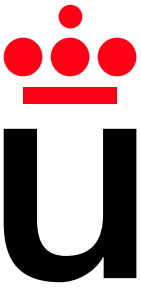


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

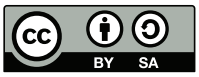
Grado en Ingeniería Informática

Curso 2024-2025

Trabajo Fin de Grado

QUANTUM LIBRARY: DESARROLLO DE UNA APLICACIÓN PARA LA GESTIÓN DE COLECCIONES DE MULTIMEDIA

Autor: Daniel Fernández Alvarado
Tutor: Michel Maes Bermejo



Agradecimientos

Quiero expresar mi agradecimiento a todas las personas que han hecho posible la realización de este TFG. En primer lugar, a mi familia y a mi pareja, por su apoyo incondicional durante todo este proceso de aprendizaje y trabajo. También agradezco profundamente a mis profesores y tutores de la universidad, quienes me han brindado los conocimientos necesarios para llevar a cabo este proyecto. A mis compañeros y amigos, que siempre han estado a mi lado, ofreciendo su apoyo y motivación. Y, finalmente, a mi tutor, Michel Maes Bermejo, quien ha estado guiándome a lo largo de todo este proceso. Gracias a su ayuda, conocimientos, paciencia y dedicación, este TFG ha sido una experiencia constructiva, satisfactoria y, sobre todo, muy enriquecedora.

Resumen

Hoy en día, la abundancia de contenido multimedia puede resultar abrumadora, no solo por su cantidad, sino también por su distribución a través de numerosas plataformas que utilizamos a diario. En este contexto, QuantumLibrary se presenta como una solución que unifica de manera visual y completa todos nuestros contenidos.

Este documento describe en detalle el desarrollo del Trabajo de Fin de Grado (TFG), cuyo objetivo principal ha sido la creación de una API que sirva como plataforma unificadora de diversas fuentes de información, brindando un acceso consolidado a datos que actualmente se encuentran dispersos en distintas plataformas. Además de la API, se ha diseñado y desarrollado una interfaz web intuitiva que permite a los usuarios interactuar con esta plataforma de manera moderna, accesible y dinámica. La API se ha implementado utilizando Spring-Boot, mientras que la aplicación web se ha construido con Next.js, aprovechando las ventajas de ambos frameworks.

A lo largo de este trabajo, se profundiza en el análisis del objetivo y el alcance de la aplicación, detallando cómo se ha diseñado una arquitectura flexible y modular que permita un fácil mantenimiento y escalabilidad. Se abordan las herramientas y tecnologías utilizadas, así como la metodología adoptada durante el desarrollo del proyecto. También se discuten los principales desafíos encontrados y las soluciones implementadas para resolverlos de manera eficaz. Además, se realiza un análisis detallado del código desarrollado, evaluando su calidad, las pruebas realizadas para garantizar su funcionamiento correcto y el proceso de despliegue de la aplicación en un entorno de producción. Finalmente, se exploran los próximos pasos para seguir mejorando la plataforma y ampliar su funcionalidad en el futuro.

Palabras clave:

- Java
- SpringBoot
- NextJS
- API REST
- BBDD
- Tailwind
- JUnit
- NextAuth
- NextUI
- AWS
- Tailwind
- CI/CD
- JWT
- SSL
- DOCKER

Índice de contenidos

Índice de figuras

Índice de códigos

1 Introducción

1.1. Contexto

Antiguamente, se utilizaban estanterías y otros medios físicos para mostrar y almacenar todo el contenido que consumíamos y disfrutábamos. Todo lo que hoy disfrutamos digitalmente alguna vez tuvo una existencia tangible: DVD para películas, libros, vinilos para música, cartuchos y discos para videojuegos. Esto no solo permitió una organización visual de nuestras colecciones, sino que también sirvió como reflejo de nuestras preferencias y gustos personales. Hoy en día, el mundo está plagado de contenido multimedia online distribuido a lo largo de distintas plataformas y medios: series, películas, animes, videojuegos... Cada uno de estos tipos de contenido está fragmentado en diversas aplicaciones y servicios, lo que a menudo hace que el manejo y la organización de nuestras colecciones digitales sea un desafío.

Aquí es donde entra en juego *Quantum Library*, una propuesta que pretende ser una biblioteca universal para todo tipo de contenido. *Quantum Library* se presenta como una solución innovadora que aborda la necesidad de organizar y acceder fácilmente a todo el contenido multimedia consumido.

La implementación inicial se ha centrado en los videojuegos debido a varias razones. En primer lugar, los videojuegos son un medio de entretenimiento que genera una gran cantidad de datos relevantes para los usuarios, como logros, horas de juego, niveles completados, entre otros. Estos datos pueden ser aprovechados para ofrecer una experiencia de usuario más rica y personalizada. Además, existen múltiples *APIs* y servicios externos que proporcionan información detallada sobre

los videojuegos, lo que facilita la integración y la automatización de la gestión de contenidos dentro de *Quantum Library*. En el futuro, el proyecto puede extenderse para abarcar todos los contenidos imaginables, tratando de especializarse en cada uno de ellos de manera similar a los videojuegos.

Dentro de nuestra biblioteca, encontraremos diversas categorías prefabricadas para ordenar nuestros juegos. En las dos primeras, *finalizados* y *completados*, los juegos son añadidos automáticamente al cumplirse las condiciones. En la categoría de *finalizados*, encontramos aquellos juegos en los que hemos terminado su historia principal, pero no hemos completado al 100 % su contenido, y en *completados*, encontramos aquellos juegos que hemos terminado y conseguido el 100 % del progreso y sus logros.

Por otro lado, tenemos las categorías *"backlog"* del 1 al 3, que funcionan de manera manual. Cada una de ellas actúa como una lista de juegos pendientes y la prioridad que el usuario quiera dar a cada uno de ellos. Esta última categoría tiene el propósito de permitir la visualización al usuario de una manera más clara de todos los juegos que tiene pendientes y la prioridad que les asigna, pudiendo llegar a usar la categoría con menos prioridad para ignorarlos por completo. Estas categorías permiten al usuario tener una visión clara y organizada de su colección, ayudando a tomar decisiones sobre qué juego iniciar a continuación.

Además de estas categorías, *Quantum Library* cuenta con un sistema de etiquetas (*tags*). Estas etiquetas las podremos añadir individualmente a cada uno de los juegos que queramos. Nos ayudan a encontrar el tipo de juego que nos apetezca jugar en un determinado momento, además de proporcionar información relevante sobre cada uno de ellos. Otra función destacada es el botón de selección aleatoria, que muestra un juego al azar dentro de los filtros aplicados por el usuario, ayudando a decidir qué jugar cuando hay demasiadas opciones.

Todas estas funciones cumplen un propósito por sí mismas, pero también sirven un bien conjunto para los jugadores conocidos como completacionistas, los cuales buscan terminar al 100 % la mayoría de juegos que puedan y encuentran una gran diversión en ello. Para estos, se ha diseñado la sección de estadísticas, donde se mostrarán gráficos relevantes para el jugador, como el porcentaje de juegos completados, finalizados, sin terminar y sin empezar, la media de logros obtenidos por juego, la cantidad total de logros obtenidos o la media de horas invertidas para completar un juego. De estas estadísticas, el jugador puede elegir eliminar las *backlog* que él elija de la ecuación. Esto es así para que los juegos que un jugador no quiera jugar no corrompan sus estadísticas totales.

Hasta ahora, hemos expuesto las funciones relevantes a la biblioteca del jugador. Sin embargo, *Quantum Library* pretende lidiar con otro problema en el campo de la consumición multimedia. Ponerse de acuerdo con un grupo de personas sobre cuál debería ser el siguiente título a disfrutar a veces puede resultar un reto. Distintos gustos y una amplia gama de elecciones a veces provocan una

gran pérdida de tiempo solo para ponerse de acuerdo.

Para abordar este problema, *Quantum Library* cuenta con un sistema de amigos y grupos, donde podemos añadir a nuestros amigos a un determinado grupo y se mostrarán los juegos que todos en conjunto compartáis en la biblioteca y que cuenten con ciertos *tags* especiales que permitan a más de una persona jugar a dicho juego, como por ejemplo *co-op*, *online* o *MMO*. La función de selección aleatoria tiene un giro adicional en este contexto, ya que cada jugador puede votar por los títulos que más le apetezcan dentro de los juegos del grupo. El sistema elige el juego con más votos o, en caso de empate, selecciona aleatoriamente entre los más votados.

Otro desafío que *Quantum Library* pretende resolver es la distribución de información entre plataformas. Existen muchas *APIs* diferentes que ofrecen diversas funciones, y la idea es unificarlas para ofrecer la mejor experiencia posible al usuario. La más importante implementación dentro de *Quantum Library* es la integración con la *API* de `Steam`, la mayor plataforma de compra de videojuegos en PC. Con ella, el usuario es capaz de importar toda su biblioteca de `Steam` de manera simple. Además, usando una *API* que realiza *scraping* sobre la propia página de `Steam`, se les asigna a cada uno de estos juegos los *tags* que la comunidad les haya dado en la página de la tienda de `Steam`.

El objetivo es integrar la biblioteca con el mayor número de *APIs* posibles que ofrezcan más información o que faciliten la experiencia al usuario. La opción de importar por ahora solo está disponible para `Steam`, ya que es la única plataforma que expone su *API* de manera pública. Sin embargo, otra integración importante es con `Steam Grid Database`, que nos ofrece un gran catálogo de portadas e imágenes creadas por la comunidad para nuestros juegos. Aunque el nombre mencione directamente a `Steam`, dentro de esta *API* podemos encontrar todos los juegos disponibles en todas las plataformas posibles. Esto hace posible que el usuario busque los juegos por nombre en su base de datos y utilice su fuente de carátulas e imágenes creadas por la comunidad para los juegos, permitiéndole personalizar su biblioteca con portadas atractivas.

2

Objetivos

El objetivo de *Quantum Library* es presentarse como una solución integral para la gestión de contenido multimedia digital, comenzando con una sólida base en la organización y gestión de videojuegos. Con su enfoque en la automatización, personalización y colaboración, *Quantum Library* no solo facilita la vida de los usuarios al organizar su contenido, sino que también mejora significativamente la experiencia de consumo multimedia, adaptándose a las necesidades individuales y grupales de manera eficiente y efectiva. Además de todo esto, la aplicación pretende rellenar un hueco en el mercado donde las *APIs* de videojuegos ofrecen información dispersa unificando todas ellas en una sola.

Como fase inicial del proyecto, el objetivo principal es desarrollar una aplicación funcional centrada en la gestión de bibliotecas de videojuegos, garantizando una experiencia de usuario refinada. Para ello, se han definido los siguientes objetivos específicos:

2.1. Usuarios

- Implementar un sistema de autenticación que permita a los usuarios crear una cuenta e iniciar sesión, asegurando que cada nombre de usuario sea único.
- Permitir la edición de información personal, incluyendo imagen de perfil, correo y contraseña, manteniendo el nombre de usuario inmutable como identificador único.

2.2. Biblioteca de Videojuegos

- Desarrollar un sistema de gestión de bibliotecas que permita la creación, edición, recuperación y eliminación de juegos en la colección personal de cada usuario.
- Mantener un catálogo global con información base de todos los juegos registrados, sirviendo como referencia para las bibliotecas personales.
- Integrar imágenes de portada en todos los juegos, permitiendo a los usuarios elegir entre opciones personalizadas o imágenes de *Steam Grid Database* (SGBD). En el catálogo global, únicamente se almacenarán imágenes obtenidas de SGBD para evitar contenido inadecuado.
- Implementar un método para importar automáticamente los juegos de la biblioteca de *Steam*, utilizando *Steam Spy* para obtener etiquetas y características del juego.

2.3. Estadísticas

- Desarrollar un sistema de estadísticas que analice y visualice dinámicamente los datos de uso de los juegos.
- Permitir la personalización de métricas, ofreciendo a los usuarios la posibilidad de excluir ciertas categorías del cálculo.

2.4. Grupos y Funcionalidad Social

- Implementar un sistema de gestión de grupos que permita a los usuarios crearlos, enviar invitaciones y aceptar o rechazar solicitudes.
- Desarrollar bibliotecas compartidas dentro de los grupos, mostrando únicamente juegos que cuenten con opciones multijugador.
- Permitir a los usuarios votar los juegos dentro de la biblioteca grupal, mostrando los resultados de manera visible para todos los miembros del grupo.

2.5. Seguridad

- Garantizar la protección de datos personales mediante autenticación basada en tokens JWT.

- Asegurar que todas las conexiones y transmisiones de datos estén cifradas en cada punto del sistema.

2.6. Interfaz y Experiencia de Usuario

- Diseñar una interfaz moderna, intuitiva y funcional, alineada con las expectativas de usuarios habituados a plataformas similares.
- Priorizar una navegación fluida y una experiencia visual atractiva que optimice la usabilidad sin comprometer el rendimiento.

3

Tecnologías, Herramientas y Metodologías

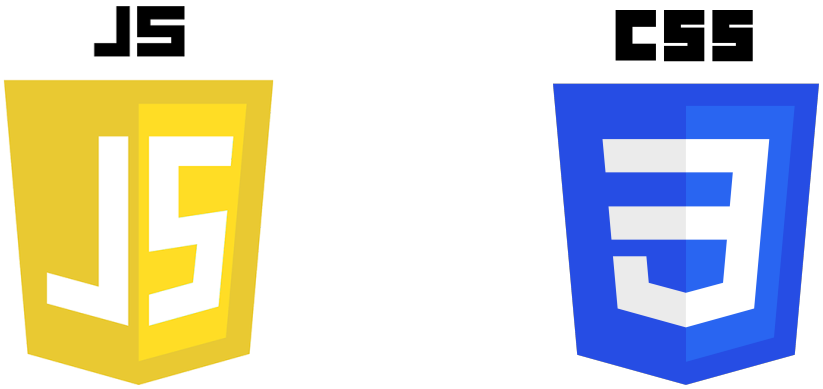
En este capítulo se detallan las diversas tecnologías, herramientas y metodologías utilizadas a lo largo del desarrollo del proyecto.

3.1. Lenguajes utilizados

- **Backend**
  Java (Figura 3.1) [1] fue el lenguaje de programación principal utilizado para construir la lógica del *backend* de la aplicación, en su versión 21. Es conocido por su robustez, escalabilidad, y su capacidad para gestionar grandes volúmenes de operaciones de manera eficiente. La elección de `Java 21` se debió a sus mejoras significativas en rendimiento y nuevas características del lenguaje, como optimizaciones en la gestión de memoria o actualizaciones que facilitan un desarrollo más limpio y seguro.

Figura 3.1: Logo de *Java*

- **Frontend**
  La aplicación *front-end* está desarrollada utilizando una combinación de lenguajes que permiten construir interfaces de usuario dinámicas, altamente interactivas y estilizadas. La lógica principal de la aplicación se encuentra escrita en `JavaScript` (Figura 3.2a) [2], lo que facilita la manipulación del `DOM` virtual y la implementación de funciones modernas para el manejo del estado, la interacción del usuario y el flujo de datos. Además, se utiliza `CSS` (Figura 3.2b) como lenguaje de estilos para definir el diseño visual, empleando clases utilitarias que permiten una personalización granular de los componentes y adaptabilidad a diferentes tamaños de pantalla. La integración de estas tecnologías permite estructurar y dar forma a la aplicación, brindando una experiencia de usuario optimizada y visualmente coherente.



(a) Logo de JavaScript (b) Logo de `CSS`

Figura 3.2: Lenguajes en Front-end

3.2. Tecnologías

3.2.1. Backend

`Spring Boot` (Figura 3.3a)[3] fue el *framework* elegido para el desarrollo del *backend*. Es una extensión del conocido *framework* `Spring` que permite la creación rápida de aplicaciones `Java` basadas en microservicios. `Spring Boot` simplifica la configuración y despliegue de aplicaciones, proporcionando un enfoque de desarrollo ágil mediante la automatización de tareas comunes, como la configuración de la base de datos, el manejo de dependencias, o el arranque de la aplicación con un servidor embebido. Esto permitió desarrollar el *backend* del proyecto de manera más eficiente, utilizando las características automáticas que optimizan el tiempo de desarrollo, como la configuración automática de componentes y la

integración de bibliotecas, lo que minimiza la necesidad de configuración manual y permite centrarse en la lógica de la aplicación. **JWT (JSON Web Token)** (Figura 3.3b)[4] se utilizó para gestionar la autenticación y autorización de usuarios en el *backend*. `JWT` es un estándar abierto que permite transmitir de manera compacta y segura información entre partes como un objeto `JSON`. En este caso, se generaron *tokens* firmados que se enviaban entre el servidor y el cliente, lo que permitía validar las identidades de los usuarios sin necesidad de mantener el estado de la sesión en el servidor. Esta tecnología facilita la seguridad en el acceso a la `API REST`, asegurando que solo los usuarios autenticados y autorizados puedan acceder a recursos sensibles, como servicios y datos específicos.

