spring:message code="login.username.placeholder"/> name="username">
```

Figura 25: Fragmento de la página del login.

Fuente: Propia

La etiqueta `<sec:csrfInput>` es necesaria en el formulario de login. Esto se debe a que Spring Security tiene implementado seguridad frente a ataques a través de **CSRF**. Con este tag generamos un input de CSRF para poder autenticarnos correctamente y así no tener bloqueos con esta protección.

6.6.1.5 Pagina de creación y edición de juegos

La página de creación y edición de los juegos es similar, por lo que la trataré como una sola. La particularidad de estas páginas es que la gestión de los errores se hace con la implementación que viene por defecto:

```
<c:url var="editGame" value="/game/edit/${game.id}" />
<form:form method="post" action="${editGame}" modelAttribute="game">
    <div class="form-group">
        <label class="control-label col-sm-2 display-6" for="title"><spring:message
            code="title.form.label"/></label>
        <div class="col-sm-10">
            <form:input path="title" id="title" cssClass="form-control" value="${game.title}" />
            <form:errors path="title" cssClass="error"/>
        </div>
    </div>
```

Figura 26.1: Fragmento de la página de edición.

Fuente: Propia

Con la validación JSR 303, cuando haya un error de validación, directamente se mostrará en la etiqueta `<form:errors>` como si fuese un span debajo de la etiqueta input.

La página se ve tal que así:

The screenshot shows a web form titled "Enter The Game Details". At the top left is a "Go Home" link. The main title is centered over a blurred background image of a video game controller. Below the title are three input fields: "Title:", "Description:", and "Steam Id:", each with a corresponding text input box. A "Save Changes" button is located below the Steam Id field. At the bottom of the page is a dark footer bar with the copyright notice "Copyright © Gamespotz 2023".

Figura 26.2: Pantalla de edición de juegos.

Fuente: Propia

6.6.2 Creación de los controladores

Para que el servidor responda a ciertas peticiones que le hagamos y muestre las interfaces, debemos de crear controladores, que serán los encargados de manejar esas peticiones en función de la URL y el método HTTP utilizado para ejecutar una acción u otra.

Aquí encontramos dos métodos similares que muestran las interfaces de inicio y edición de un juego:

```
    @GetMapping("/")
    public String home(Model model, @PageableDefault(size = 20) Pageable pageable) {
        model.addAttribute("games", gameFacade.findAll(pageable));
        return "index";
    }

    ✎ Jorge Moreno Brasero
    @GetMapping("/{id}")
    public String showById(@PathVariable Long id, Model model) {
        gameFacade.findById(id)
            .ifPresent(gameDto -> model.addAttribute("game", gameDto));
        return "gameById";
    }
```

Figura 27.1: Fragmento del controlador de juegos.

Fuente: Propia

En la imagen tenemos la etiqueta `@GetMapping`, que hace que el método responda a las peticiones GET que estén dirigidas a las URL `'/'` y `'/game/{id}'` con cualquier id. La etiqueta `@PathVariable` hace que la parte de la URL que corresponda a donde está `{id}` se inyecte directamente en el parámetro del método.

La etiqueta `@PageableDefault` hace que la configuración por defecto de la paginación sea de una forma determinada, en este caso, con tamaño de 20 elementos por página. La paginación la conseguimos gracias al objeto `Pageable` del parámetro del método. En este objeto se almacenan los parámetros de la URL relacionados con la paginación.

Aquí encontramos un ejemplo de la validación JSR 303 y una redirección:

```
@PostMapping("game/edit/{id}")
public String showUpdatedGame(@Valid @ModelAttribute("game") GameDto game,
                                BindingResult result,
                                RedirectAttributes redirectAttributes,
                                Locale locale) {
    if (result.hasErrors()) {
        return "editGame";
    }
    redirectAttributes.addFlashAttribute("message", messageSource.getMessage("update.confirmation.workflow",
            new String[]{game.getTitle()}, locale));
    redirectAttributes.addFlashAttribute("type", MESSAGE_TYPE_SUCCESS);
    return "redirect:/game/" + gameFacade.save(game).getId();
}
```

Figura 27.2: Fragmento del controlador de los juegos.

Fuente: Propia

Este método con la anotación `@PostMapping` el controlador responderá a las peticiones con el método HTTP *POST* a la URL especificada. En el método encontramos la validación con la anotación `@Valid`. El objeto `BindingResult` recogerá los errores relacionados con la validación.

Para la redirección necesitamos pasar ciertos atributos como el mensaje que se va a mostrar en la siguiente interfaz tras modificar el juego para que el usuario sepa cómo va el flujo de la aplicación. Esto lo conseguimos con el objeto `RedirectAttributes`.

Además de todo esto, para que el mensaje del flujo del programa esté internacionalizado se captura el idioma del cliente con el objeto `Locale` que está como parámetro del método.

Aquí tenemos 2 métodos que devuelven como respuesta un JSON:

```
@GetMapping(value = "/game/steam/news/{steamId}", produces = MediaType.APPLICATION_JSON_VALUE)
@ResponseBody
public List<GameSteamNewsData> getSteamNews(@PathVariable Integer steamId) {
    return gameFacade.getSteamNews(steamId);
}

▲ Jorge Moreno
@GetMapping(value = "/game/steam/covers", produces = MediaType.APPLICATION_JSON_VALUE)
@ResponseBody
public List<GameCoverData> getGameCover(@RequestParam("steamIds") List<Integer> steamIds, Locale locale) {
    return gameFacade.getAllCovers(steamIds, locale);
}
```

Figura 27.3: Fragmento del controlador de juegos.

Fuente: Propia

Con la etiqueta `@ResponseBody` conseguimos que el controlador no intente dirigirnos a una página y directamente genere el JSON a partir del objeto que devuelve el método. Estos métodos se utilizan en los scripts de JavaScript que llenan la página principal o la página por juego.

Debido al aumento de la popularidad del uso de microservicios con el fin de reducir las aplicaciones con estructuras monolíticas consiguiendo así aplicaciones más modularizadas y confiables, además del uso de front ends desacoplados como **React** o **Angular**, me he visto en la necesidad de aprender cómo se crean controladores REST.³

Aquí un ejemplo de un método para guardar un usuario:

```
@PostMapping("/user")
public ResponseEntity<List<String>> save(@RequestBody UserDto user, BindingResult result) {
    try {
        userValidator.validate(user, result);
        if (result.hasErrors()) {
            return ResponseEntity.badRequest().body(result.getAllErrors() .stream()
                .map(ObjectError::toString) Stream<String>
                .collect(Collectors.toList()));
        }
        userFacade.save(user);
        return ResponseEntity.status(HttpStatus.CREATED).build();
    } catch (Exception ex) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}
```

Figura 28: Fragmento del controlador REST de usuarios.

Fuente: Propia

En este ejemplo el método está respondiendo a las peticiones *POST* que se mandan a la URL especificada y la petición requiere de un body en forma de JSON válido con la estructura del objeto *UserDto*.

Los métodos de los controladores REST deben devolver objetos del tipo *ResponseEntity* donde se especifica el código de la petición y un body si es necesario.

En este método a su vez, tenemos validación con el objeto *userValidator*, que es un validador que nos permite comprobar todas las validaciones que tengamos en nuestro DTO aplicadas con las anotaciones de JSR 303 o las etiquetas custom que hayamos creado. Los mensajes de error de validación están internacionalizados como en los formularios de login y registro.

7. PRUEBAS

7.1 Plan de pruebas

Dentro de las pruebas que haremos en el proyecto encontramos los tests unitarios, los tests de integración y los tests end-to-end. Los 2 primeros se harán gracias a JUnit y el último utilizaremos la librería Selenide.

Para los servicios haré tests de integración. Esto se debe a que nuestros repositorios extienden del de JPA por lo que la funcionalidad de los mismos está asegurada. Necesitamos saber simplemente si los servicios se conectan correctamente con el repositorio y acceden a la base de datos.

En los Converters haré tests unitarios, para comprobar que realmente se están encapsulando los datos de los modelos en los DTOs.

Para finalizar, haré tests end-to-end para probar que las interfaces funcionan correctamente, tanto que la página principal se visualiza como que el registro, login y el logout funcionan correctamente.

7.2 Resultado de las pruebas

En los test de integración del servicio encontramos el siguiente resultado:

GameServiceImpl

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods |
|---|---------------------|------|-----------------|------|--------|------|--------|-------|--------|---------|
| • findByTitle(String, Pageable) | | 0% | | n/a | 1 | 1 | 1 | 1 | 1 | 1 |
| • save(GameModel) | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| • findAll(Pageable) | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| • findById(Long) | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| • delete(Long) | | 100% | | n/a | 0 | 1 | 0 | 2 | 0 | 1 |
| • findByTitle(String) | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| • findAll() | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| • GameServiceImpl() | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| Total | 6 of 39 | 84% | 0 of 0 | n/a | 1 | 8 | 1 | 9 | 1 | 8 |

Figura 29.1: Resultado tests al servicio.

Fuente: Propia

Como se puede ver en la imagen, hay un 84% de cobertura de código. Todos los tests se ejecutan satisfactoriamente.

En los tests unitarios de los converters encontramos los siguientes resultados:

GameModelToGameDtoConverter

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods |
|-------------------------------|---------------------|------|-----------------|------|--------|------|--------|-------|--------|---------|
| convert(GameModel) | [green bar] | 100% | | n/a | 0 | 1 | 0 | 7 | 0 | 1 |
| GameModelToGameDtoConverter() | [green bar] | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| Total | 0 of 21 | 100% | 0 of 0 | n/a | 0 | 2 | 0 | 8 | 0 | 2 |

Figura 29.2: Resultado tests del converter de *GameModel* a *DTO*.

Fuente: Propia

En estos tests encontramos una cobertura del 100% donde todos los tests se ejecutan satisfactoriamente. El resultado del Converter de DTO a *GameModel* tiene la misma cobertura y el mismo resultado.

En los tests end-to-end encontramos los siguientes resultados:

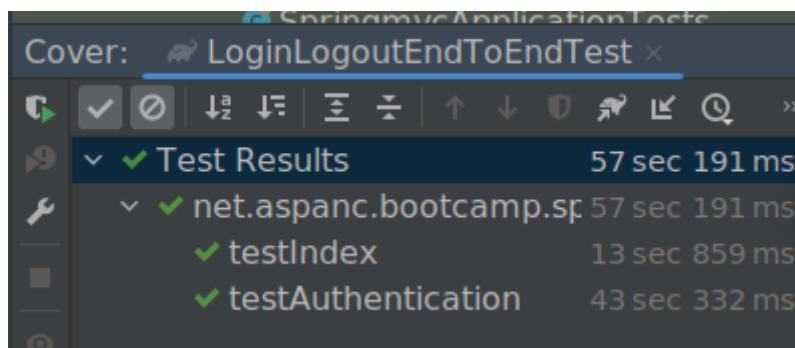


Figura 29.3: Resultado tests end-to-end.

Fuente: Propia

Como se puede ver en la imagen, tanto el test que determina si está la página principal como el que verifica que la autenticación funciona correctamente se han ejecutado correctamente.

Después de ver el resultado de las pruebas podemos concluir que la aplicación ha sido correctamente testada y que es muy complicado que en un futuro si se tuviese que implantar en un entorno real surgiese algún error de funcionalidad.

Esta parte es de las partes más importantes en la programación y se debe hacer meticulosamente y con la máxima cobertura posible.

8. EXPLOTACIÓN

Para poder explotar este producto debemos de poder implantarlo de la manera más fácil y generalizada posible. Cuanto más sencillo sea su ejecución en cualquier entorno, menos tiempo nos llevará poder sacar rédito del producto en un entorno real.

Por esto, la solución que he encontrado que mejor encaja con nuestros requisitos es la dockerización de la aplicación.

Dockerizar una aplicación Spring implica empaquetarla en un contenedor de Docker, lo que ofrece una serie de ventajas significativas. En primer lugar, la dockerización permite crear entornos de desarrollo, pruebas y producción consistentes y reproducibles. Esto facilita la colaboración entre equipos de desarrollo y mejora la portabilidad de la aplicación, ya que se puede ejecutar de manera uniforme en diferentes entornos y sistemas operativos.

Además, al utilizar contenedores, se logra una mayor eficiencia en el uso de recursos, ya que los contenedores son livianos y comparten el mismo sistema operativo subyacente.

La escalabilidad también se ve mejorada, ya que se pueden generar múltiples instancias del contenedor de la aplicación para manejar cargas de trabajo más pesadas.

8.1 Dockerización

Para conseguir la dockerización de un proyecto de Spring necesitamos un fichero llamado *Dockerfile*. Este fichero necesita tener una estructura y una sintaxis determinada.

En este fichero debemos incluir las dependencias necesarias para que se pueda ejecutar la aplicación. Este es el Dockerfile que he creado:

```
FROM sapmachine:11.0.18
#
WORKDIR /app

COPY . /app

CMD ["./gradlew", "bootRun"]
```

Figura 30: Dockerfile.

Fuente: Propia

Podemos ver en la imagen que al inicio del documento incluimos el JDK 11.0.18 de SapMachine para poder ejecutar la aplicación Java. Copiamos todos los ficheros que tiene el proyecto y finalmente ejecutamos el comando ‘*./gradlew bootRun*’ que corresponde a una tarea de Gradle que ejecuta la aplicación.

Con esto ya tendríamos la imagen de Docker de nuestra aplicación web. Existe actualmente una problemática, necesitamos de una base de datos para poder ejecutar correctamente esta aplicación de Spring.

8.2 Sincronización de Dockers

Para poder tener una base de datos conectada a la aplicación web podemos usar una imagen Docker de MySQL.

Para el despliegue de la base de datos y posteriormente el programa, debemos crear un archivo *docker-compose* que se ejecutará para poder sincronizar e interconectar los dos contenedores.

Este es el fichero docker-compose que he generado:

```
services:  
  mysql:  
    container_name: mysql-container  
    image: mysql:latest  
    ports:  
      - "3306:3306"  
    volumes:  
      - ./init.sql:/docker-entrypoint-initdb.d/init-script.sql  
      - ./data:/var/lib/mysql:rw  
    environment:  
      - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}  
      - MYSQL_DATABASE=${MYSQL_DATABASE}
```

Figura 31.1: Contenedor de MySQL del docker-compose.

Fuente: Propia

En el fichero declaramos que vamos a tener un contenedor que se llamará ‘mysql-container’ que vendrá de la última imagen del repositorio de mysql. Además se configurará la base de datos que tendrá contenida y la contraseña del usuario root. Los valores los tendremos en variables de entorno.

Para la persistencia de los datos y los datos de ejemplo tenemos los volúmenes ‘/var/lib/mysql’ y ‘/docker-entrypoint-initdb.d/init-script.sql’ respectivamente. Esto lo que hará es que si mapeamos estas ubicaciones con un dump de la base de datos con los datos de ejemplo y un fichero donde se guardarán los datos en tiempo real de la base de datos podremos incluir datos de ejemplo y tener un backup de los datos que tiene la base de datos.

Además del contenedor de MySQL, necesitaremos un contenedor de nuestra aplicación web:

```
games-api:  
  container_name: games-app-container  
  image: jmorenobra3/internship-api:latest  
  ports:  
    - "8080:8080"  
  depends_on:  
    - mysql  
  environment:  
    - SPRING_DATASOURCE_URL=${SPRING_DATASOURCE_URL}${MYSQL_DATABASE}  
    - SPRING_DATASOURCE_USERNAME=${SPRING_DATASOURCE_USERNAME}  
    - SPRING_DATASOURCE_PASSWORD=${MYSQL_ROOT_PASSWORD}
```

Figura 31.2: Contenedor de la aplicación web en el docker-compose.

Fuente: Propia

En la imagen podemos ver como declaramos el contenedor con nombre *games-app-container* que se obtendrá de ultima imagen *jmorenobra3/internship-api* que está contenida en un repositorio de **Docker Hub**.

Éste tendrá mapeado el puerto 8080 al puerto 8080 del host y depende del contenedor de MySQL. Como hago en el contenedor de MySQL, le configuro ciertas propiedades con variables de entorno.

8.3 Creación de la imagen de Docker

Para crear la imagen de Docker de la aplicación necesitamos ejecutar un comando. Este comando se ejecutará gracias a una tarea de Gradle propia que he generado. La tarea de Gradle es la siguiente:

```
tasks.register('buildDockerImages') { Task it ->
    doFirst {
        String version = project.getProperties().get("imageVersion")
        if (version == null) {
            version = "latest"
        }
        exec {
            commandLine 'docker', 'login'
            commandLine 'docker', 'buildx', 'build', '--platform', 'linux/amd64,linux/arm64', '-t',
                '|jmorenobrasero3/internship-api:' + version, '--push', '.'
        }
    }
}
```

Figura 32: Tarea de Gradle.

Fuente: Propia

En el fichero build.gradle pondremos el código de la imagen. Este código lo que hace es crearnos la tarea de Gradle llamada '*buildDockerImages*' en la que si no pasamos ningún parámetro se pone automáticamente la versión de la imagen generada en latest, si queremos especificar la versión de la imagen, podemos pasarla a la tarea como parámetro en el comando.

8.4 Implantación en un entorno real

Para poder hacer la implantación en un entorno real debemos tener un servidor. Ese servidor puede ser un servidor físico o en la nube. He escogido una máquina en la nube de Azure.

He creado una máquina virtual de Ubuntu Server en Azure:

| Size | |
|-----------------------|---|
| Size | Standard B2s |
| vCPUs | 2 |
| RAM | 4 GiB |
| Disk | |
| OS disk | games-api-internship_OsDisk_1_855aec3db59496b803eaa807fdb8e7f |
| Encryption at host | Disabled |
| Azure disk encryption | Not enabled |
| Ephemeral OS disk | N/A |
| Data disks | 0 |

Figura 33: Especificaciones de la VM de Azure.

Fuente: Propia

Como podemos ver en la imagen es una máquina virtual con 2 procesadores y 4 GB de RAM. Es la segunda máquina más pequeña que podemos arrancar en Azure. Si lo intentamos con la más pequeña, al iniciar el Docker de MySQL junto con la aplicación web, va muy lenta la máquina. De disco tengo un SSD.

Como veremos en la imagen siguiente, tengo configurado un DNS a la máquina virtual para poder acceder a la página más fácilmente:

| | |
|------------------------|---|
| Operating system | : Linux (ubuntu 20.04) |
| Size | : Standard B2s (2 vcpus, 4 GiB memory) |
| Public IP address | : 20.47.80.91 |
| Virtual network/subnet | : games-api-vnet/default |
| DNS name | : games-api-internship.francecentral.cloudapp.azure.com |
| Health state | : - |

Figura 34.1: DNS y configuraciones de red.

Fuente: Propia

Para poder acceder a la aplicación web necesito poder acceder a la red de la máquina virtual desde el exterior por el puerto 8080:

| Priority | Name | Port | Protocol | Source | Destination | Action | ... |
|----------|-------------------------------|------|----------|-------------------|----------------|--|-----|
| 300 | ⚠ SSH | 22 | TCP | Any | Any | Allow | ... |
| 310 | AllowAnyCustom8080Inbound | 8080 | TCP | Any | Any | Allow | ... |
| 65000 | AllowVnetInBound | Any | Any | VirtualNetwork | VirtualNetwork | Allow | ... |
| 65001 | AllowAzureLoadBalancerInBound | Any | Any | AzureLoadBalancer | Any | Allow | ... |
| 65500 | DenyAllInBound | Any | Any | Any | Any | Deny | ... |

Figura 34.2: Directiva de puertos.

Fuente: Propia

Para poder correr las imágenes de Docker necesitamos Docker. En la página oficial de Docker encontramos los pasos a seguir. Los comandos que debemos ejecutar son los siguientes:

```
sudo apt update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
echo "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
echo "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | sudo tee /etc/apt/sources.list.d/docker.list
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
sudo apt install docker-compose
```

Figura 35: Comandos para instalar Docker y Docker Compose.

Fuente: Propia

Con estos comandos conseguimos tener el demonio de Docker y el binario de Docker Compose para poder correr nuestro archivo *docker-compose.yml*.

Una vez lo mencionado anteriormente, simplemente necesitamos el archivo *docker-compose.yml*, crear o configurar las variables de entorno en el archivo oculto *.env* y tener un dump de la base de datos si queremos tener datos de prueba.

Esta es la carpeta que tendremos con todos esos archivos:

```
jorge@games-api-internship:~/docker$ ls -a
.  ..  .env  docker-compose.yml  init.sql
```

Figura 36: Carpeta con lo necesario para correr el docker-compose

Fuente: Propia

Este es un ejemplo de lo que contiene el archivo *.env*:

```
# Variables de entorno para el servicio mysql
MYSQL_ROOT_PASSWORD=root
MYSQL_DATABASE=app

# Variables de entorno para el servicio games-api
SPRING_DATASOURCE_URL=jdbc:mysql://mysql:3306/
SPRING_DATASOURCE_USERNAME=root
SPRING_DATASOURCE_PASSWORD=${MYSQL_ROOT_DATABASE}
```

Figura 37: Archivo *.env*.

Fuente: Propia

El dump de la base de datos se puede conseguir con el comando *mysqldump*. Este comando nos genera un fichero tal que así:

```
-- MariaDB dump 10.19 Distrib 10.6.12-MariaDB, for debian-linux-gnu (x86_64)
--
-- Host: kubernetes.docker.internal      Database: app
-----
-- Server version      8.0.19