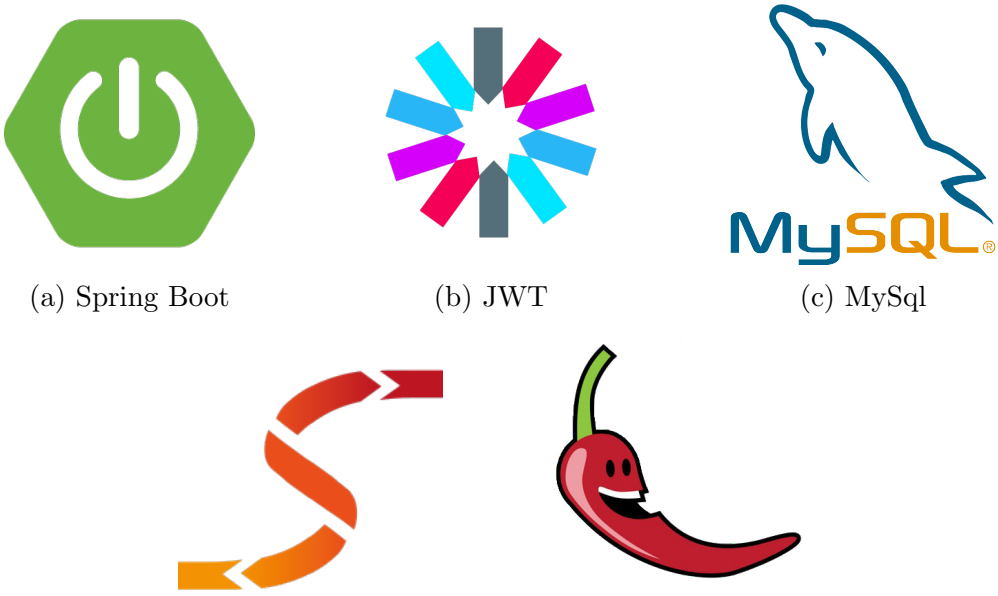
Para el almacenamiento de datos, `MySQL` (Figura 3.3c)[5] fue la base de datos relacional utilizada en el *backend*. `MySQL` es un sistema de gestión de bases de datos muy popular por su fiabilidad, alto rendimiento y su capacidad para gestionar grandes cantidades de datos de forma eficiente. Se eligió `MySQL` por su compatibilidad con aplicaciones de alto rendimiento, ya que permite realizar operaciones rápidas de lectura y escritura, así como una gestión eficiente de la información mediante el uso de tablas y relaciones. La estructura relacional de `MySQL` también facilitó la implementación de operaciones transaccionales, asegurando la consistencia de los datos.

Para la gestión de la conversión de objetos entre las diferentes capas de la aplicación, se empleó `MapStruct` (Figura 3.3d)[6]. `MapStruct` es un *framework* que permite la transformación eficiente de objetos entre diferentes tipos, como por ejemplo, convertir entidades a `DTOs` (*Data Transfer Objects*) y viceversa. Esto facilita la transferencia de datos entre las capas del *backend* sin necesidad de escribir código manual para cada conversión, lo que reduce los errores y mejora la mantenibilidad del código a largo plazo.

Además, se utilizó **Lombok** (Figura 3.3e)[7], una librería que ayuda a reducir la cantidad de código repetitivo en las clases `Java`. `Lombok` genera automáticamente métodos como *getters, setters*, constructores y otros métodos comunes necesarios en las clases de `Java`, lo que mejora la legibilidad y la limpieza del código. Una de sus funcionalidades clave fue la implementación del patrón `Builder`, definido por el *"Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides)* en 1994 en su libro seminal *Design Patterns: Elements of Reusable Object-Oriented Software* [8], que se utiliza para la creación de objetos complejos de una manera flexible y clara. `Lombok` facilita la implementación de este patrón mediante la anotación `@Builder`, lo que permitió construir objetos de manera controlada y con un código más legible.

Estas tecnologías proporcionaron una base robusta y escalable para el desarrollo del *backend*, asegurando que el proyecto fuera capaz de manejar solicitudes y datos de manera eficiente, mantener la seguridad del sistema y facilitar el mantenimiento del código. A través de estas herramientas y *frameworks*, se logró un *backend* optimizado tanto en rendimiento como en seguridad y facilidad de

desarrollo.



(a) Spring Boot (b) JWT (c) MySql

(d) Mapstruct (e) Lombok

Figura 3.3: Logos de las tecnologías usadas en backend

3.2.2. Pruebas

Para garantizar la calidad del *backend*, se ha utilizado `JUnit 5` (Figura 3.4)[9] como *framework* principal para las pruebas unitarias. `JUnit` proporciona anotaciones que facilitan la creación y gestión de los *tests*, como `@BeforeEach` y `@AfterEach`, que permiten ejecutar métodos antes y después de cada prueba, respectivamente. Además, el sistema de aserciones de `JUnit` permite verificar el comportamiento esperado de los métodos, como la validación de las respuestas de la API o la comprobación de datos, asegurando que se cumplan los requisitos de negocio definidos en el proyecto.

Dada la naturaleza del proyecto, que depende de la integración con múltiples servicios externos, las pruebas de integración son cruciales para garantizar el correcto funcionamiento y la estabilidad a lo largo del ciclo de vida del sistema. Por ello, se implementaron pruebas de integración continuas que permiten verificar de manera sistemática la comunicación entre nuestro sistema y las `APIs` externas. Para ello, se utilizó `WebClient` [10], una herramienta que facilita la realización de peticiones `HTTP` asincrónicas, permitiendo evaluar detalladamente las respuestas de las APIs externas. `WebClient` se integra de forma eficiente en el flujo de pruebas del sistema, facilitando la simulación de diversos escenarios, como la verificación de errores, la correcta interpretación de respuestas y la validación

de los datos recibidos. Este enfoque ha sido fundamental para asegurar que el *backend* pueda gestionar adecuadamente las respuestas de servicios de terceros, manteniendo la coherencia y consistencia de los datos y, por ende, una integración estable y confiable.

Además, se han creado pruebas para garantizar el correcto funcionamiento de nuestra propia API, utilizando la herramienta `MockMvc` proporcionada por `Spring Test`. Esta herramienta permite simular solicitudes `HTTP` sin necesidad de desplegar el servidor completo. Gracias a `MockMvc`, se pudo probar el comportamiento de los *endpoints* de manera aislada, verificando que las respuestas generadas sean correctas y que las peticiones sean procesadas adecuadamente.

Figura 3.4: Logo de JUnit5

3.2.3. Frontend

Para el desarrollo del *frontend*, se utilizó **Next.js** (Figura 3.5a)[11], un *framework* basado en **React** (Figura 3.5b)[12] que facilita la creación de aplicaciones web modernas y altamente optimizadas. `Next.js` incluye características avanzadas como el renderizado del lado del servidor (`SSR`) y la generación estática de páginas (`SSG`), lo que permite mejorar tanto el rendimiento como la optimización en motores de búsqueda (`SEO`). Estas capacidades garantizan tiempos de carga rápidos y una mayor visibilidad de la aplicación en los resultados de búsqueda. Una de las características más destacadas de `Next.js` es su soporte nativo para crear una **SPA (Single Page Applications)** [13], lo que permite que, al navegar entre las diferentes secciones de la aplicación, el contenido se actualice dinámicamente sin necesidad de recargar la página completa. Esto ofrece una experiencia de usuario mucho más fluida, ya que el navegador solo carga los datos necesarios, reduciendo considerablemente los tiempos de espera.

Por otro lado, `Next.js` implementa de manera nativa una optimización de imágenes mediante su componente `<Image>` [14], que proporciona características avanzadas como carga diferida (*lazy loading*), redimensionamiento automático y conversión de formatos a opciones más eficientes como `WebP`. Esto tiene un impacto significativo en el rendimiento de métricas clave como el **LCP (Largest Contentful Paint)** definido por `Google` [15], que mide el tiempo que tarda en renderizarse el elemento de contenido más grande visible dentro de la ventana del navegador desde el momento en que comienza a cargarse la página. La optimización de imágenes en `Next.js` contribuye a mejorar la experiencia del usuario de manera significativa dada la naturaleza del proyecto. Sin embargo, este proceso se realiza dinámicamente en el lado del servidor y solo la primera vez que cada

imagen es solicitada. Posteriormente, las imágenes optimizadas se almacenan en caché, eliminando la necesidad de volver a procesarlas. El inconveniente de este enfoque es que, si un gran número de imágenes intenta ser optimizado simultáneamente durante la primera renderización, puede ocasionar un alto consumo de `CPU` que llega a saturar el *host*, generando caídas críticas en la máquina que sostiene el servicio, sobre todo al tratarse del servicio gratuito ofrecido por `AWS`. Para mitigar este problema una vez que se llegó a producción, se integró **Sharp** (Figura 3.5c)[16], una biblioteca de procesamiento de imágenes extremadamente eficiente dentro del ecosistema de `Node.js`. `Sharp` permite redimensionar, convertir y optimizar imágenes de manera altamente eficiente. Esto permitió reducir el impacto en la `CPU` durante la optimización de imágenes, evitando caídas del servidor y garantizando tiempos de carga rápidos sin sacrificar la calidad visual.

Gracias a todo esto, `Next.js` permitió construir una experiencia web optimizada, dinámica y de alto rendimiento, con un enfoque en la eficiencia y la reducción de la latencia en este proyecto.

Para el diseño y estilización de la interfaz de usuario, se implementó **Tailwind CSS** (Figura 3.5d)[17], un *framework* de diseño basado en clases utilitarias que permite aplicar estilos directamente en los elementos `HTML` mediante clases predefinidas. Este enfoque facilita la creación de diseños responsivos, ya que permite personalizar rápidamente las interfaces para que se adapten a diferentes tamaños de pantalla y dispositivos. `Tailwind CSS` no solo agiliza el proceso de desarrollo, sino que también promueve la consistencia visual en toda la aplicación, gracias a su enfoque modular y a la capacidad de definir variables globales que pueden ser reutilizadas a lo largo de todo el *frontend*. Así, se garantiza que toda la aplicación tenga una apariencia coherente y bien definida, sin sacrificar la flexibilidad.

Además, se utilizó **NextUI** (Figura 3.5e)[18], una biblioteca de componentes diseñada específicamente para integrarse con `Next.js` basada en `Tailwind CSS`. `NextUI` proporciona una serie de componentes preconstruidos y altamente optimizados, lo que permitió acelerar el desarrollo de la interfaz de usuario. Al utilizar esta biblioteca, se pudo centrar más en la lógica y la funcionalidad de la aplicación, delegando el diseño visual en los componentes bien estructurados de `NextUI`. Esto no solo mejoró la productividad, sino que también facilitó la creación de una interfaz coherente, moderna y fácil de usar.

Para añadir una capa extra de interactividad y dinamismo, se incorporó **Framer Motion** (Figura 3.5f)[19], una potente librería de animación para React. Esta librería permite agregar transiciones suaves y animaciones personalizadas a los componentes, lo que mejora la experiencia del usuario al hacer que la interacción con la aplicación sea más atractiva y envolvente. Gracias a esta herramienta, se pudo enriquecer la experiencia del usuario con efectos visuales fluidos, como desvanecimientos, deslizamientos y otros tipos de animaciones, lo que contribuyó a crear una interfaz más moderna y atractiva visualmente.

Por otro lado, se utilizó **React Icons** (Figura 3.5g)[20] para mejorar la representación visual de la información, proporcionando iconos intuitivos que facilitan la interpretación de los datos y la navegación dentro de la aplicación. Esto contribuye a una experiencia de usuario más fluida y accesible, permitiendo una mejor identificación de acciones y secciones clave dentro de la interfaz.

En menor medida, para garantizar una visualización clara y efectiva de las estadísticas del usuario, se utilizó **Chart.js** (Figura 3.5h)[21], una biblioteca que proporciona herramientas versátiles para la representación de gráficos. Además, se integró **React Chart.js 2** [22], una capa de compatibilidad que adapta `Chart.js` para su uso en aplicaciones desarrolladas con React, permitiendo una gestión más eficiente del estado y la actualización dinámica de los gráficos dentro de la interfaz de usuario.

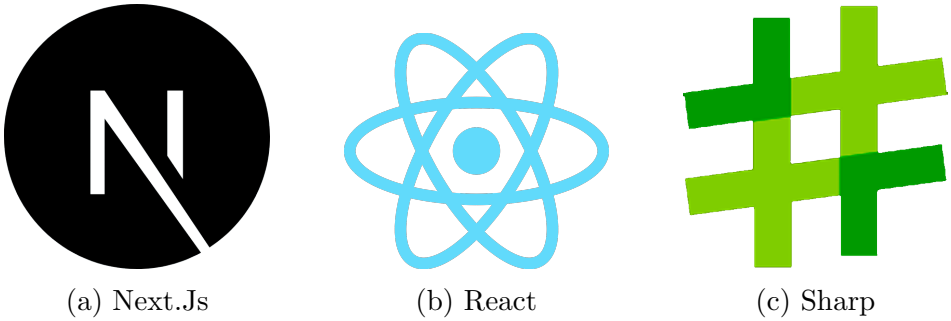
Finalmente, para gestionar la autenticación de los usuarios de manera segura y eficiente, se eligió **NextAuth** (Figura 3.5i)[23]. Esta librería proporciona una solución de autenticación simple y flexible para aplicaciones basadas en `Next.js`, permitiendo integrar diferentes proveedores de autenticación como redes sociales, cuentas personalizadas, entre otros. `NextAuth` se integra de forma fluida con la API del *backend* y garantiza una gestión segura de las sesiones de los usuarios, protegiendo los datos y manteniendo una navegación fluida y continua a través de la aplicación. Con esta herramienta, se facilitó la implementación de un sistema de inicio de sesión robusto, sin complicar el flujo de trabajo.

3.2.4. Contenerización

Para el despliegue del proyecto se ha utilizado **Docker** (Figura 3.6a)[24] como herramienta de contenerización. Docker permite empaquetar la aplicación junto con todas sus dependencias, configuraciones y librerías en contenedores independientes que pueden ejecutarse de manera consistente en cualquier entorno. Esto facilita el despliegue, la escalabilidad y la portabilidad del software.

Entre los principales beneficios se encuentra la portabilidad, ya que encapsula el entorno completo de la aplicación permitiendo que los contenedores funcionen de forma idéntica en desarrollo, pruebas y producción. Además, ofrece una alta escalabilidad al permitir desplegar múltiples instancias de servicios, proporcionando aislamiento entre servicios para evitar conflictos de dependencias y simplificando el despliegue automatizado.

La aplicación ha sido construida utilizando **Docker Compose** (Figura 3.6b)[25], lo que permite definir y lanzar los servicios del *frontend* y *backend* de manera simultánea. Esta configuración declarativa y automatizada facilita la infraestructura del proyecto. Dentro del archivo *docker-compose.yml*, se definen las imágenes de los contenedores junto con sus configuraciones de red. Para habilitar una comunicación eficiente entre ambos, se implementa un puente de red interno que



(a) Next.Js (b) React (c) Sharp

(d) Tailwind Css (e) Next Ui (f) Framer Motion

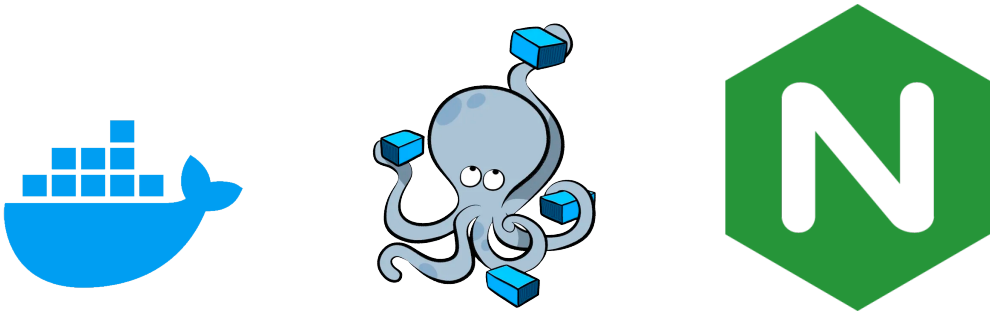
(g) React Icons (h) Chart.Js (i) Next Auth

Figura 3.5: Logos de las tecnologías usadas en frontend

permite que los servicios interactúen directamente entre sí utilizando sus nombres de contenedor como referencia, sin depender de rutas externas.

Dentro del mismo Docker Compose se ha configurado **Nginx** (Figura 3.6c)[26] como un *proxy* inverso encargado de gestionar el tráfico entrante y redirigir las solicitudes `HTTPS` hacia los contenedores pertinentes, ya sea el *frontend* o el *backend*. `Nginx` permite manejar el cifrado y desencriptado de las peticiones `HTTPS`, optimiza el rendimiento mediante la redirección eficiente de solicitudes y facilita la conversión automática de tráfico `HTTP` a `HTTPS`.

En conjunto, esta arquitectura basada en contenedores proporciona un entorno escalable, seguro y eficiente para la ejecución de la aplicación.



(a) Docker (b) Docker Compose (c) Nginx

Figura 3.6: Logos de las tecnologías usadas en contenerización

3.2.5. APIs

En este proyecto, se utilizaron varias `APIs` clave que enriquecen la experiencia de usuario y permiten la integración con servicios externos relevantes. Estas `APIs` proporcionan acceso a información valiosa sobre juegos, usuarios y datos visuales relacionados con la plataforma **Steam** (Figura 3.7a)[27], lo que mejora tanto la funcionalidad como la presentación de la aplicación. A continuación, se detallan las principales APIs empleadas y cómo cada una contribuyó al desarrollo y la mejora de la aplicación.

Para obtener información detallada sobre los juegos y la actividad de los usuarios en la plataforma de `Steam`, se utilizó la **Steam API** [28]. Esta `API` permite acceder a una amplia variedad de datos relacionados con los juegos, como estadísticas de jugadores, detalles de juegos y contenido relacionado con las cuentas de usuario. Gracias a la `Steam API`, la aplicación puede recuperar la información personal de cada usuario incluyendo su imagen de perfil y su biblioteca.

En cuanto a la presentación visual de la aplicación, se empleó la **SGDB (Steam Grid Database)** (Figura 3.7b)[29], una `API` que proporciona acceso a

imágenes, íconos y otros recursos visuales asociados con los juegos. Esta `API` fue clave para enriquecer la interfaz de usuario, permitiendo que la aplicación mostrara gráficos atractivos y bien presentados. Con `SGDB`, se optimizó la experiencia visual al incorporar imágenes de alta calidad de los juegos, lo que hace que la navegación sea más interactiva y agradable para los usuarios.

Finalmente, para mejorar las capacidades analíticas y proporcionar estadísticas detalladas, se integró la API **SteamSpy** (Figura 3.7c)[30]. Esta API ofrece información detallada sobre las estadísticas de los juegos en `Steam`, incluyendo el número de jugadores, tendencias de ventas, clasificaciones y otros datos analíticos valiosos. Al utilizar `SteamSpy`, la aplicación pudo ofrecer a los usuarios análisis de datos más profundos y actualizados, permitiendo visualizar los tags establecidos por la comunidad en cada juego.

En conjunto, estas APIs contribuyeron significativamente a la creación de una aplicación rica en datos, interactiva y visualmente atractiva, mejorando tanto la experiencia del usuario como la funcionalidad de la aplicación.

(a) Steam (b) Steam Grid Data Base (c) SteamSpy

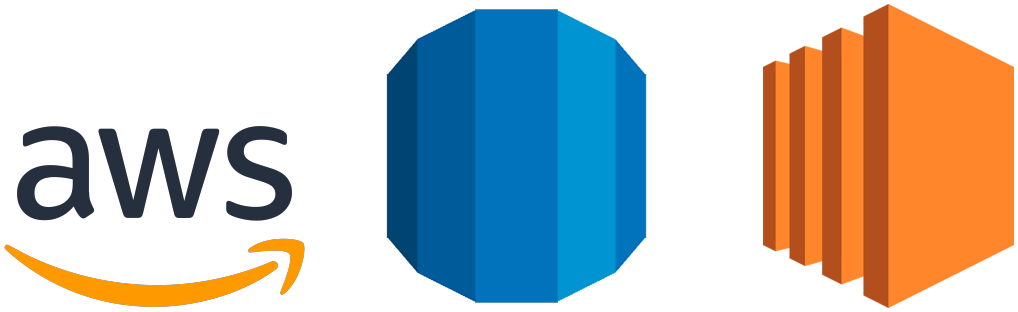
Figura 3.7: Logos de las tecnologías usadas en APIs

3.2.6. AWS

En este proyecto, se utilizaron diversos servicios de **AWS (Amazon Web Services)** (Figura 3.8a)[31] para gestionar la infraestructura de la aplicación de manera eficiente y escalable.

Para la gestión de la base de datos, se utilizó **AWS RDS (Relational Database Service)** (Figura 3.8b)[32], un servicio completamente administrado que facilita la configuración, operación y escalabilidad de bases de datos relacionales. En este proyecto, `AWS RDS` se encargó de gestionar de manera segura y escalable la base de datos `MySQL`.

Para el despliegue de la aplicación, se utilizó **AWS EC2 (Elastic Compute Cloud)** (Figura 3.8c)[33], un servicio que proporciona instancias escalables y personalizables para ejecutar aplicaciones en la nube.



(a) AWS (b) RDS (c) EC2

Figura 3.8: Logos de las tecnologías usadas en AWS

3.2.7. SSL

En la fase final del proyecto, se implementó el cifrado completo del sitio web para garantizar una navegación segura mediante el protocolo `HTTPS`. Inicialmente, se utilizó **Cloudflare** (Figura 3.9a)[34] para cifrar las conexiones entre los clientes y el servidor de manera flexible. Esta configuración flexible permitía el acceso mediante `HTTPS` al *frontend*, pero el *backend* seguía funcionando en `HTTP`, lo cual introducía una posible vulnerabilidad al mantener una comunicación no cifrada.

Posteriormente, se configuró **Certbot** (Figura 3.9b)[35] para obtener un certificado SSL gratuito proporcionado por `Let's Encrypt` [36], con el fin de cifrar también las comunicaciones del *backend*. Además, se integró `Nginx` dentro del entorno definido por `Docker Compose` para actuar como *proxy* inverso. `Nginx` se encargó de redirigir las solicitudes externas a los servicios internos de manera segura y eficiente, gestionando tanto la terminación `SSL` como las reglas de redireccionamiento de tráfico `HTTP` a `HTTPS`.

(a) Cloudflare (b) Certbot

Figura 3.9: Logos de las tecnologías usadas en SSL

3.3. Herramientas

En el proyecto se utilizaron diversas herramientas que facilitaron la gestión del código, la automatización de tareas y la administración de bases de datos,

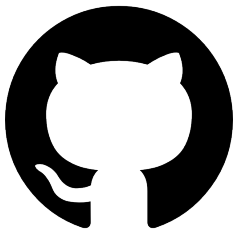
entre otros aspectos. A continuación, se describen las herramientas principales utilizadas en el desarrollo.

3.3.1. Control de Versiones

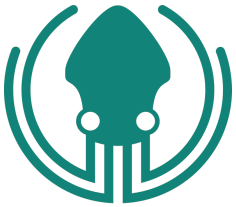
GitHub (Figura 3.10a)[37] fue la plataforma de control de versiones elegida para alojar el código fuente del proyecto. Proporcionó un entorno colaborativo para realizar revisiones de código, gestionar problemas y asegurar el historial de desarrollo mediante repositorios, ramas y *pull requests*.

Para la gestión de los repositorios Git de manera gráfica, se utilizó **GitKraken** (Figura 3.10b)[38]. Esta herramienta visual facilitó la administración de ramas, *commits y pull requests*, mejorando la experiencia del manejo de versiones de manera intuitiva.

En cuanto a la integración y despliegue continuo (CI/CD), se implementó **GitHub Actions** [39]. Esta herramienta permitió automatizar tareas de construcción, pruebas y despliegue, mejorando la eficiencia y calidad del desarrollo al ejecutar *pipelines* automáticamente tras cada *commit*.



(a) GitHub



(b) GitKraken

Figura 3.10: Logos de las tecnologías usadas en Control de Versiones

3.3.2. Desarrollo

`IntelliJ IDEA` (Figura 3.11a)[40] fue el entorno de desarrollo integrado (IDE) utilizado para todo el desarrollo del proyecto, tanto en el *backend* como en el *frontend*. Ofreció herramientas avanzadas para depuración, integración con `Git`, análisis de código y pruebas, facilitando un desarrollo eficiente en ambos lados de la aplicación. Además, los numerosos *plugins* disponibles en su *marketplace* ayudaron a optimizar y personalizar el entorno de desarrollo, proporcionando funcionalidades adicionales que mejoraron la productividad y calidad del trabajo realizado.

Para gestionar las dependencias y las tareas de construcción en el *backend*, se utilizó `Maven` (Figura 3.11b)[41]. Esta herramienta permitió la gestión auto-

matizada de las librerías necesarias, configuraciones y compilación del proyecto, mejorando la eficiencia y mantenibilidad.

La documentación de la API REST del *backend* fue facilitada por `Swagger` (Figura 4.3)[42]. Esta herramienta permite crear documentación interactiva y accesible tanto para los desarrolladores como para los usuarios, simplificando el entendimiento y la integración de la API.

Para la documentación del código fuente en Java, se utilizó `JavaDocs` (Figura 3.11d)[43]. Esta herramienta automatiza la generación de documentación clara y accesible, permitiendo que otros desarrolladores puedan entender o mantener el código de manera eficiente en el futuro.

(a) IntelliJ Idea (b) Maven (c) Swagger (d) JavaDocs

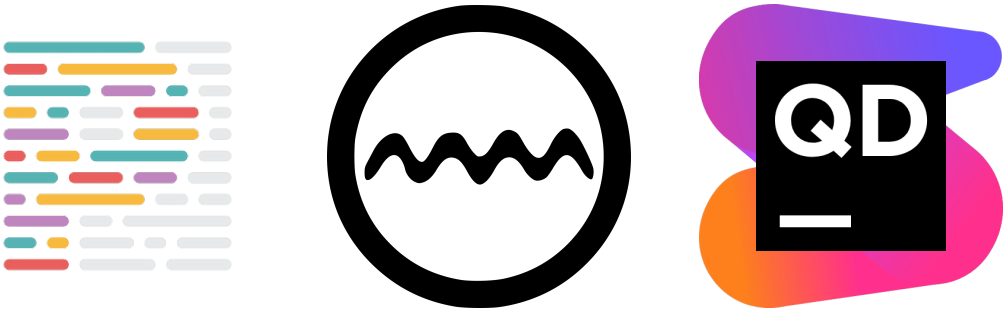
Figura 3.11: Logos de las tecnologías usadas en Desarrollo

3.3.3. Calidad del Código

Para mantener la mejor calidad de código posible, se han utilizado las librerías `Prettier` (Figura 3.12a)[44] y `Trivago Prettier Sort Imports` [45]. `Prettier` se encarga de formatear el código, asegurando que mantenga una estructura coherente y uniforme a lo largo de toda la aplicación, además de cumplir con las normas generales de limpieza y legibilidad de código. Por su parte, `Trivago Prettier Sort Imports` organiza automáticamente los *imports*, lo que facilita su lectura y mejora la claridad del código, contribuyendo a un entorno de desarrollo más ordenado y eficiente.

`Sonar Lint` (Figura 3.12b)[46] fue la herramienta principal utilizada para el análisis estático del código. Se empleó para garantizar la calidad del código y detectar vulnerabilidades de seguridad, ayudando a identificar problemas de estilo, complejidad y posibles errores, lo que asegura un código más eficiente y seguro.

Como complemento a Sonar, se utilizó (Figura 4.19)`Qodana` [47]. Esta herramienta, centrada en el análisis estático, extendió la identificación de vulnerabilidades y mejoró la calidad del código, proporcionando un análisis más personalizado para los desarrolladores.



(a) Prettier (b) Sonar Lint (c) Qodana

Figura 3.12: Logos de las tecnologías usadas en Calidad de Codigo

3.3.4. Gestión de Bases de Datos

Para la gestión y administración de la base de datos MySQL durante el desarrollo, se utilizó `DBeaver` (Figura 3.13)[48]. Esta herramienta de código abierto ofreció una interfaz gráfica sencilla para gestionar las conexiones y operaciones sobre la base de datos, facilitando las interacciones durante la administración del proyecto.

Figura 3.13: Logo de DBeaver

3.3.5. Peticiones API

Se utilizó `Postman` (Figura 3.14)[49] para realizar las pruebas tanto de nuestra propia API como de servicios externos. Esta herramienta resultó fundamental para verificar el correcto funcionamiento del servicio *backend* de manera aislada, sin necesidad de depender del *frontend*. Esto permitió detectar y resolver problemas en el *backend* de forma más eficiente durante el desarrollo.

`Postman` facilita el manejo de variables de entorno, lo que permite cambiar fácilmente entre entornos de desarrollo y producción sin necesidad de modificar manualmente las rutas ni otras configuraciones específicas. Esta funcionalidad es especialmente útil para mantener una configuración limpia y automatizar pruebas en distintos escenarios.

Además, esta herramienta permite la ejecución de *scripts* previos a las peticiones (*pre-request scripts*), los cuales resultaron clave para automatizar tareas como el manejo de inicios de sesión mediante la obtención y almacenamiento de *tokens* de autenticación. Esto fue indispensable para probar de manera eficiente los recursos protegidos de la API, simulando flujos de autenticación completos.

Figura 3.14: Logo de Postman

3.3.6. Manejo de Secretos

Para proteger la información sensible, como claves API o contraseñas, dentro de la aplicación, se optó por utilizar los secretos de `GitHub`. Esta es una de las pocas opciones gratuitas disponibles para gestionar secretos de manera segura en el entorno de desarrollo. Sin embargo, su uso está restringido a las acciones realizadas dentro de la plataforma de `GitHub`. Para integrarlos de manera efectiva en el flujo de trabajo, se decidió cargar todos los secretos necesarios como variables de entorno en el archivo `docker-compose`, tanto durante la construcción de la aplicación como en su posterior ejecución.

De esta manera, se logró garantizar que la aplicación mantuviera altos estándares de seguridad sin comprometer la flexibilidad o la eficiencia durante el proceso de despliegue y construcción.

3.3.7. Diseño de logotipo

Para la creación del logo principal de `Quantum Library`, se ha utilizado `Photoshop` (Figura 3.15)[50], aprovechando sus potentes herramientas de diseño gráfico para crear una imagen visualmente atractiva y representativa de la identidad del proyecto.

Figura 3.15: Logo de PhotoShop

3.4. Metodologías

La metodología de desarrollo de software utilizada a lo largo del proyecto ha sido iterativa e incremental. En este enfoque, se partió de una base definida del proyecto, con un conjunto inicial de funcionalidades y una arquitectura establecida. A medida que avanzaba el desarrollo, se fue completando y ampliando el sistema en diversas áreas, añadiendo nuevas características o mejorando las existentes en función de las necesidades emergentes.

Cada iteración permitió una evaluación constante de los avances y una retroalimentación continua, lo que permitió identificar posibles áreas de mejora, corregir errores y ajustar la dirección del proyecto conforme a los requisitos. Este proceso de desarrollo flexible fue clave para adaptar el proyecto a cambios de alcance, optimizar la eficiencia y asegurar que el producto final cumpliera con los objetivos de calidad y funcionalidad definidos al inicio.

Además, se incorporaron nuevas funcionalidades de manera incremental, permitiendo probar cada adición de forma independiente antes de integrarlas completamente. Este enfoque no solo facilitó la detección temprana de problemas, sino que también permitió que el proyecto evolucionara de manera controlada, sin necesidad de reestructuraciones importantes.

Las fases de prueba y validación fueron fundamentales en cada iteración, asegurando que cada componente del sistema cumpliera con los estándares de calidad antes de ser integrado con el resto del sistema. Esto también permitió realizar ajustes a las funcionalidades existentes basándose en la experiencia adquirida durante el proceso de desarrollo.

Al final del proyecto, la metodología iterativa e incremental permitió no solo cumplir con los objetivos iniciales, sino también introducir mejoras y optimizaciones en áreas que podrían haber sido pasadas por alto en un enfoque de desarrollo más tradicional. Esto contribuyó a una mayor estabilidad, escalabilidad y eficiencia del producto final.

3.4.1. Git

Al tener tanto el proyecto *backend* como el *frontend* en un mismo repositorio, se adoptó un modelo de desarrollo de Git basado en una versión modificada de GitFlow. En lugar de contar con una única rama `development`, se crearon dos ramas principales a partir de `main`: una rama para el *backend*, denominada `Back`, y otra para el *frontend*, llamada `Front`. A partir de estas surgían las ramas `Feature`, utilizadas para desarrollar las distintas características de la aplicación, las cuales se eliminaban una vez fusionadas en su rama correspondiente, `Back` o `Front`.

Cada una de estas ramas operaba de manera individual siguiendo la lógica de GitFlow. A partir de una rama de desarrollo específica (`Back/development` o `Front/development`), se añadían nuevas funcionalidades o configuraciones en ramas `Feature` o `Config` dedicadas a cada área. Este enfoque permitió gestionar de manera eficiente el desarrollo independiente del *backend* y el *frontend*, facilitando un flujo de trabajo organizado y evitando conflictos, además de mejorar la modularidad del proyecto.

Cuando una rama alcanzaba un nivel de estabilidad adecuado, sus cambios se fusionaban en `Back/development` o `Front/development`, según correspondiera, integrando los avances del *backend* o *frontend* de forma coherente y manteniendo el proyecto actualizado sin comprometer su estabilidad. Este enfoque permitió trabajar de manera paralela e independiente en ambas áreas, sincronizando de manera eficiente el desarrollo del proyecto.

También se crearon ramas específicas para grandes configuraciones o cambios tecnológicos importantes, agrupadas bajo el nombre `Configuration`, las cuales permanecían aisladas de las ramas principales del *backend* y el *frontend* para evitar inestabilidades.

Cuando el proyecto alcanzaba una meta importante o una funcionalidad clave estaba lista y probada, los cambios se fusionaban en la rama `main`, permitiendo el despliegue de los avances. Este paso garantizaba que las nuevas características o mejoras estuvieran disponibles en la versión de producción, representando un hito importante y asegurando que las funcionalidades del *backend* y *frontend* estuvieran alineadas y listas para su uso en el entorno real.

Un fragmento representativo del historial de nuestras ramas se puede observar en la figura 3.16.

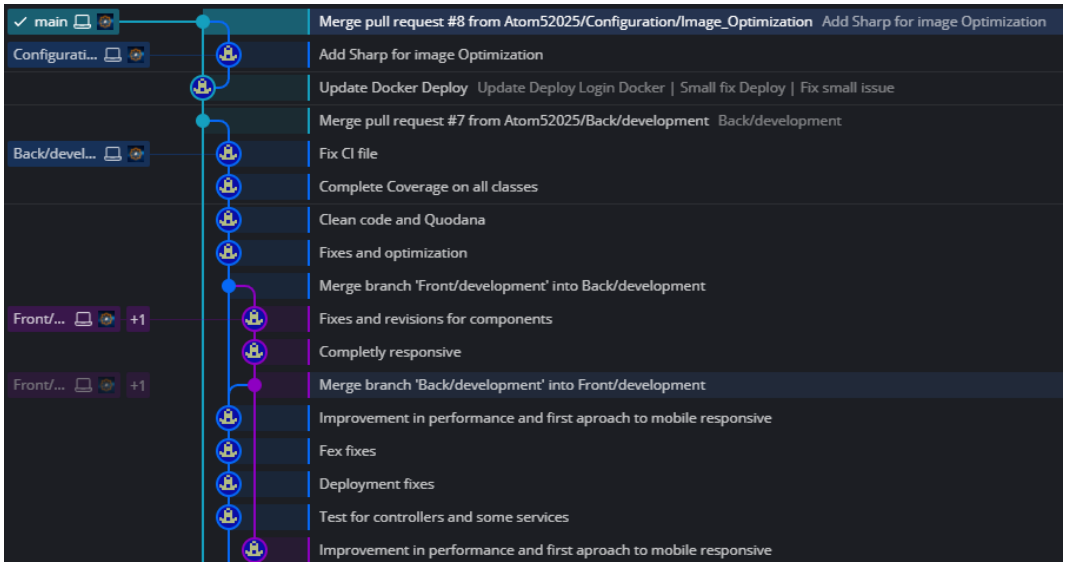


Figura 3.16: Flujo de trabajo en Git

4

Descripción Informática

Nuestra aplicación se divide en *frontend* y *backend*. Nuestro *backend* ofrece una `API` pública reuniendo información de diversas `APIs` que luego almacenamos en nuestra propia base de datos, mientras que el *frontend* ofrece una experiencia optimizada al usuario para poder visualizar su contenido en cualquier dispositivo.

4.1. Requisitos

En esta sección abordaremos los requisitos planteados en la aplicación a nivel de historias funcionales desde el punto de vista del usuario. También se nombrarán otros tipos de requisitos a la hora de desarrollar el proyecto que no están relacionados en sí con el propio funcionamiento de la aplicación.