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8mb4 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `games`
--
```

Figura 38: Dump de la base de datos.

Fuente: Propia

Teniendo todo esto, simplemente ejecutando el comando *docker-compose up* tendremos nuestra aplicación web en la máquina de Azure:

```
jorge@games-api-internship:~/docker$ docker-compose up
Creating network "docker_default" with the default driver
Pulling mysql (mysql:latest)...
latest: Pulling from library/mysql
3e0c3751e648: Pull complete
7914193c6f0e: Pull complete
fe4b3f820487: Pull complete
63683b304e3d: Pull complete
6ad9069836bd: Pull complete
```

Figura 39.1: Ejecución del comando *docker-compose*.

Fuente: Propia

El docker-compose primero traerá el contenedor de MySQL, después traerá el contenedor de la aplicación y los irá ejecutando en el orden que está especificado en el fichero. El final de la ejecución del comando:

```
restartedMain] o.s.s.web.DefaultSecurityFilterChain : Creating filter chain: any request, [org.springframework.security.web.context.SecurityContextPersistenceFilter@56322f43, org.springframework.security.web.header.HeaderWriterFilter@56322f43, org.springframework.security.web.authentication.logout.LogoutFilter@1e314776, org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter@1e5bef81, org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@5ddc0340f453, org.springframework.security.web.session.SessionManagementFilter@67cc754e, org.springframework.security.web.access.intercept.FilterSecurityInterceptor@1fc107e]
restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
restartedMain] o.s.b.a.w.s.WelcomePageHandlerMapping : Adding welcome page template: index
restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
restartedMain] n.a.b.springmvc.SpringmvcApplication : Started SpringmvcApplication in 4.743 seconds (JVM running for 5.296)
```

Figura 39.2: Final de la ejecución del comando *docker-compose*.

Fuente: Propia

Las últimas dos líneas nos muestran que la aplicación se ha iniciado en el puerto 8080 y que la JVM se ha iniciado hace 4 segundos, por lo que el programa ya está corriendo correctamente. Esta es la prueba de ello:

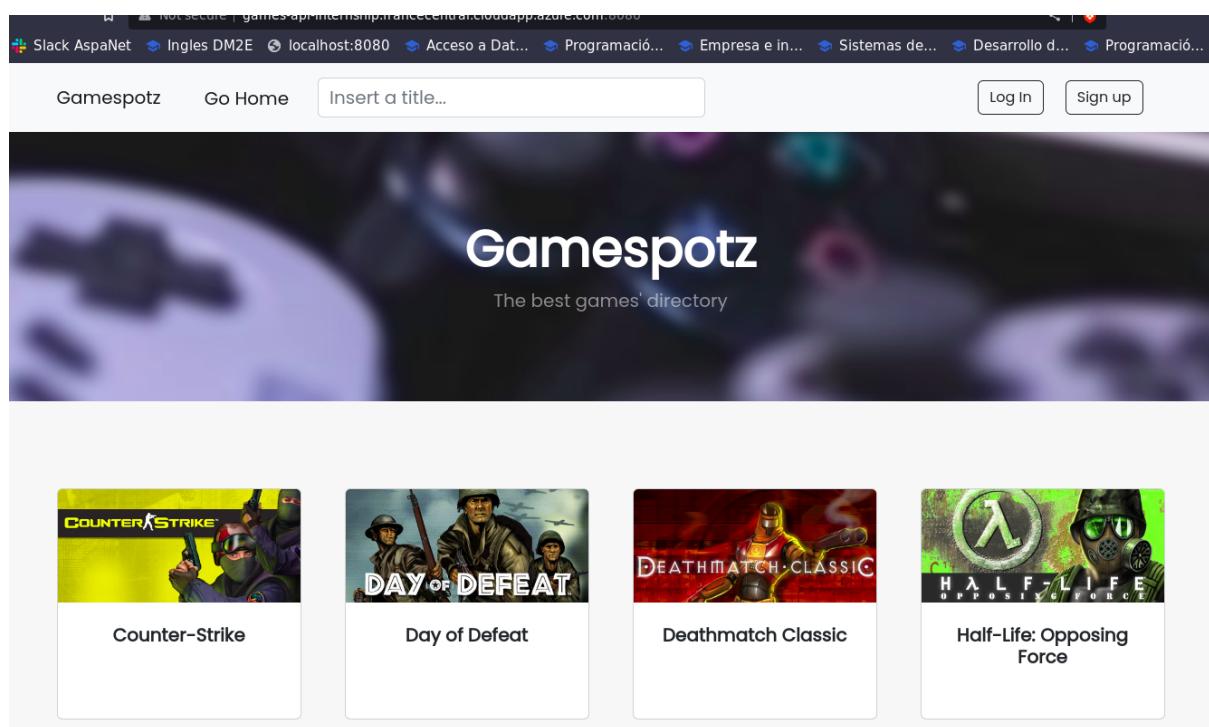


Figura 40: Página principal de la aplicación implantada en Azure.

Fuente: Propia

9. CONCLUSIONES

9.1 Conclusiones personales

Personalmente creo que he conseguido realizar todos los objetivos planteados en este proyecto además de utilizar en amplitud la mayoría de las tecnologías utilizadas actualmente en el mercado laboral.

La mayoría de las limitaciones encontradas en la realización de este proyecto han sido personales. La principal limitación ha sido el tiempo. He necesitado de tiempo para aprender cómo utilizar las tecnologías incluidas en el proyecto además del tiempo que he empleado en tener un código limpio y bien estructurado.

9.2 Posibles aplicaciones futuras

En un desarrollo con más tiempo, en un futuro se podría hacer este proyecto con un front-end desacoplado. Las tecnologías actuales más utilizadas para hacer esto son **Angular**, **React** y **Vue**.

Un front-end desacoplado, se refiere a la separación de la interfaz de usuario de una aplicación web. En lugar de tener una estructura monolítica donde el front-end y el back-end están estrechamente integrados, un enfoque desacoplado implica que el front-end se construya como una entidad separada e independiente del back-end. Esto se usa para crear una interfaz de usuario dinámica y en tiempo real, que se comunica con el back-end a través de APIs.