4.1.1. Requisitos Funcionales

- **RF1.** Como usuario, puedo registrar una nueva cuenta.
- **RF2.** Como usuario, puedo iniciar sesión en mi cuenta.
- **RF3.** Como usuario, puedo visualizar una biblioteca global.
- **RF4.** Como usuario registrado, puedo acceder a mi biblioteca.
- **RF5.** Como usuario registrado, puedo añadir nuevos juegos a mi biblioteca.

  - **RF5.1** Crear nuevo y agregar.
  - **RF5.2** Seleccionar de la biblioteca global y agregar.

- **RF6** Como usuario registrado, puedo importar de `Steam`.
  - **RF6.1** Juegos.
  - **RF6.2** Perfil.

- **RF7.** Como usuario registrado, no debo encontrarme un juego duplicado en mi biblioteca.

- **RF8.** Como usuario registrado, puedo cambiar los detalles de un juego de mi biblioteca.
  - **RF3.1** Cambiar imagen.
  - **RF3.2** Cambiar información completacionismo.

- **RF9.** Como usuario registrado, puedo filtrar mi biblioteca.
  - **RF9.1** Por categoría.
  - **RF9.2** Por etiqueta.
  - **RF9.3** Por nombre.

- **RF10.** Como usuario registrado, puedo seleccionar aleatoriamente un juego.

- **RF11.** Como usuario registrado, puedo crear un grupo nuevo.
  - **RF11.1** Invitar más usuarios a un grupo.
  - **RF11.2** Salir de un grupo.
  - **RF11.3** Visualizar la biblioteca de juegos comunes del grupo.
  - **RF11.4** Filtrar la biblioteca grupal.
  - **RF11.5** Votar un juego en un grupo.
  - **RF11.6** Seleccionar un juego al azar en base a los votos.

- **RF12.** Como usuario registrado, puedo editar la información de mi perfil.
  - **RF12.1** Cambiar imagen.
  - **RF12.2** Cambiar correo.
  - **RF12.3** Cambiar contraseña.

- **RF13** Como usuario registrado, puedo visualizar mis estadísticas.
  - **RF13.1** Añadir y eliminar categorías de la ecuación.

4.1.2. Requisitos No Funcionales

- **RNF1.** La aplicación web debe estar optimizada. Los tiempos de carga de las pantallas deben ser aceptables.
- **RNF2.** La aplicación web debe dar feedback de las operaciones realizadas.
- **RNF3.** La aplicación web debe ser accesible y correctamente visualizable desde cualquier dispositivo.
- **RNF4.** La aplicación debe guardar todos los datos obtenidos en una base de datos relacional.
- **RNF5.** La aplicación debe tener un sistema de autenticación basada en credenciales gestionada mediante tokens `JWT`.

4.2. Arquitectura de la aplicación

En la Figura 4.1 se muestra un diagrama que describe la arquitectura de la aplicación. La estructura de la aplicación se divide principalmente en dos partes: el *frontend* y el *backend*.

La aplicación está completamente desplegada en AWS. El *frontend* se encuentra en un contenedor basado en `React` y `Next.js`, y cuenta con un *backend interno* para optimizar el rendimiento. Además, utilizamos diversas librerías de apoyo para el diseño, como `Tailwind`, `NextUI` y `FramerMotion`.

El *backend* está desarrollado como una `API REST` utilizando `Spring Boot` y `Java`, y se ejecuta en un contenedor independiente. Al desplegar la aplicación con `Docker Compose`, se crea un contenedor adicional de `Nginx`, que actúa como un proxy inverso, redirigiendo las solicitudes a los contenedores correspondientes y facilitando la comunicación entre el *frontend* y el *backend*. El despliegue se realiza en una máquina `EC2` basada en `Ubuntu`, que se conecta con una instancia de `RDS` que aloja la base de datos `MySQL`. Se ha establecido una conexión directa entre ambos para minimizar el tiempo de latencia en las peticiones.

Además, configuramos `Cloudflare`, que proporciona comunicación segura mediante `HTTPS`, gracias a su certificado `SSL`, funcionando como un proxy entre el usuario y el servidor.

La aplicación sigue el patrón de diseño Modelo Vista Controlador (`MVC`) tanto en el *backend* como en el *frontend*.

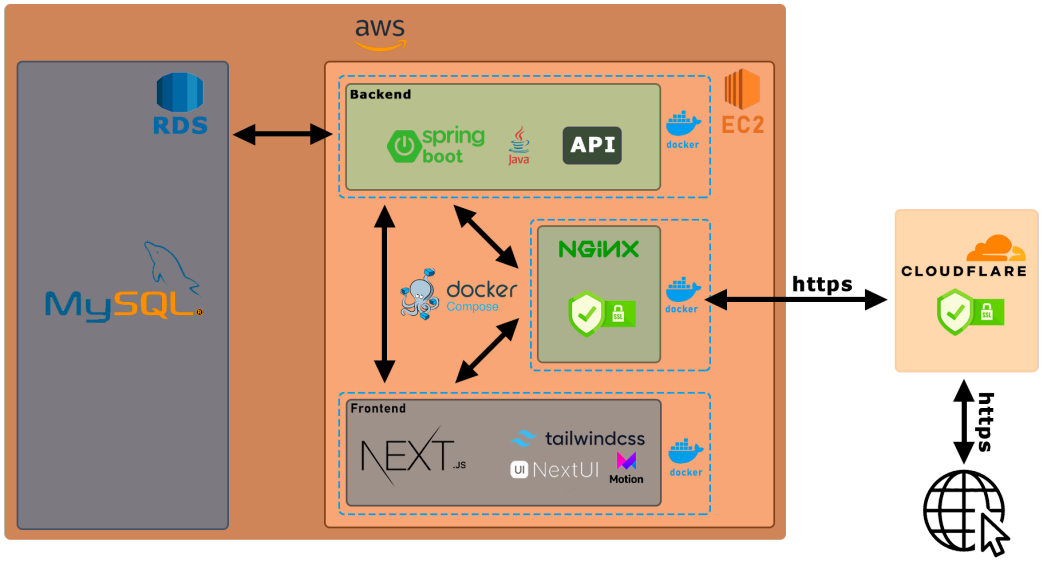


Figura 4.1: Arquitectura aplicación

4.2.1. Backend

La estructura del proyecto *backend* se presenta en base a los estereotipos de `Spring Boot` y `JPA`:

- **Repositorio**
  Para la gestión de datos y la creación automática de las tablas de la aplicación, se ha utilizado `JPA` (*Java Persistence API*). `JPA` permite generar las tablas y definir sus relaciones basándose en los modelos de datos, que hemos marcado con la anotación *@Entity*. Esta estrategia facilita la sincronización entre la estructura de la base de datos y el modelo de datos de la aplicación, asegurando que la persistencia sea manejada de manera eficiente y sin intervención manual.

  Para mejorar el rendimiento y optimizar los tiempos de respuesta, hemos implementado *projections*. Esta técnica permite seleccionar y recuperar únicamente ciertos campos específicos en lugar de cargar las entidades completas, reduciendo así la cantidad de datos transferidos y procesados durante cada consulta.

  En la estructura del proyecto, los modelos de datos y las interfaces que gestionan las consultas y la persistencia se encuentran en las carpetas **model** y **repository**, respectivamente, como se ilustra en la figura 4.4a.

- **Servicios**
  Para gestionar la aplicación, se han creado nueve microservicios, todos ellos divididos en interfaz e implementación para facilitar la legibilidad de los

métodos expuestos. Cada microservicio maneja únicamente una de las tablas de la base de datos mediante su respectivo repositorio. Si fuera necesario el uso de una tabla externa a la suya, se hace uso del microservicio pertinente. De esta manera, se mantiene una estructura modular y escalable.

La arquitectura de los microservicios puede observarse en la figura 4.2. Los microservicios principales son **GameService**, **UserService** y **GroupService**, que manejan los repositorios de juegos, usuarios y grupos, respectivamente. A partir de ellos, encontramos **UserGameService**, que hace uso de **GameService** y **UserService** para gestionar los juegos de cada usuario, y **UserGroupsService**, que hace uso de **GroupService** y **UserService** para gestionar los grupos de cada usuario.

Por otro lado, tenemos **SteamService**, que hace uso únicamente de **SteamSpyService** y **SteamGridDBService** para realizar las peticiones necesarias y obtener toda la información disponible sobre los juegos de una biblioteca de `Steam`. Por último, **AuthService** gestiona las necesidades de autenticación y seguridad de la aplicación.

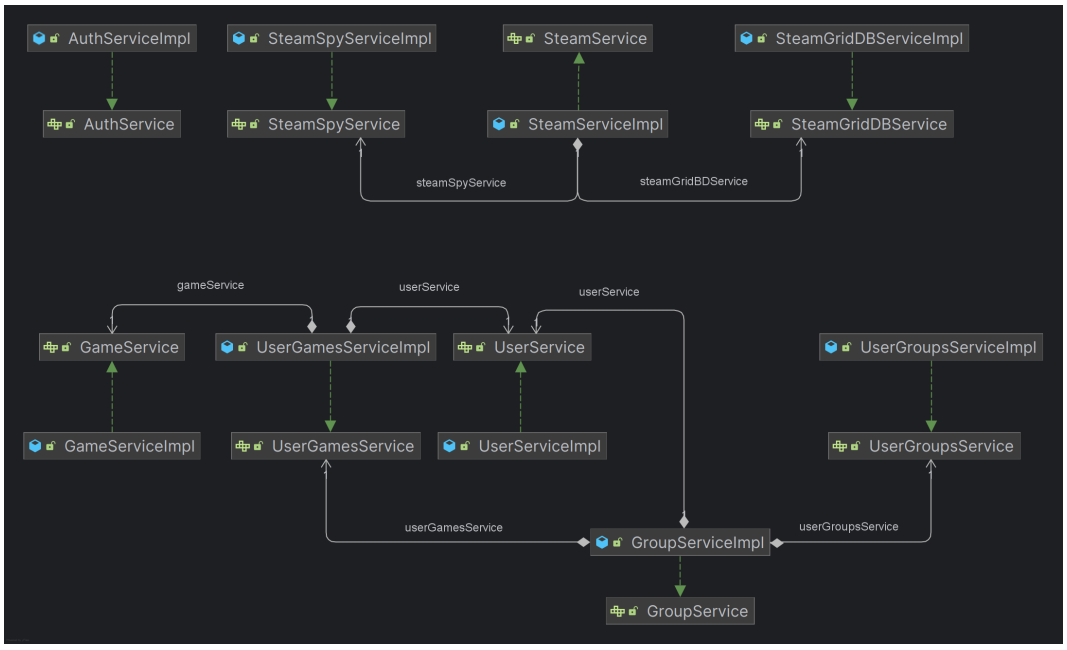


Figura 4.2: Arquitectura de los servicios

- **Controlador**
  Todas las solicitudes de la aplicación han sido organizadas en controladores y `APIs`, estructuradas según el microservicio correspondiente, para garantizar una separación clara de responsabilidades y mejorar la mantenibilidad.

  En la carpeta **api** (Figura 4.4a), hemos definido y expuesto las distintas rutas de nuestra aplicación, cada una asignada al microservicio pertinente.

Además, hemos integrado `Swagger` para generar automáticamente la documentación de la `API`. Esta documentación detalla cada una de las llamadas y sus respectivas descripciones. La documentación generada por `Swagger` se muestra en la figura 4.3.

Por otro lado, en la carpeta **rest** (Figura 4.4a), se encuentran los controladores principales de la aplicación. Estos controladores se encargan de mapear las solicitudes recibidas por la `API` y redirigirlas al método adecuado del microservicio correspondiente. Este mapeo garantiza que cada solicitud se procese de manera eficiente y que la lógica de negocio esté adecuadamente distribuida entre los distintos microservicios.

- **DTOs**
  Para manejar la transferencia de datos entre las distintas capas de la aplicación de manera eficiente y segura, hemos utilizado `DTOs` (*Data Transfer Objects*). Los *DTOs* se encargan de transportar únicamente la información necesaria en cada contexto específico, evitando exponer directamente las entidades del modelo de datos.

  Además, mejoran el rendimiento general de la aplicación, ya que permiten definir estructuras de datos optimizadas para cada operación, minimizando la carga de datos innecesarios.

  Podemos encontrarlos en la estructura del proyecto dentro de la carpeta **dto** (Figura 4.1).

- **Seguridad**
  Para gestionar la seguridad de la aplicación, se utilizó `Spring Security` con `JWT`, cuya configuración puede encontrarse en la carpeta **security**. Además, se aplicó un filtro a todas las peticiones entrantes para asegurar la autenticación en las rutas requeridas. Podemos encontrarlo en **filter**; ambas carpetas pueden observarse en la figura 4.1.

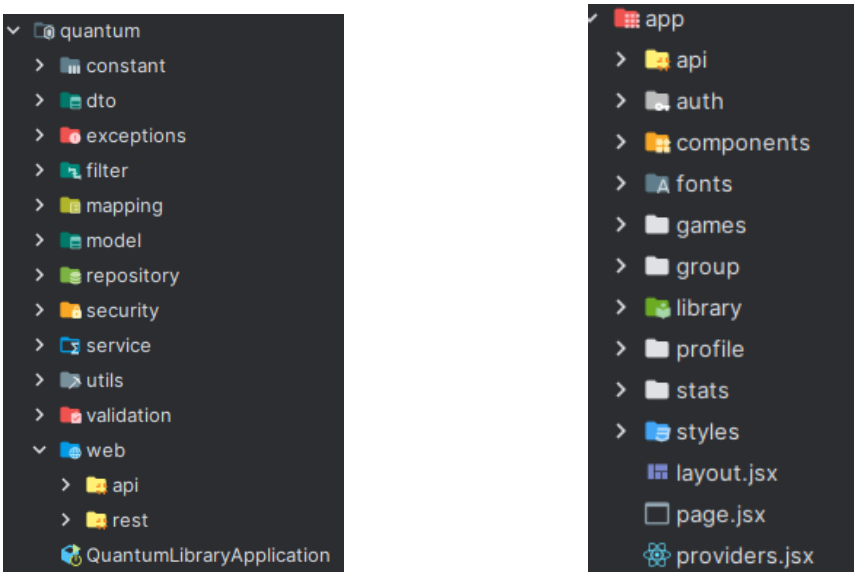
4.2.2. Frontend

El *frontend* sigue una estructura de diseño basada en `Next.js`, utilizando su sistema de enrutamiento por páginas. Esto significa que la indexación de las rutas se organiza mediante carpetas, cada una conteniendo un archivo **page.jsx**. Podemos observar la estructura del proyecto *front* en la figura 4.4b.

La aplicación incluye las siguientes carpetas principales: **games**, **group**, **library**, **profile** y **stats**. Cada una de estas carpetas define rutas específicas que proporcionan acceso a diferentes secciones del sitio:

- **games**: Página de la biblioteca global de juegos.

Figura 4.3: Documentación `API`



(a) Estructura del proyecto back (b) Estructura del proyecto front

Figura 4.4: Estructuras del proyecto

- **group**: Rutas relacionadas con las secciones de grupos.
- **library**: Rutas que corresponden a nuestra propia biblioteca.
- **profile**: Página dedicada a nuestro perfil de usuario.
- **stats**: Página que muestra las estadísticas.

Además, el archivo **page.jsx**, ubicado en la carpeta raíz, define el contenido de la página principal de la aplicación. Este enfoque estructurado facilita la organización y gestión del enrutamiento en el desarrollo del *frontend*.

Aunque la aplicación cuenta con un número limitado de rutas, la verdadera complejidad y profundidad radican en los componentes que conforman cada una de estas rutas. Al seguir una arquitectura de `SPA` (*Single Page Application*), se ha asegurado que todos los elementos de la página se carguen de manera óptima, lo cual es crucial dada la gran cantidad de datos que se manejan y muestran.

En cuanto a la gestión de la seguridad del *frontend*, utilizamos `NextAuth` para manejar la autenticación mediante credenciales y gestionar de manera segura el almacenamiento del token de sesión. `NextAuth` ofrece páginas predefinidas para el inicio de sesión; sin embargo, estas han sido reemplazadas por versiones personalizadas, ya que no proporcionan soporte nativo para la creación de nuevas cuentas, lo que requería una solución adaptada a nuestras necesidades.

4.2.3. Base de datos

Para almacenar los datos, se ha utilizado una base de datos relacional implementada con `MySQL`.

Con el fin de almacenar las grandes cantidades de información de las que se compone la aplicación, se han utilizado varias tablas con información mínima en cada una de ellas, donde lo importante son sus relaciones con otras tablas. De esta manera, se implementaron seis tablas. Al ser una biblioteca personal, la parte principal de la aplicación son los usuarios. Estos se almacenan en `Users_t`, y por otro lado, tenemos `Games_t`, que almacena la información intrínseca e inmutable de cualquier videojuego añadido a la aplicación, conformando así la biblioteca global. Ambas tablas constituyen `User_Games_t`, que almacena la información personal con la que cada usuario quiere personalizar un juego en su biblioteca personal. Tanto `Games_t` como `User_Games_t` requieren de tablas auxiliares adicionales (`Game_Tags_t` y `User_Game_Tags_t`) para almacenar los tags, debido al gran número de ellos que puede contener cada uno de los diferentes juegos.

Por otro lado, encontramos la parte social de la aplicación, los grupos, cuya información básica se almacena en `Groups_t`. Esta se relaciona junto a `Users_t` para formar `User_Groups_t`, que almacena la información de cada usuario con cada grupo al que ha sido invitado o se ha unido, además de qué votos ha realizado en cada uno de ellos. Además, se hace uso de las tablas `***_t_seq` para la creación de los IDs únicos.

Esta arquitectura está descrita en el esquema relacional de la figura 4.5.

4.3. Diseño e Implementación

4.3.1. Backend

La implementación del *backend* de la aplicación se ha diseñado para proporcionar una `API` robusta, optimizada para el manejo y cruce de múltiples datos y estadísticas. Se han desarrollado una serie de métodos especializados, algunos de los cuales siguen el patrón clásico de `CRUD (Create, Read, Update, Delete)`, mientras que otros implementan operaciones más complejas, que involucran múltiples llamadas a servicios externos para obtener y procesar datos relevantes.

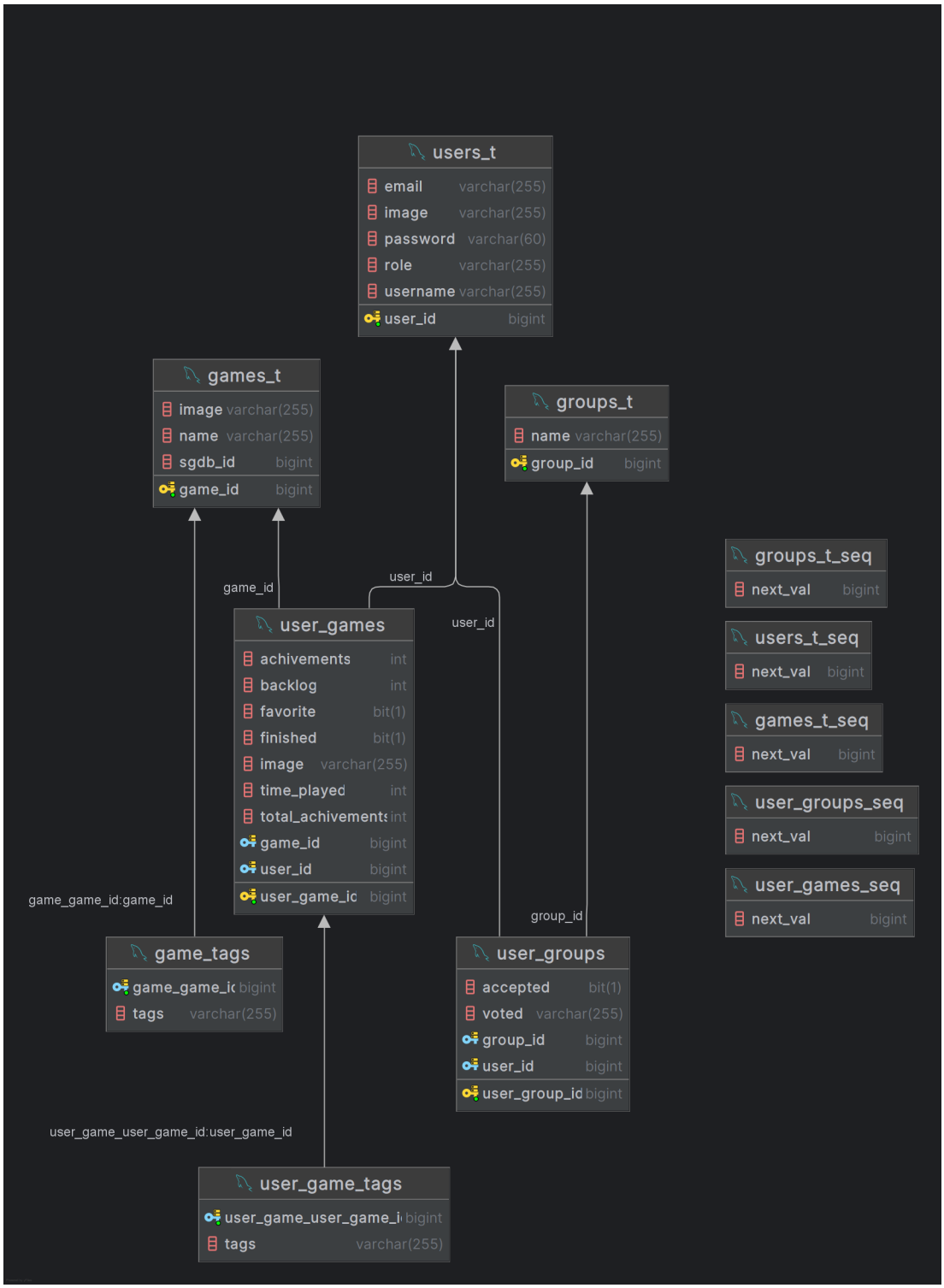


Figura 4.5: Esquema relacional

Principios SOLID

Durante el desarrollo de la aplicación, se ha hecho un uso exhaustivo del principio SOLID de Responsabilidad Única SRP (*Single Responsibility Principle*). Cada servicio se encarga de un único modelo y repositorio respectivo al nombre de su clase, manteniendo así la responsabilidad sobre el mismo. De igual manera, ocurre con nuestros controladores, que hacen uso de un único servicio. Esto mejora la organización, la legibilidad y la mantenibilidad del código.

Por otro lado, el principio Abierto/Cerrado OCP (*Open/Closed Principle*) se ha implementado al utilizar interfaces y un alto grado de polimorfismo (con un valor del 175.40 % según las métricas MOOD), lo que permite la expansión del sistema sin necesidad de modificar las clases existentes. Esto contribuye a un diseño flexible y extensible, donde nuevos métodos y servicios pueden ser integrados sin afectar el comportamiento ya establecido de la aplicación. Este enfoque favorece la evolución del sistema con el tiempo, cumpliendo con la visión de que el software debe estar abierto a la extensión, pero cerrado a la modificación.

Este último comparte una gran relación con el principio de Inversión de Dependencias DIP (*Dependency Inversion Principle*), según el propio Robert C. Martin, quien lo resumió en su documento técnico *Design Principles and Design Patterns* del año 2000, donde definió los principios SOLID [51]:

> "If the OCP states the goal of OO architecture, the DIP states the primary mechanism."

Es decir, si el principio Abierto/Cerrado OCP establece el objetivo de la arquitectura Orientada a Objetos, la inversión de dependencias DIP establece el mecanismo principal. De esta manera, todos nuestros servicios y controladores se comunican únicamente mediante las interfaces dispuestas entre ellos, cumpliendo así el principio.

La implementación de los principios SOLID no solo mejora la estructura del código, sino que también optimiza la escalabilidad y facilita la integración de nuevas funcionalidades en el futuro. Gracias a la adherencia a estas buenas prácticas, el sistema es más fácil de mantener, probar y extender, lo que contribuye significativamente a la resiliencia de la aplicación. Además, gracias al uso de técnicas como el encapsulamiento y el polimorfismo, se minimiza el impacto de posibles errores durante el desarrollo, mejorando así tanto la experiencia de desarrollo como la eficiencia del mismo.

Complejidad Ciclomática y Mantenibilidad

Las métricas de complejidad ciclomatica (*v(G)*) (Figura 4.6), introducidas por Thomas J. McCabe en 1976 en su artículo *A Complexity Measure* [52], reflejan un sistema bien diseñado y mantenible en términos de la cantidad de caminos linealmente independientes. El análisis de la complejidad ciclomatica del proyecto refleja una media (*v(G)avg*) que no supera un valor de 4 en la mayoría de los paquetes, lo que sugiere que los métodos individuales son sencillos y escalables. El pico de complejidad ciclomatica total de 177 en el paquete de implementación de servicios se debe al gran número de métodos implementados, aunque estos son relativamente simples. Esto refuerza la escalabilidad y mantenibilidad futura del sistema, alineándose con las mejores prácticas de desarrollo.

package	v(G)avg	v(G)tot
quantum	1,00	1
quantum.exceptions	1,00	11
quantum.filter	3,67	11
quantum.model	1,29	9
quantum.security	1,00	6
quantum.security.jwt	1,33	8
quantum.service.impl	2,68	177
quantum.utils	2,33	7
quantum.validation	3,00	9
quantum.web.rest	1,03	32
Total		**271**
Average	1,98	27,10

Figura 4.6: Complejidad Ciclomática

Eficiencia del Código y Documentación

El código escrito se ha documentado exhaustivamente, como se ilustra en la figura 4.7 en la columna `JLOC (Javadoc Lines of Code)`. Se ha utilizado `Javadoc` para detallar las funcionalidades y objetivos de cada método y clase, documentando así todo el proyecto para su fácil manipulación y mantenimiento a largo plazo. El paquete con mayor concentración de líneas de código es el de implementación de servicios, evidenciando la importancia de este módulo en la arquitectura general.

package	JLOC	LOC
		24
quantum	0	22
quantum.service	316	422
quantum.service.impl	414	1.686
quantum.security	0	90
quantum.security.jwt	0	77
quantum.validation	64	152
quantum.filter	0	72
quantum.utils	0	38
quantum.constant	21	36
quantum.dto.auth	3	83
quantum.dto.common	3	19
quantum.dto.game	6	109
quantum.dto.group	12	136
quantum.dto.sgdb	0	85
quantum.dto.steam	0	64
quantum.dto.steamSpy	0	25
quantum.dto.user	6	126
quantum.dto.userGames	15	242
quantum.dto.userGames.steamImp	6	53
quantum.dto.userGroups	6	52
quantum.exceptions	77	161
quantum.mapping	72	132
quantum.model	15	221
quantum.repository	119	171
quantum.repository.projections	0	8
quantum.web.api	221	829
quantum.web.rest	222	516
quantum.dto		
quantum.web		
Total	**1.598**	**5.651**
Average	59,19	201,82

Figura 4.7: Líneas de código

Métricas MOOD

El proyecto se ha evaluado usando las métricas `MOOD` (*Métricas de Diseño Orientado a Objetos*), definidas por Fernando Brito e Abreu y presentadas durante la conferencia *ECOOP'95 Workshop on Quantitative Methods for Object-Oriented Systems Development* en 1995 [53], que proporcionan una visión integral de la calidad del diseño orientado a objetos. Dichas estadísticas pueden ser observadas en la figura 4.8 y extendidas a continuación:

- **AHF (Attribute Hiding Factor)**: 97.20 %, lo que indica un fuerte encapsulamiento de los atributos.
- **AIF (Attribute Inheritance Factor) y MHF (Method Hiding Factor)**: Relativamente bajos (5.6 % y 6.34 %, respectivamente), lo que sugiere que los métodos y atributos heredados son limitados pero gestionados adecuadamente.
- **PF (Polymorphism Factor)**: Un valor alto del 175.40 %, atribuible al uso extensivo de interfaces, tanto para los servicios como para los controladores.

project	AHF	AIF	CF	MHF	MIF	PF
project	97,20%	5,60%	5,11%	6,34%	3,85%	175,41%

Figura 4.8: Métricas MOOD

Resiliencia y Manejo de Errores

Se ha puesto especial énfasis en la resiliencia y capacidad de depuración de la aplicación. Todos los métodos cuentan con registros claros y cohesivos, que indican la parte del proceso en la que se encuentran (servicio o controlador). En el caso del servicio, se da un pequeño resumen de la operación que van a realizar en la base de datos y, en ambos casos, se acompaña de un comentario más descriptivo de lo que se trata de hacer.

Aunque estos registros hacen más fácil rastrear dónde la aplicación ha producido un fallo, también se ha implementado un sistema de manejo de errores propio, que permite una descripción detallada de cada posible fallo, ya sea un error en el acceso a la base de datos o un fallo lógico (por ejemplo, un intento de añadir un recurso duplicado). Estas excepciones aseguran la consistencia en los códigos de error devueltos por la `API`, como se muestra en la figura 4.9. Todas estas excepciones extienden de una base genérica que define que, por cada error, se devuelva el código del error, el mensaje del error y el código HTTP correspondiente al fallo. Este diseño se alinea con el principio `SOLID` de Abierto/Cerrado.

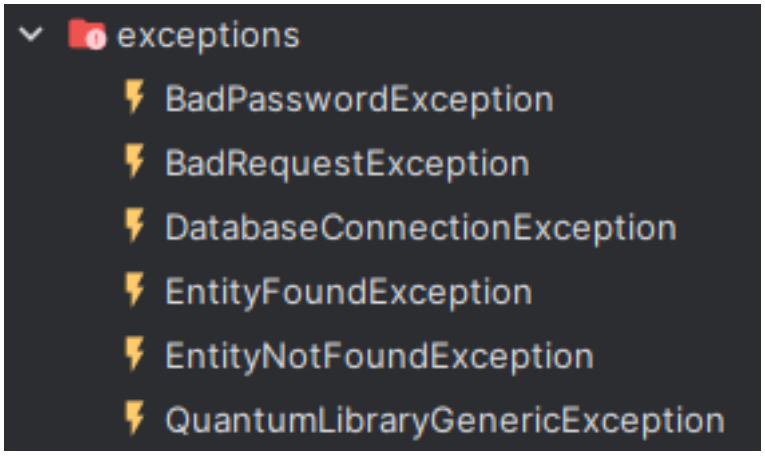


Figura 4.9: Excepciones

Gestión de Llamadas Complejas

Uno de los retos más significativos fue gestionar correctamente las llamadas relacionadas con los grupos. Estas operaciones requerían una arquitectura coherente, ya que ciertos métodos necesitaban acceder a información específica de los grupos de un usuario (almacenada en `userGroups`), mientras que otros debían operar a nivel global del grupo (como invitaciones y creación de grupos). Se resolvió este problema utilizando un diseño centralizado en el servicio `Group`, que realiza las operaciones relevantes delegando a `userGroup` solo aquellas necesarias. Esta decisión asegura la coherencia entre `API`, controladores, servicios y repositorios.

Implementación de Métodos Complejos

- **PostGroup**
  El proceso de creación de un nuevo grupo en la aplicación es una operación que, aunque conceptualmente sencilla, involucra una serie de pasos diseñados para optimizar tanto la integridad como la eficiencia del sistema. Al crear un grupo, es necesario no solo insertar una nueva entrada en la tabla `Group`, sino también generar dinámicamente instancias de `UserGroup` para cada invitación realizada. Para manejar esta relación de manera eficiente, se ha utilizado la propiedad *cascade all*. Esta configuración permite que las operaciones sobre la entidad `Group` se propaguen automáticamente a las entidades relacionadas, en este caso, `UserGroup`.

  Específicamente, al agregar las instancias de `UserGroup` a una lista dentro del objeto `Group` y realizar un único *save* en la base de datos, el mecanismo de persistencia se encarga de gestionar y sincronizar todas las entidades relacionadas sin necesidad de llamadas adicionales. Esto no solo simplifica

el código, sino que también mejora el rendimiento al minimizar las interacciones con la base de datos. Este enfoque aprovecha las capacidades de `ORM` (*Object-Relational Mapping*), garantizando consistencia y reduciendo el riesgo de errores relacionados con la inserción de datos. Podemos observar el código descrito en esta sección en el código 4.1.

```
@Override
public GroupResponse postGroup(String username, NewGroupBody body) {
    if (body.getName() == null || body.getName().isEmpty()) {
        throw new BadRequestException("Name is required");
    }

    // Generate new group
    Group newGroup = Group.builder().name(body.getName()).build();

    try {
        // Create the user group for the creator and the invited users
        User creator = userService.findUser(username);
        List<UserGroup> userGroups = new ArrayList<>();
        userGroups.add(UserGroup.builder()
                .user(creator)
                .group(newGroup)
                .accepted(true)
                .build());

        for (String invitedUser : body.getInvitedUsers()) {
            try {
                User user = userService.findUser(invitedUser);
                userGroups.add(UserGroup.builder()
                        .user(user)
                        .group(newGroup)
                        .accepted(false)
                        .build());
            } catch (EntityNotFoundException ex) {
                log.info("[SERVICE] - [USER GROUP INVITATION] - User not
    found: {}", invitedUser);
            }
        }

        // Save the user groups
        newGroup.setUserGroups(userGroups);

        log.info("[SERVICE] - [GROUP CREATION] - Saving group: {}",
    newGroup);
        newGroup = repository.save(newGroup);

    } catch (DataIntegrityViolationException ex) {
        throw new DatabaseConnectionException(DATA_INTEGRITY_ERROR, ex);
    } catch (JpaSystemException | QueryTimeoutException |
    JDBCConnectionException | DataException ex) {
        throw new DatabaseConnectionException(ex);
    }

    return mapper.map(newGroup);
}
```

Código 4.1: Crear un nuevo grupo

- **GetGroupGames**
  La operación para recuperar los juegos de un grupo representa uno de los procesos más complejos del *backend* debido a la cantidad de lógica y procesamiento que involucra. A diferencia de un simple conjunto de juegos

predefinido, los juegos asociados a un grupo se determinan dinámicamente en función de los juegos comunes que poseen los miembros que han aceptado la invitación al grupo.

Este proceso comienza obteniendo todos los `UserGroup` asociados al grupo. Una vez recuperados, se filtran aquellos usuarios que han aceptado la invitación, ya que solo ellos contribuyen a la lista de juegos comunes. Se realiza una búsqueda para determinar los juegos que todos estos usuarios tienen en común. Sin embargo, aquí no termina el trabajo; el sistema debe además contabilizar los votos que cada juego ha recibido dentro del contexto del grupo.

Para esto, se crea un mapa (estructura `Map`) donde cada clave corresponde al ID de un juego, y el valor asociado es una lista de usuarios que han votado por ese juego. Este mapeo se construye recorriendo todos los `UserGroup` y agregando las votaciones correspondientes. Finalmente, este mapa de votos se combina con la lista de juegos comunes. Para cada juego en la lista, se verifica si tiene una entrada en el mapa de votos y se agrega la información de los usuarios votantes a la respuesta final. Este procedimiento asegura que el resultado sea exhaustivo y preciso, optimizando tanto el procesamiento como la claridad de los datos devueltos. Podemos observar el código descrito en esta sección en el código 4.2.

```
@Override
public GroupGamesResponse getGroupGames(Long groupId) {
    // Search for the group
    Group group = findGroupById(groupId);

    // Gets user groups
    List<UserGroup> userGroups = group.getUserGroups();

    // Gets all accepted users in the group
    List<User> users = userGroupsService.findAcceptedUsersInGroup(groupId,
    true);

    // Gets the common games
    Set<Game> commonGames = userGamesService.getCommonGames(users);

    // Map of game IDs to users who voted for them
    Map<Long, List<String>> gameVotesMap = userGroups.stream()
            .flatMap(userGroup -> userGroup.getVoted().stream()
                    .map(gameId -> Map.entry(gameId, userGroup.getUser().
    getUsername()))
            )
            .collect(Collectors.groupingBy(Map.Entry::getKey,
                    Collectors.mapping(Map.Entry::getValue, Collectors.
    toList())));

    // Create the VotedGamesResponse using the gameVotesMap
    List<VotedGamesResponse> votedGamesResponses = commonGames.stream()
            .map(game -> {
                List<String> voters = gameVotesMap.getOrDefault(game.getId
    (), new ArrayList<>());
                return gameMapper.map(game, voters);
            })
            .collect(Collectors.toList());

    return GroupGamesResponse.builder()
            .group(group)
```

```
            .games(votedGamesResponses)
            .build();
}
```

Código 4.2: Obtener juegos de un grupo

- **Importar Juegos de Steam**
  La funcionalidad de importar juegos desde `Steam` es, sin lugar a dudas, la más exigente en términos de diseño y rendimiento de la aplicación. Esta operación se ha dividido en dos fases críticas: la obtención de los datos de juegos desde APIs externas y la importación masiva de juegos en la base de datos.