Las ventajas del enfoque desacoplado en el front-end son una mayor flexibilidad y agilidad en el desarrollo, ofrece una mejor experiencia de usuario. facilita la reutilización del código y la escalabilidad.

GLOSARIO

- **Back-end:** Es la parte de una aplicación encargada del procesamiento y la lógica del lado del servidor, incluyendo la gestión de datos y la comunicación con la base de datos. Es esencial para el funcionamiento de la aplicación.
- **Front-end:** Es la parte visible y accesible de una aplicación o sitio web con la que los usuarios interactúan directamente.
- **DTO:** Data Transfer Object. Es un objeto donde se encapsula la información del modelo de datos de la base de datos. Es el objeto que contiene los datos que se envían al front-end.
- **Façades:** Es un patrón de diseño que consiste en interfaces simplificadas que ocultan la complejidad de subsistemas internos, facilitando su uso y mantenimiento en aplicaciones de gran escala.
- **CRUD:** Se refiere a las operaciones básicas de Create, Read, Update y Delete para el manejo de datos en una aplicación.
- **SGBD:** Es un software para gestionar y administrar datos de manera eficiente y segura. Se usa sobre todo para manejar bases de datos.
- **ORM:** Es una herramienta que simplifica la interacción entre una base de datos y el código de una aplicación al mapear objetos a tablas y automatizar las operaciones de persistencia de datos.
- **Patrón MVC:** Es un patrón que divide una aplicación en tres componentes: Modelo, Vista y Controlador, para facilitar el desarrollo estructurado y modular.
- **Caché:** Es una memoria auxiliar de alta velocidad que almacena datos para acelerar su acceso y mejorar el rendimiento.
- **Cache provider:** Es un componente que gestiona y administra la caché en una aplicación para mejorar el rendimiento y la eficiencia en el acceso a los datos.
- **Responsive:** Es un diseño que se adapta automáticamente a diferentes tamaños de pantalla, proporcionando una experiencia óptima en diversos dispositivos.
- **Internacionalización:** Es adaptar una aplicación para que sea fácilmente traducible y adaptable a diferentes idiomas y regiones.
- **Principios REST:** Son un conjunto de reglas y convenciones para diseñar sistemas web. Estos principios promueven la simplicidad, la escalabilidad y la interoperabilidad entre los diferentes componentes de una aplicación. Se basa en la representación de datos en formatos como JSON o XML.

- **API RESTful:** Es una interfaz de programación que permite la comunicación estándar y escalable entre sistemas a través de HTTP, siguiendo los principios de REST.
- **POJO:** Es una clase de Java simple que se utiliza para almacenar y manipular datos en aplicaciones Java.
- **Converter:** es una clase que permite la conversión de datos entre diferentes tipos de objetos en una aplicación Spring.
- **Petición:** es un mensaje enviado por un cliente a un servidor para solicitar recursos o acciones en una aplicación web.
- **Petición asíncrona:** Es una solicitud en la que el cliente no espera bloqueado a la respuesta, permitiendo un flujo de trabajo más eficiente y mejor capacidad de respuesta.

WEBGRAFÍA