  En la primera fase, se realiza una llamada a la `API` de `Steam` para obtener la lista inicial de juegos de un usuario. Sin embargo, esta información es insuficiente. Para enriquecer los datos, se hace uso de la `API` de `SteamSpy` para obtener etiquetas oficiales (*tags*) y se consulta `SteamGridDB` para obtener carátulas personalizadas. Es importante filtrar las carátulas para asegurar que solo se utilicen aquellas con el tamaño correcto. Si no se encuentra una imagen válida o si no hay datos en `SteamGridDB`, se utilizan imágenes estáticas proporcionadas por `Steam`.

  Dado que un usuario puede tener bibliotecas que superen los 200 juegos, cada juego requiere múltiples consultas a APIs externas. Para mitigar el tiempo de respuesta, estas operaciones se ejecutan en paralelo, aprovechando la concurrencia para reducir el tiempo total a un promedio de 2 minutos para bibliotecas de 350 juegos. Esto se gestiona utilizando técnicas de procesamiento paralelo y asincronía, optimizando el rendimiento y la experiencia del usuario. Esta primera fase puede ser observada en el código 4.3.

```
@Override
public UserGamesImportList getGames(String steamId) {
    String apiUrl = EXTERNAL_API_URL + "IPlayerService/GetOwnedGames/v0001
    /?key=" + key + "&steamid=" + steamId;
    String response = webClient.get()
            .uri(apiUrl)
            .retrieve()
            .bodyToMono(String.class)
            .block();

    List<UserGameImport> userGames = new ArrayList<>();

    try {

        SteamResponse steamResponse = objectMapper.readValue(response,
    SteamResponse.class);

        steamResponse.getResponse().getGames().stream().parallel().forEach(
    game -> {
            SteamSpyGame steamSpyGame = steamSpyService.getSteamSpyInfo(
    game.getAppId());
            if (steamSpyGame == null) {
                return;
            }
            SGDBGame sgdbGame = getSGDBInfo(game.getAppId());
```

```
            String imageUrl = "https://cdn.cloudflare.steamstatic.com/steam
    /apps/" + game.getAppId() + "/library_600x900.jpg";
            if (sgdbGame != null) {
                List<SGDBGrid> sgdbGrids = getSGDBGrids(sgdbGame.getId());
                if (!sgdbGrids.isEmpty()) {
                    Optional<SGDBGrid> sgdbGrid = sgdbGrids.stream().filter
    (e -> e.getWidth() == 600 && e.getHeight() == 900).findFirst();
                    imageUrl = sgdbGrid.orElse(sgdbGrids.getFirst()).getUrl
    ();
                }
            }
            assert sgdbGame != null;
            UserGameImport newUserGame = UserGameImport.builder()
                    .name(steamSpyGame.getName())
                    .timePlayed(game.getPlaytime())
                    .image(imageUrl)
                    .sgdbId(sgdbGame.getId())
                    .tags(steamSpyGame.getTags().keySet().stream().toList()
    )
                    .build();
            userGames.add(newUserGame);
        });

    } catch (JsonProcessingException e) {
        throw new QuantumLibraryGenericException("Error parsing response
    from steam", e.getLocalizedMessage(), HttpStatus.BAD_REQUEST);
    }

    return UserGamesImportList.builder()
            .games(userGames)
            .build();
}
```

Código 4.3: Importar desde Steam

En la segunda fase, el usuario selecciona en el *frontend* los juegos que desea importar. Aquí comienza el proceso de validación y almacenamiento. Cada juego debe pasar varios filtros: si ya está presente en la biblioteca global, se referencia el existente; si no, se crea uno nuevo. Se verifica también si algún juego ya está en la biblioteca del usuario, lo que implica ignorar esos juegos durante la importación. En versiones anteriores, se realizaban múltiples búsquedas y guardados individuales, lo que resultaba ineficiente.

La versión actual implementa un mapa de juegos por sus IDs únicos de `SteamGridDB`, permitiendo recorrer solo una vez la lista de juegos globales y la biblioteca del usuario para eliminar las coincidencias. Los juegos restantes en el mapa se crean y guardan en la base de datos en una sola operación masiva. Esto reduce drásticamente las llamadas redundantes y optimiza la eficiencia del almacenamiento. Una vez completada la creación, los juegos importados se asocian con el usuario mediante la creación de entradas `UserGame` en una única transacción, asegurando así integridad y rendimiento. Podemos observar el código descrito en esta sección en el código 4.4.

```
public List<UserGame> postUserGames(String username, Map<Long, NewUserGameBody>
    gamesToImport) {
    // Get available games in ddbb
    List<Game> availableGames;
```

```
    try {
        availableGames = new ArrayList<>(gameService.getGames(Pageable.unpaged()
    ).getGames().stream().map(gamesMapper::map).toList());
    } catch (EntityNotFoundException e) {
        availableGames = new ArrayList<>();
    }
    List<Long> availableGameIds = availableGames.stream().map(Game::getSgdbId).
    toList();

    // Creates a copy of the games to import and removes games already in the
    database
    Map<Long, NewUserGameBody> gamesToCreate = new HashMap<>(gamesToImport);
    availableGameIds.forEach(gamesToCreate::remove);

    // Remove games already in the user library
    List<Long> userGames = getUserGames(username,    all    , Pageable.unpaged())
    .getGames().stream().map(e -> e.getGame().getSgdbId()).toList();
    userGames.forEach(gamesToImport::remove);

    // Create new games
    List<Game> newGames = new ArrayList<>();
    for (Map.Entry<Long, NewUserGameBody> entry : gamesToCreate.entrySet()) {
        Game newGame = Game.builder()
                .name(entry.getValue().getName())
                .tags(entry.getValue().getTags())
                .image(entry.getValue().getImage())
                .sgdbId(entry.getKey())
                .build();
        newGames.add(newGame);
    }

    // Add new games to available games
    availableGames.addAll(newGames);

    // Save new games
    gameService.postGames(newGames);

    // Search the user
    User user = userService.findUser(username);

    // Create new user games
    List<UserGame> newUserGames = new ArrayList<>();
    for (Map.Entry<Long, NewUserGameBody> entry : gamesToImport.entrySet()) {
        Game game = availableGames.stream().filter(e -> e.getSgdbId().equals(
    entry.getKey())).findFirst().orElse(null);
        if (game == null) continue;
        UserGame newUserGame = generateNewUserGame(entry.getValue(), user, game)
    ;
        newUserGames.add(newUserGame);
    }

    // Saves userGames
    try {
        log.info(   [SERVICE] - [USER GAME CREATION] - Saving {} user games ,
    newUserGames.size());
        return repository.saveAll(newUserGames);
    } catch (DataIntegrityViolationException ex) {
        throw new DatabaseConnectionException(DATA_INTEGRITY_ERROR, ex);
    } catch (JpaSystemException | QueryTimeoutException |
    JDBCConnectionException | DataException ex) {
        throw new DatabaseConnectionException(ex);
    }
}
```

Código 4.4: Guardar juegos

Seguridad

La seguridad de la aplicación se ha implementado mediante el uso de tokens `JWT`, configurando un esquema estándar de `Spring Security` para su validación y gestión, ilustrado en el código 4.5. Sin embargo, durante fases posteriores del desarrollo, se detectó una vulnerabilidad crítica: al realizar operaciones sobre un usuario especificado en la ruta, era posible manipular identificadores y ejecutar acciones sobre cuentas ajenas, siempre que se dispusiera de un *token* válido, aunque perteneciera a otro usuario.

Para resolver esta brecha de seguridad, se reestructuró la lógica de autenticación en la API. En concreto, se incorporó una función que aprovecha el filtro de seguridad generado por el token que extrae los detalles del usuario autenticado. Este proceso garantiza que todas las acciones sensibles se realicen únicamente sobre los recursos asociados al usuario autenticado, bloqueando así cualquier intento de acceso no autorizado a cuentas ajenas.

Por último, se añadió un simple sistema de roles que permite a usuarios preseleccionados realizar acciones sobre la biblioteca global de nuestra API.

```
@Override
    protected void doFilterInternal(HttpServletRequest request,
    HttpServletResponse response, FilterChain filterChain) throws
    ServletException, IOException {
        try {
            String jwt = parseJwt(request);
            if (jwt != null && JwtUtil.verifyToken(jwt)) {
                String username = JwtUtil.getUserNameFromJwtToken(jwt);
                UserDetails userDetails = userService.loadUserByUsername(
    username);
                UsernamePasswordAuthenticationToken authentication = new
    UsernamePasswordAuthenticationToken(
                        userDetails, null, userDetails.getAuthorities());
                authentication.setDetails(new WebAuthenticationDetailsSource().
    buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(
    authentication);
            }
        } catch (Exception e) {
            log.error("Cannot set user authentication: {}", e.
    getLocalizedMessage());
        }
        filterChain.doFilter(request, response);
    }
```

Código 4.5: Filtro de seguridad

4.3.2. Frontend

En la sección del *frontend* han sido destacables numerosas implementaciones y medidas que se han tomado para resolver distintos problemas encontrados a lo largo del desarrollo. A continuación, detallaremos las soluciones implementadas.

Jerarquía de Pantallas y Peticiones

Al tratarse de una `SPA` (*Single Page Application*), la página se encuentra dividida en múltiples componentes, lo que garantiza una carga dinámica y eficiente del contenido sin necesidad de recargar la página completa. Esta estructura modular permite una experiencia de usuario fluida y una gestión de estado más controlada.

Se distinguen cuatro tipos principales de componentes en la arquitectura del proyecto:

- **Layout:** Encapsula a todos los componentes hijos que se encuentren en su jerarquía.
- **Loading:** Componentes estáticos que se muestran de forma transitoria mientras se cargan los datos asociados a un componente principal. Estos proporcionan una experiencia de carga más fluida, evitando pantallas en blanco.
- **Page:** Componente que representa el contenido principal de cada ruta. En él se presenta la información concreta a la que se accede.
- **Reutilizables:** Componentes diseñados para ser utilizados en múltiples partes de la aplicación, lo que favorece la encapsulación, el mantenimiento y la reutilización del código.

Estructura Jerárquica de la Aplicación

La jerarquía completa de la aplicación se estructura de la siguiente manera:

- **Raíz de la Página**
  - **Layout:**
    - Contiene la meta-información global de la web.
    - Incluye el *Provider* 4.3.2 para la gestión de contexto global.
    - Integra el componente de navegación principal, disponible en toda la aplicación.
  - **Page:** Información general y resumen de *Quantum Library*.
    - **Ruta Perfil** (`/profile`)
      - **Page:** Visualización y edición de los datos de la cuenta del usuario autenticado.
    - **Ruta Estadísticas** (`/stats`)

- ◇ **Page:** Resumen y análisis de estadísticas de la biblioteca personal, incluyendo horas jugadas y logros obtenidos.

- **Ruta Biblioteca Global** (`/games`)
  - ◇ **Page:** Visualización de todos los juegos almacenados en la base de datos global de la aplicación.
- **Ruta Biblioteca Personal** (`/library/games/[category]`)
  - ◇ **Nota sobre `[category]`:** Esta parte de la ruta es dinámica. El valor introducido en la URL se recupera en el *frontend* mediante el *hook* *useParams* de `React` y se utiliza para filtrar los juegos por categoría. En caso de que se proporcione una categoría no reconocida, la aplicación devuelve todos los juegos del usuario por defecto.
  - ◇ **Layout:** Menú lateral vertical que proporciona acceso rápido a todas las categorías disponibles.
  - ◇ **Page:** Muestra la biblioteca personal del usuario para la categoría seleccionada. (Figura 4.10)
- **Ruta Grupos** (`/group/[groupId]/games`)
  - ◇ **Page:** Visualiza la biblioteca de juegos en común de los miembros de un grupo específico. La parte `[groupId]` de la ruta se recupera dinámicamente y se utiliza para obtener la información correspondiente al grupo en cuestión.

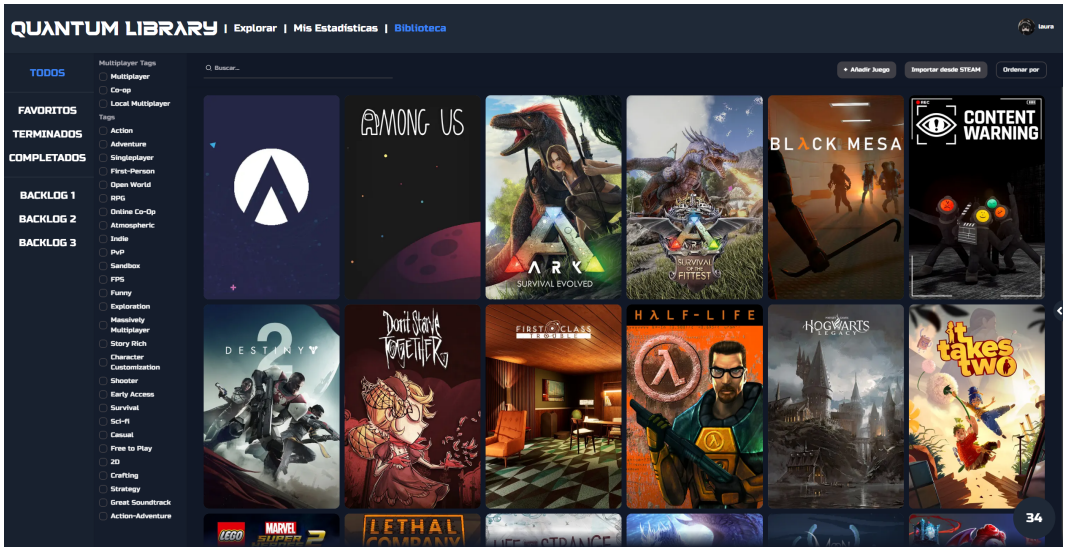


Figura 4.10: Página de biblioteca personal

Providers

Los *providers* permiten que los datos o funciones necesarias estén disponibles en cualquier parte de la aplicación sin necesidad de pasarlos manualmente de

un componente a otro, haciendo la arquitectura más modular y limpia. En el desarrollo de la aplicación, se ha hecho uso de dos *providers* fundamentales para facilitar la gestión de la sesión del usuario y la implementación de componentes de interfaz de usuario consistentes: el `SessionProvider` y el `NextUI Provider`.

- **SessionProvider** de `NextAuth` se encarga de proporcionar la información de la sesión a través de todos los componentes de la aplicación, lo que permite saber si hay un usuario registrado actualmente y proteger las rutas necesarias en caso negativo. En caso afirmativo, permite acceder a la información del usuario, como su nombre, su imagen o su dirección de correo electrónico.
- **NextUI Provider** facilita la integración y gestión de los componentes visuales de la aplicación proporcionados por la biblioteca `NextUI`. Además, facilita la configuración de temas y la personalización de estilos a nivel global.

Actualización dinámica de datos

A la hora de realizar cambios en la biblioteca de videojuegos, es fundamental garantizar que la información se mantenga sincronizada tanto en el servidor como en el cliente. Tradicionalmente, tras realizar una actualización en el servidor mediante una petición al *backend*, el cliente debería volver a solicitar todos los datos para reflejar los cambios. Sin embargo, esto generaría una sobrecarga innecesaria en el sistema y afectaría negativamente a la experiencia del usuario debido al aumento en el tiempo de respuesta.

Para evitar esta recarga completa de datos, se ha optado por un enfoque basado en la gestión eficiente del estado dentro del cliente utilizando las herramientas ofrecidas por **React**. Concretamente, todos los datos de la biblioteca son recuperados y almacenados en el componente padre, quien actúa como la fuente de verdad. Posteriormente, tanto los datos como sus respectivas funciones de actualización son pasadas a los componentes hijos a través de *props*. Este patrón permite a los componentes hijos modificar los datos directamente a través de las funciones del padre, garantizando que los cambios se reflejen de manera inmediata en toda la aplicación sin la necesidad de nuevas peticiones al servidor.

La clave de este proceso radica en el uso del *hook useState* de `React` para la gestión del estado. Cuando un componente hijo invoca la función de actualización proporcionada por el padre, `React` gestiona el re-renderizado de forma eficiente, provocando únicamente la actualización del componente padre y sus descendientes afectados. Esto se debe a que `React`, gracias a su *virtual DOM*, compara la representación previa del *DOM* con la actual y actualiza únicamente las partes que han cambiado. Esta estrategia minimiza el coste computacional y optimiza

el rendimiento de la aplicación. Como resultado, conseguimos cumplir con uno de nuestros requerimientos no funcionales: optimizar las pantallas de nuestra web y reducir al máximo los tiempos de carga para mejorar la experiencia del usuario.

Podemos analizar en detalle este comportamiento observando los siguientes fragmentos de código:

En el fragmento de código 4.6, correspondiente al archivo `page.jsx` de la ruta de la biblioteca personal, se observa cómo se realiza la obtención inicial de los datos desde el *backend*. Posteriormente, estos datos son inyectados en el componente `UserContentDisplay` mediante *props*, estableciendo la fuente de verdad en el componente padre.

```
// Get Games Data
const url = 'api/user/' + session.user.username + '/games?category=' + params.
    category;
const data = await GET(url, session.user.token, ['games']);

// Get Groups Data
const urlGroups = 'api/user/' + session.user.username + '/groups';
const dataGroups = await GET(urlGroups, session.user.token);

return <UserContentDisplay data={data.games} gData={dataGroups || { accepted:
    [], pending: [] }} />;
```

Código 4.6: Obtención de datos para biblioteca personal en frontend

En el fragmento 4.7, el componente `UserContentDisplay` convierte los datos recibidos a estado reactivo mediante *hooks* de `React`. Este paso es clave, ya que es el que permite que cualquier modificación se propague automáticamente por toda la jerarquía de componentes sin necesidad de nuevas peticiones al servidor.

```
export default function UserContentDisplay({ data, gData }) {
  const [games, setGames] = useState(data);
  const [filteredGames, setFilteredGames] = useState(games.sort((a, b) => a.game
    .name.localeCompare(b.game.name)));
  const [searchParam, setSearchParam] = useState('');
}
```

Código 4.7: Conversión de datos a estado reactivo en UserContentDisplay

Los *hooks* definidos en el componente padre son pasados a componentes hijos a través de *props*. En los fragmentos 4.8 y 4.9, se aprecia cómo componentes como `UserGameCard` y `EditUserGameModal` reciben *setGames* para actualizar el estado directamente desde niveles más profundos de la jerarquía.

```
{filteredGames?.map((entry) => (
    <div key={entry.game.id}>
      <UserGameCard entry={entry} setGames={setGames} />
    </div>
))}
```

Código 4.8: Anidación en UserContentDisplay

```
{session && <EditUserGameModal userGame={entry} setGames={setGames} session={
    session} />}
```

Código 4.9: Anidación en UserGameCard

Finalmente, en el fragmento 4.10, se muestra el proceso de edición de un juego en el componente `EditUserGameModal`. Aquí se utiliza la función *setGames* para actualizar el estado de la biblioteca personal. Gracias a la naturaleza reactiva de *useState*, `React` se encarga de re-renderizar únicamente los componentes afectados, evitando recargas innecesarias de toda la vista.

```
const editForm = async (onClose) => {
  const formURL = `api/user/${session.user.username}/games/${userGame.game.id}`;
  const requestBody = { tags, timePlayed, image, favorite, finished };

  try {
    const res = await PATCH(formURL, session.user.token, requestBody);
    setGames((prevGames) => prevGames.map((game) => (game.game.id === res.game.
    id ? res : game)));
    onClose();
  } catch (error) {
    console.error('Error al editar el juego');
  }
};
```

Código 4.10: Actualización de juego en frontend

Integración con APIs

El uso de APIs externas ha tenido un impacto significativo en la calidad de nuestro *frontend*, especialmente en la integración con la `API` de `SGDB` (*Steam Grid DB*). Esta `API` ha permitido implementar una barra de búsqueda con funcionalidad de autocompletado. Para ello, ofrece un *endpoint* que, dado un texto de búsqueda, devuelve las entradas cuyo nombre coincide o se asemeja al texto ingresado (Figura 4.11).

Gracias a esta funcionalidad, la implementación del *hook* **useAsyncList** y el uso de *signal* para gestionar las peticiones, el usuario puede visualizar los resultados de su búsqueda mientras sigue escribiendo, sin necesidad de presionar *Enter*. Este comportamiento mejora la experiencia del usuario al ofrecer resultados en tiempo real. El código para realizar esta búsqueda asíncrona se puede observar en el fragmento de código 4.11.

```
let list = useAsyncList({
    async load({ signal, filterText }) {
      try {
        if (filterText) {
          let res = await GETSIGNAL('api/sgdb/search?term=' + filterText,
    session.user.token, signal);
          return {
            items: res.data,
          };
        } else {
          return {
            items: [],
          };
        }
      } catch (error) {
        console.error('Error:', error);
      }
    },
```

Figura 4.11: Componente de búsqueda dinámica

```
  });
```

Código 4.11: Barra de búsqueda asíncrona

Además, esta integración ha permitido la carga dinámica de portadas para los videojuegos en los modales. Cuando un usuario hace clic en un juego, se abre el modal correspondiente y, en ese momento, se realiza una solicitud a la `API` de `SGDB` para recuperar todas las portadas disponibles en la plataforma (Figura 4.12).

Si el usuario selecciona una portada diferente de la primera opción, al abrir el modal se calcula y muestra qué número de las portadas disponibles corresponde a la imagen seleccionada. Este comportamiento mejora la interacción al permitir que el usuario vea una variedad de opciones y elija la portada que prefiera. Ambos comportamientos se pueden observar en el siguiente fragmento de código 4.12.

```
const getGrids = async (key) => {
    const formURL = `api/sgdb/getGrids/${key}`;
    let res = await GET(formURL, session.user.token);
    let filteredGrids = res.data.filter((item) => item.width === 600 && item.
    height === 900).map((item) => item.url);
    setGrids(filteredGrids);
    return filteredGrids;
  };

  useEffect(() => {
    if (isOpen) {
      // Evita hacer la peticion si ya se han obtenido las car tulas
      if (grids.length !== 0) {
        let index = grids.findIndex((grid) => grid === game.image);
        if (index !== -1) setImageKey(index);
        else setCustomImage(game.image);
      } else {
```

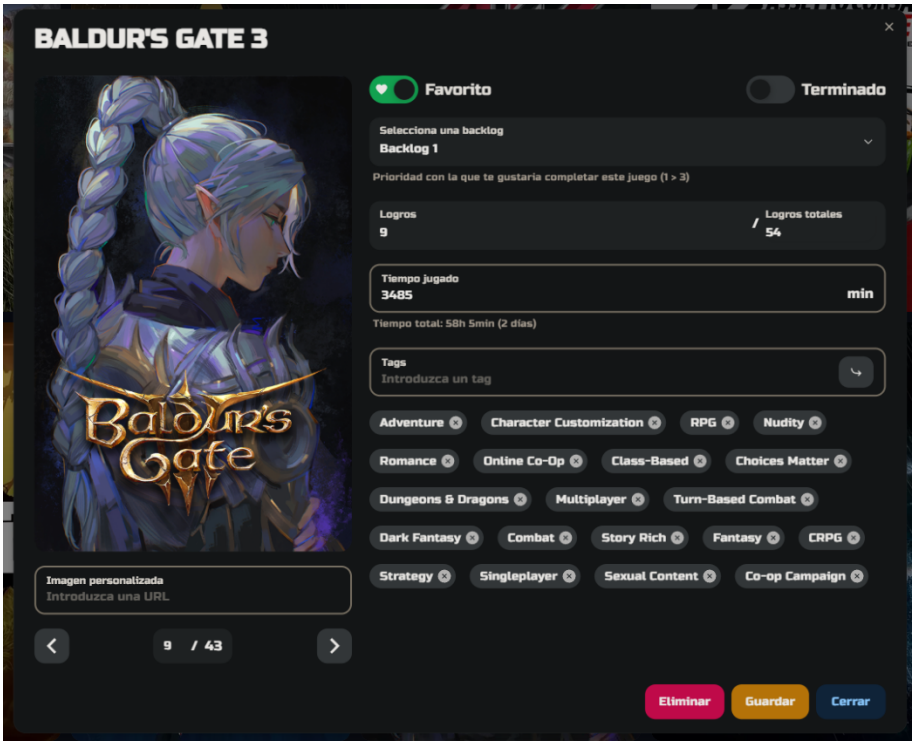


Figura 4.12: Modal de información de juego en la biblioteca

```
        // Obtener caratulas
        getGrids(game.sgdbId).then((filteredGrids) => {
          let index = filteredGrids.findIndex((grid) => grid === game.image);
          if (index !== -1) setImageKey(index);
          else setCustomImage(game.image);
        });
      }
    }
  }, [isOpen]);
```

Código 4.12: Obtención de carátulas en frontend

Votación grupal

Una parte esencial del proyecto es el componente social que permite compartir tu biblioteca con otros usuarios, mostrando los títulos en común que tienen todos los miembros del grupo y que incluyen algún elemento cooperativo o multijugador. A partir de esta funcionalidad, se incorpora un sistema de votación que permite decidir de forma colaborativa a qué juego jugar, evitando disputas entre los participantes (Figura 4.13).

Esta funcionalidad ha sido posible gracias a que el *backend* devuelve, junto a cada juego, una lista de los usuarios que han votado por él. De este modo, en el *frontend* únicamente es necesario mapear dicha información para mostrar un icono sobre cada juego indicando quién lo ha votado. Además, se resalta visual-

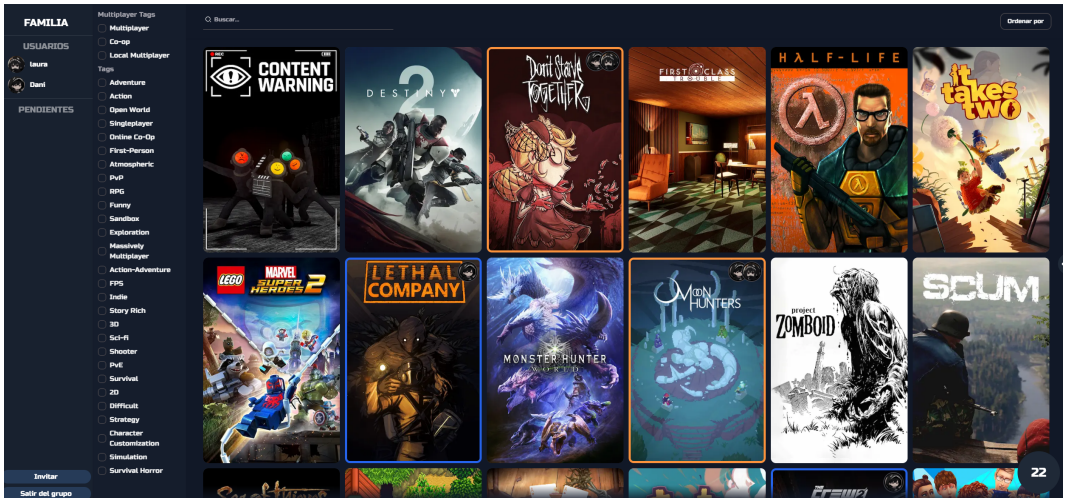


Figura 4.13: Página grupal

mente cada título según el votante: en color naranja si hemos votado nosotros, y en azul si ha votado otro usuario. Para emitir un voto sobre un juego, basta con hacer clic sobre él. Asimismo, se ha implementado una funcionalidad para seleccionar un juego de manera aleatoria basada en los votos: al hacer *hover* sobre el número de juegos disponibles, aparece un botón que permite ejecutar esta acción. Si hay un juego con mayoría clara, se abrirá su modal directamente; en caso de empate, se seleccionará uno al azar entre los más votados. Este comportamiento puede observarse en el código 4.13.

```
const randomGame = () => {
  // Encuentra el n mero m ximo de votos
  const maxVotes = Math.max(...filteredGames.map((game) => game.votes.length));

  // Filtra los juegos que tienen el n mero m ximo de votos
  const mostVotedGames = filteredGames.filter((game) => game.votes.length ===
    maxVotes);

  // Selecciona un juego aleatorio de entre los m s votados
  const randomIndex = Math.floor(Math.random() * mostVotedGames.length);
  setRandom(mostVotedGames[randomIndex]?.id);
};
```

Código 4.13: Selección aleatoria del juego más votado

4.3.3. Base de Datos

La base de datos se divide en distintas tablas generadas mediante JPA a partir de los modelos dados. Estas han sido diseñadas para optimizar el espacio y el tiempo de respuesta al realizar una petición, de modo que todas las consultas sean lo más livianas posible. A continuación, se describe cada una de ellas, su propósito, sus distintas columnas y cómo están relacionadas unas con otras.

- **games_t** Esta tabla almacena los juegos que han sido agregados por algún

usuario sin duplicados (basándose en el ID de `SGDB`).

- **game_id**: Identificador único.
- **name**: Nombre del juego.
- **image**: Enlace de la imagen. Se obliga a cargar una desde `SGDB` (*SteamGridDatabase*) para evitar imágenes maliciosas en la biblioteca global.
- **sgbd_id**: Identificador único de la `API` **games_t** en `SGDB`.

- **users_t** Esta tabla almacena los usuarios registrados con toda su información.
  - **user_id**: Identificador único.
  - **email**: Correo electrónico del usuario.
  - **image**: Enlace de la imagen de perfil.
  - **password**: Contraseña del usuario cifrada.
  - **role**: Rol del usuario.
  - **username**: Nombre único del usuario.
- **groups_t** Esta tabla almacena los grupos creados con sus respectivos nombres.
  - **group_id**: Identificador único.
  - **name**: Nombre del grupo.
- **user_groups** Esta tabla almacena los usuarios unidos a algún grupo y su relación con ellos, además de los juegos que votan en cada grupo. En esta tabla se decidió almacenar una lista con los IDs de los juegos votados en lugar de referencias a otras tablas, con el objetivo de optimizar las consultas y facilitar su gestión. Dado que los juegos comunes del grupo pueden cambiar al invitar a un nuevo miembro, este método permite encontrar coincidencias de IDs de manera más eficiente, en lugar de manejar referencias directas.
  - **user_group_id**: Identificador único.
  - **accepted**: Booleano que indica si el usuario ha aceptado la invitación al grupo.
  - **voted**: Lista de *IDs* de juegos votados en el grupo.
  - **group_id**: Clave foránea con relación **muchos a uno** con la tabla **groups_t**.
  - **user_id**: Clave foránea con relación **muchos a uno** con la tabla **users_t**.
- **user_games** Esta tabla almacena todos los juegos pertenecientes a algún usuario, junto con su información correspondiente.

- **user_game_id**: Identificador único.
- **achievements**: Número de logros conseguidos.
- **backlog**: Nivel de prioridad en la lista de pendientes.
- **favorite**: Booleano que indica si el usuario considera el juego como favorito.
- **finished**: Booleano que indica si el usuario ha terminado el juego.
- **image**: Enlace de la imagen del juego.
- **time_played**: Tiempo total jugado.
- **total_achievements**: Número total de logros del juego.
- **game_id**: Clave foránea con relación **muchos a uno** con la tabla **games_t**.
- **user_id**: Clave foránea con relación **muchos a uno** con la tabla **users_t**.

- **user_game_tags** Esta tabla auxiliar almacena los *tags* asociados a cada juego de los usuarios en la tabla **user_games**.
  - **user_game_id**: Identificador único y clave foránea con relación **muchos a uno** con la tabla **user_games**.
  - **tags**: Lista de etiquetas sin repeticiones.

4.4. Gestión de la Calidad del Proyecto

En esta sección se presentan las estrategias y herramientas utilizadas para evaluar y mejorar la calidad de la aplicación. Se detallará la cobertura de pruebas implementadas y los análisis estáticos y de seguridad que se han realizado.

4.4.1. Pruebas

Para garantizar una correcta cobertura de nuestra aplicación, se han implementado *tests* de integración con nuestra `API`, asegurando que las respuestas sean consistentes y correctas. También se han desarrollado pruebas específicas para las APIs externas utilizadas, verificando su correcto funcionamiento y la estabilidad en el formato de los datos devueltos. Además, se han incorporado *tests* unitarios, que han sido clave para detectar posibles errores y asegurar que las modificaciones en el código no afecten la funcionalidad final de la aplicación (Figura 4.14).

Dado que el desarrollo de una aplicación desde cero implica grandes desafíos, especialmente cuando está dirigida a un público acostumbrado a interfaces modernas, intuitivas y optimizadas, se optó por un enfoque basado en la interacción

directa con usuarios reales. En lugar de depender exclusivamente de pruebas *end-to-end* tradicionales, que en este contexto resultaban más una carga que una herramienta efectiva, se implementó un sistema de `QA` centrado en la experiencia del usuario como parte integral del proceso de diseño y mejora continua.

Este enfoque permitió una evaluación constante de la usabilidad, recopilando *feedback* directo de los usuarios en escenarios reales. A medida que el diseño, las funcionalidades y la navegación de la aplicación se ajustaban según sus necesidades y preferencias, se garantizó que la interfaz fuera no solo funcional, sino también intuitiva y agradable. La interacción con usuarios reales resultó clave para detectar problemas de usabilidad, identificar mejoras en la navegación y validar la efectividad de las decisiones de diseño, asegurando que la aplicación se alineara con las expectativas del público objetivo. Este proceso de pruebas y ajustes continuos fue fundamental para la creación de una aplicación optimizada, adaptable y centrada en el usuario.

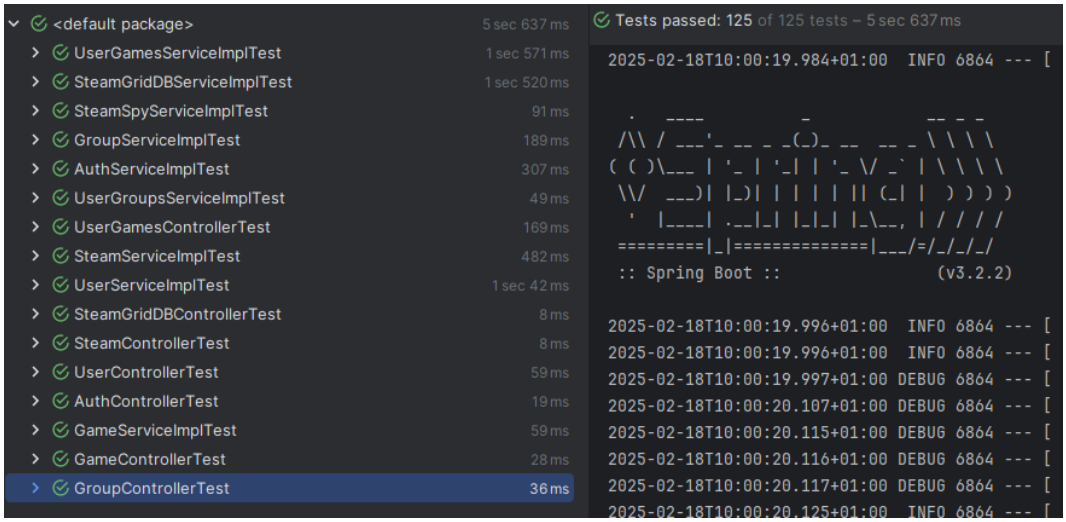


Figura 4.14: Número de tests totales

El conjunto de pruebas ofrece una cobertura total del código superior al 80 % (Figura 4.15), un porcentaje considerado suficiente dentro de la comunidad de desarrollo de software. Aumentar este porcentaje significativamente implicaría un uso excesivo de recursos sin aportar un beneficio proporcional, ya que probar todas las posibles casuísticas del código no siempre representa una ventaja real en términos de estabilidad y mantenimiento.