1. Ofertas de trabajo programador Java - [LinkedIn](#).
 2. Ofertas de trabajo programador Java - [Infojobs](#).
 3. Microservicios - [Red Hat](#).
-
- Información sobre Oracle SQL Server - [Oracle](#).
 - Información sobre MySQL - [MySQL](#).
 - Información sobre Microsoft SQL Server - [SQL Server](#).
 - Información sobre Spring - [Spring](#).
 - Información sobre Django - [Django](#).
 - Información sobre ASP.NET - [ASP.NET](#).
 - Información sobre Laravel - [Laravel](#).
 - Información sobre Symfony - [Symfony](#).
 - Los frameworks web más utilizados - [CodingNomads](#).
 - Información sobre Spring Boot - [Spring Boot](#).
 - Información sobre Spring Security- [Spring Security](#).
 - Información sobre Spring MVC - [Spring MVC](#).
 - ¿Qué es JSP? - [JSP](#).
 - Información sobre JSP - [JSP](#).
 - Información sobre Bootstrap - [Bootstrap](#).
 - Información sobre Materialize - [Materialize](#).
 - Información sobre Tailwind - [Tailwind](#).
 - Información sobre CSS - [CSS](#).
 - Información sobre jQuery - [jQuery](#).
 - Información sobre HTML - [HTML](#).
 - Información sobre AJAX - [AJAX](#).
 - Información sobre Ehcache - [Ehcache](#).
 - Información sobre i18n - [i18n](#).
 - ¿Cómo usar JUnit 4 en Spring? - [JUnit en Spring](#).
 - Información sobre JUnit - [JUnit](#).
 - Información sobre Selenium - [Selenium](#).
 - Información sobre Docker - [Docker](#).
 - Información sobre Docker Compose - [Docker Compose](#).

- Arquitectura de una aplicación web - [Existek](#).
- Información sobre Ibasco, la librería para acceder a la API de Steam - [Ibasco](#).
- Información sobre los DTOs - [DTOs](#).
- Información sobre IntelliJ IDEA - [IntelliJ IDEA](#).
- Información sobre JetBrains- [JetBrains](#).
- Información sobre los beans en Java - [beans de Java](#).
- Información sobre la inyección de dependencias - [Inyección de dependencias](#).
- ¿Qué es la IoC? - [Inversión de Control](#).
- Información sobre Spring Initializr - [Spring Initializr](#).
- Versión Java 11 LTS de Sap Machine - [Java 11 LTS](#).
- Información sobre SAP Machine - [SAP Machine](#).
- Información sobre Lombok - [Lombok](#).
- Información sobre la dependencia de Spring Web - [Spring Web](#).
- Información sobre H2 Database - [H2 Database](#).
- Información sobre Jakarta Persistence API - [JPA](#).
- Información sobre el Driver de MySQL - [MySQL Driver](#).
- Información sobre JDBC - [JDBC](#).
- Información sobre la validación de formularios de Spring - [Spring Validation](#).
- Información sobre Spring Session - [Spring Session](#).
- Información sobre Tomcat embebido - [Tomcat Embed](#).
- Información sobre Font Awesome - [Font Awesome](#).
- Información sobre Jakarta Standard Tag Library - [JSTL](#).
- Información sobre el patrón de Repositorio - [Repositorio](#).
- Información sobre los naming conventions - [naming convention](#).
- Información sobre el patrón de Abstracción - [Abstracción](#).
- ¿Qué es el JSR 303?- [JSR 303](#).
- La plantilla de Bootstrap que he utilizado - [Plantilla de Bootstrap](#).
- Información sobre los modales de Bootstrap - [Modales](#).
- Información sobre el CSRF - [CSRF](#).
- Información sobre React - [React](#).
- Información sobre Angular - [Angular](#).
- Información sobre Vue - [Vue](#).
- Página principal de Docker Hub - [Docker Hub](#).

ANEXO

Código fuente e imagen de Docker en el repositorio de [Docker Hub](#).