Dicho esto, se ha puesto un énfasis especial en la validación de los métodos propios implementados en nuestros servicios. Gracias a esto, se ha alcanzado un 100 % de cobertura en los métodos y más del 90 % en las líneas de código evaluadas, como se muestra en la figura 4.16. De la misma manera, los *tests* de integración aplicados a los controladores han logrado alcanzar casi un 100 % de cobertura (Figura 4.17).

Element	Class, %	Method, %	Line, %
quantum	86% (112/129)	82% (480/580)	81% (1105/1350)
repository	100% (0/0)	100% (0/0)	100% (0/0)
constant	100% (0/0)	100% (0/0)	100% (0/0)
web	100% (7/7)	100% (45/45)	99% (100/101)
service	100% (9/9)	100% (85/85)	91% (552/604)
model	100% (10/10)	92% (65/70)	90% (65/72)
dto	85% (67/78)	80% (249/311)	80% (249/311)
mapping	66% (4/6)	76% (10/13)	71% (86/121)
exceptions	100% (6/6)	57% (8/14)	64% (11/17)
security	83% (5/6)	54% (13/24)	60% (35/58)
QuantumLibraryApplication	100% (1/1)	50% (1/2)	33% (1/3)
utils	50% (1/2)	40% (2/5)	27% (3/11)
validation	50% (1/2)	33% (1/3)	9% (2/22)
filter	50% (1/2)	12% (1/8)	3% (1/30)

Figura 4.15: Cobertura total del código de la aplicación

impl	100% (9/9)	100% (85/85)	91% (552/604)
UserServiceImpl	100% (1/1)	100% (14/14)	100% (74/74)
GameServiceImpl	100% (1/1)	100% (10/10)	100% (55/55)
AuthServiceImpl	100% (1/1)	100% (5/5)	100% (22/22)
GroupServiceImpl	100% (1/1)	100% (17/17)	98% (106/108)
UserGroupsServiceImpl	100% (1/1)	100% (8/8)	89% (35/39)
SteamServiceImpl	100% (1/1)	100% (7/7)	88% (47/53)
SteamGridDBServiceImpl	100% (1/1)	100% (6/6)	85% (30/35)
SteamSpyServiceImpl	100% (1/1)	100% (4/4)	85% (12/14)
UserGamesServiceImpl	100% (1/1)	100% (14/14)	83% (171/204)

Figura 4.16: Cobertura total de la implementación de la aplicación

web	100% (7/7)	100% (45/45)	99% (100/101)
api	100% (0/0)	100% (0/0)	100% (0/0)
rest	100% (7/7)	100% (45/45)	99% (100/101)
GroupController	100% (1/1)	100% (10/10)	100% (26/26)
UserGamesController	100% (1/1)	100% (9/9)	100% (23/23)
UserController	100% (1/1)	100% (8/8)	100% (20/20)
GameController	100% (1/1)	100% (6/6)	100% (14/14)
SteamController	100% (1/1)	100% (4/4)	100% (4/4)
SteamGridDBController	100% (1/1)	100% (4/4)	100% (4/4)
AuthController	100% (1/1)	100% (4/4)	90% (9/10)

Figura 4.17: Cobertura total de los controladores de la aplicación

4.4.2. Análisis

Para mantener la calidad del código, se utilizó Sonar en su versión local, que ofrece análisis efectivos para identificar problemas comunes como *bugs*, *imports* no utilizados y posibles vulnerabilidades. Esta herramienta proporciona reportes detallados que permiten a los desarrolladores limpiar y optimizar el código de manera eficiente. Para finalizar el proyecto se aseguró que el código quedase sin reportes de Sonar (Figura 4.18). Sin embargo, es importante señalar que, aunque la versión local es útil, carece de la funcionalidad completa y la profundidad de análisis que ofrecen sus versiones web, las cuales requieren una suscripción.

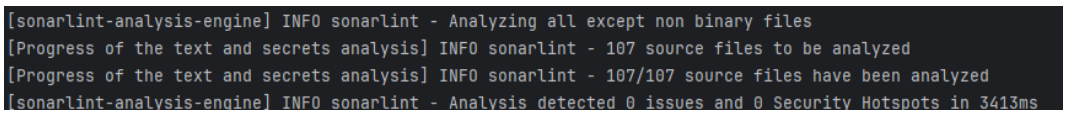


Figura 4.18: Reporte sonar

Para completar el análisis de vulnerabilidades, se emplearon las capacidades de `Qodana`, que proporciona un informe gratuito. Este informe, presentado en la figura 4.19, permite evaluar las posibles amenazas de seguridad y entender mejor las áreas de riesgo, aportando un valor complementario a las evaluaciones automatizadas.

4.5. Distribución y despliegue

En esta sección se detallan las implementaciones realizadas para la distribución y el despliegue de nuestra aplicación.

4.5.1. Integración continua

Para garantizar una integración continua, se ha utilizado GitHub, específicamente las funciones ofrecidas por `GitHub Actions` para crear una *Pipeline* (Figura 4.20) que ejecute las acciones necesarias hasta el despliegue de la aplicación. Esta herramienta permite automatizar la ejecución de los *tests* implementados y la correcta compilación del proyecto cada vez que se realiza una *pull request* en la rama *main* o mediante ejecución manual.

En este *Pipeline*, primero se construye la aplicación estableciendo las variables de entorno necesarias desde los secretos de GitHub y omitiendo los *tests* para verificar que la compilación se realice correctamente sin que estos interfieran en el proceso. A continuación, los *tests* se ejecutan por separado.

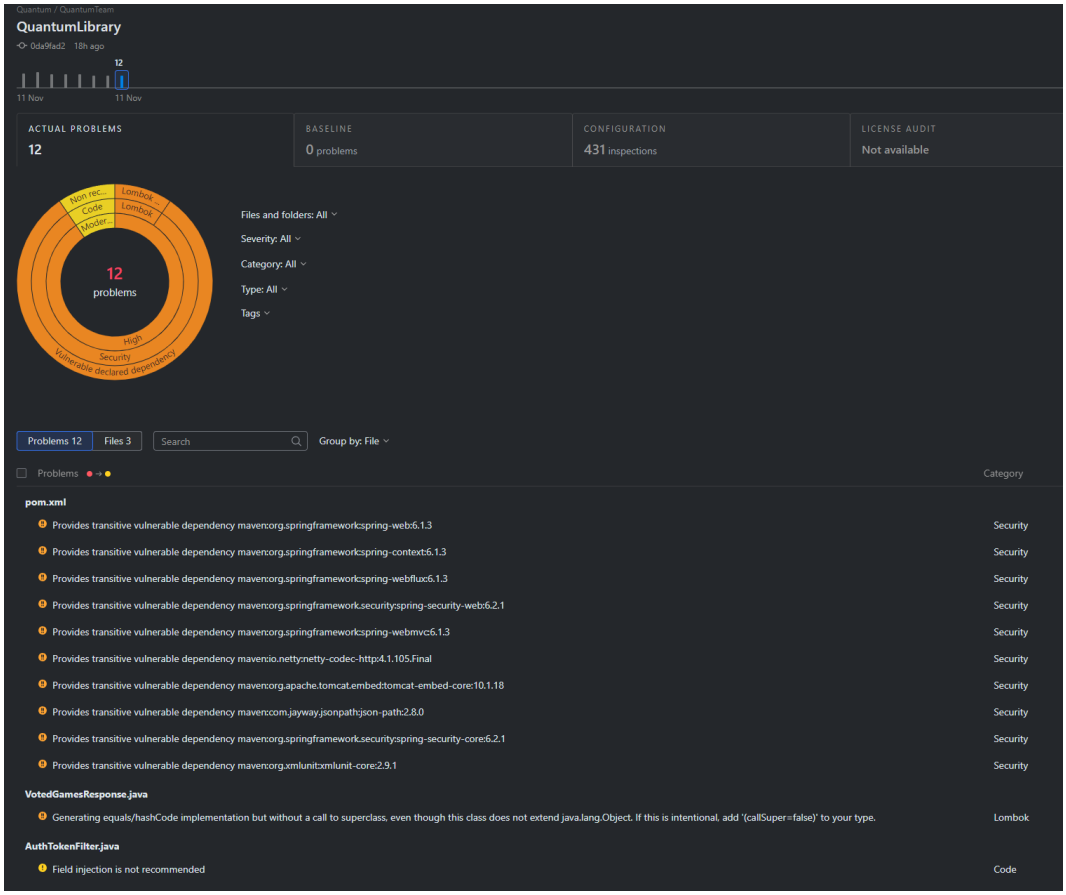


Figura 4.19: Análisis vulnerabilidades

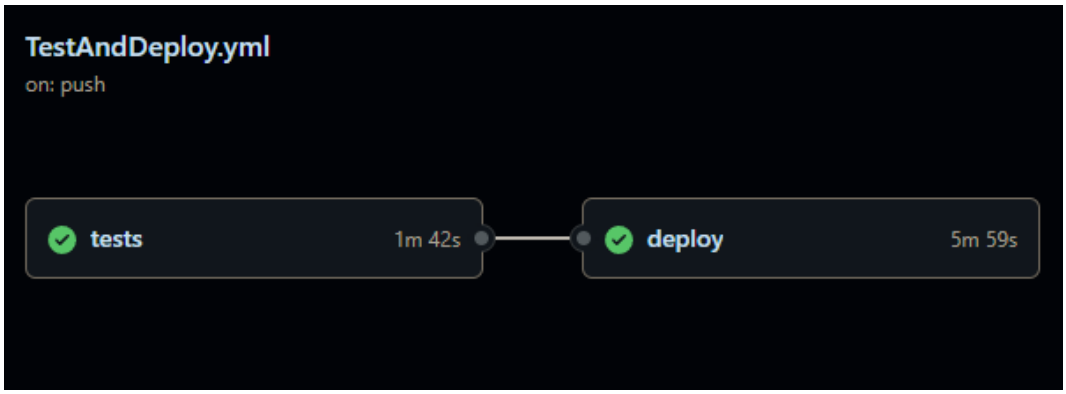


Figura 4.20: Pipeline de Github

Para su correcto funcionamiento, dichos *tests* requieren un servicio de `MySQL` instalado en la máquina virtual que GitHub crea durante la ejecución del *Pipeline*. Los detalles del *Action* pueden observarse en el código 4.14, y sus respectivos *jobs*, en la figura 4.21b.

```
tests:
  env:
    STEAM_API_KEY: ${{ secrets.STEAM_API_KEY }}
    STEAMDB_API_KEY: ${{ secrets.STEAMDB_API_KEY }}
    SB_SECURITY_PASSWORD: ${{ secrets.SB_SECURITY_PASSWORD }}

  runs-on: ubuntu-latest

  services:
    mysql:
      image: mysql:5.7
      env:
        MYSQL_ROOT_PASSWORD: root_password
        MYSQL_DATABASE: test_db
        MYSQL_USER: test_user
        MYSQL_PASSWORD: test_password
      ports:
        - 3306:3306
      options: >-
        --health-cmd=  mysqladmin   ping -h localhost -u root --password=
  root_password
        --health-interval=10s
        --health-timeout=5s
        --health-retries=3

  steps:
    - uses: actions/checkout@v4
    - name: Set up JDK 21
      uses: actions/setup-java@v4
      with:
        java-version: '21'
        distribution: 'temurin'
        cache: maven

    - name: Build with Maven
      run: mvn clean install -DskipTests

    - name: Run Tests
      env:
        DB_TEST_URL: jdbc:mysql://localhost:3306/test_db
        DB_TEST_USER: test_user
        DB_TEST_PASSWORD: test_password
      run: mvn test
```

Código 4.14: Action CI with MySql

4.5.2. Despliegue continuo

Una vez completada la fase de integración continua, se procede a construir, contenerizar y desplegar la aplicación. El proceso de despliegue comienza con la recuperación del código fuente, seguido de la configuración de la clave `SSH` para la conexión con la instancia `EC2`, utilizando un secreto almacenado en `GitHub`. A continuación, se instala `Docker` y `Docker Compose` en la máquina virtual que ejecuta el flujo de trabajo. Posteriormente, se inicia sesión en `Docker Hub` uti-

lizando las credenciales guardadas en los secretos de `GitHub`, y se instalan las dependencias necesarias para la aplicación.

Una vez preparadas todas las herramientas, el proceso continúa con el despliegue de la aplicación. Para realizar el despliegue se cuenta con dos archivos `Docker Compose`, uno se encarga de construir los contenedores de la aplicación (ver código 4.16) y el segundo se encarga de preparar las imágenes para su levantamiento (ver código 4.15), ambos archivos contienen información sensible, por ello, son almacenados en los secretos de GitHub. Primero se recupera de los secretos el `Docker Compose` encargado de la construcción de la aplicación, que se ejecutará con *docker-compose build* para construir los contenedores, los cuales se suben a Docker Hub bajo la etiqueta *latest*.

```
  backend:
    image: atom52025/quantum-library-backend:latest
    ports:
      -   8080  :8080
    environment:
      - DB_URL=jdbc:mysql://quantumlibrarydb.cf46mamakhyg.eu-west-3.rds.
    amazonaws.com:3306/QuantumLibrarySQL
      - DB_USERNAME=****
      - DB_PASSWORD=****
      - SB_SECURITY_PASSWORD=****
      - STEAM_API_KEY=****
      - STEAMDB_API_KEY=****
    networks:
      - app-network

  frontend:
    image: atom52025/quantum-library-frontend:latest
    ports:
      -   3000  :3000
    depends_on:
      - backend
    networks:
      - app-network

  nginx:
    image: nginx:latest
    container_name: nginx
    ports:
      -   80  :80
      -   443  :443
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
      - /etc/letsencrypt:/etc/letsencrypt
      - /var/lib/letsencrypt:/var/lib/letsencrypt
    depends_on:
      - frontend
      - backend
    networks:
      - app-network

networks:
  app-network:
    driver: bridge
```

Código 4.15: Script de Docker Compose Up

```
services:
  backend:
    build:
```

```
      context: .
      dockerfile: backend/Dockerfile
    image: atom52025/quantum-library-backend:latest
    ports:
      -   8080  :8080

  frontend:
    build:
      context: .
      dockerfile: frontend/Dockerfile
      args:
        NEXTAUTH_SECRET: ****
        NEXT_PUBLIC_EXTERNAL_API_URL: https://quantum-library.com
        NEXT_PUBLIC_INTERNAL_API_URL: http://backend:8080
        NEXTAUTH_URL: https://quantum-library.com
    image: atom52025/quantum-library-frontend:latest
    ports:
      -   3000  :3000
    depends_on:
      - backend
```

Código 4.16: Script de Docker Compose Build

Una vez que las imágenes han sido subidas y están listas, se establece la conexión `SSH` con la máquina `EC2` y se copia el archivo `Docker Compose` encargado de levantar la aplicación. En esta etapa, se descargan las versiones actualizadas de las imágenes desde `Docker Hub`, se eliminan las imágenes obsoletas y, finalmente, la aplicación se inicia en el entorno `EC2` con *docker-compose up*.

Además, en esta fase se introduce un nuevo contenedor definido dentro del archivo `Docker Compose`, el cual ejecutará una instancia de `Nginx`. Este contenedor actuará como *proxy* inverso, recibiendo todas las peticiones externas a través de `HTTPS` y redirigiéndolas al contenedor correspondiente según el caso. Este comportamiento está configurado y puede observarse en el código 4.17.

```
worker_processes 1;

events {
    worker_connections 1024;
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    # Redirigir HTTP a HTTPS
    server {
        listen 80;
        server_name quantum-library.com;

        # Certbot Challenge Route
        location /.well-known/acme-challenge/ {
            root /var/lib/letsencrypt;
        }

        # Redirect all HTTP traffic to HTTPS
        location / {
            return 301 https://$host$request_uri;
        }
    }

    # Configuracion de HTTPS con Certbot
```

```
    server {
        listen 443 ssl;
        server_name quantum-library.com;

        ssl_certificate /etc/letsencrypt/live/quantum-library.com/fullchain.pem;
        ssl_certificate_key /etc/letsencrypt/live/quantum-library.com/privkey.
    pem;

        # Exclude NextAuth routes on the backend
        location /api/auth/ {
            proxy_pass http://frontend:3000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }

        # Proxy requests to the frontend container
        location / {
            proxy_pass http://frontend:3000;  # Pass requests to frontend
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }

        # Proxy requests starting with /api/ to the backend container
        location /api/ {
            proxy_pass http://backend:8080;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
}
```

Código 4.17: Configuración Nginx

Esta separación en las fases de creación y levantamiento de las imágenes es clave para un despliegue exitoso. En versiones previas, se intentaba crear y lanzar los contenedores directamente en nuestra máquina `EC2`, lo que provocaba que la conexión se cerrara debido al alto tiempo de espera requerido para la construcción de la aplicación, causando así el fallo en el despliegue.

Nuevamente, podemos observar los detalles del *Action* en el código 4.18 y sus respectivos *jobs* en la figura 4.21a.

```
  deploy:
    runs-on: ubuntu-latest
    needs: tests
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up SSH key
        uses: webfactory/ssh-agent@v0.5.3
        with:
          ssh-private-key: ${{ secrets.EC2_SSH_KEY }}

      - name: Install Docker
        run: |
          curl -fsSL https://get.docker.com -o get-docker.sh
          sudo sh get-docker.sh

      - name: Install Docker Compose
        run: |
```

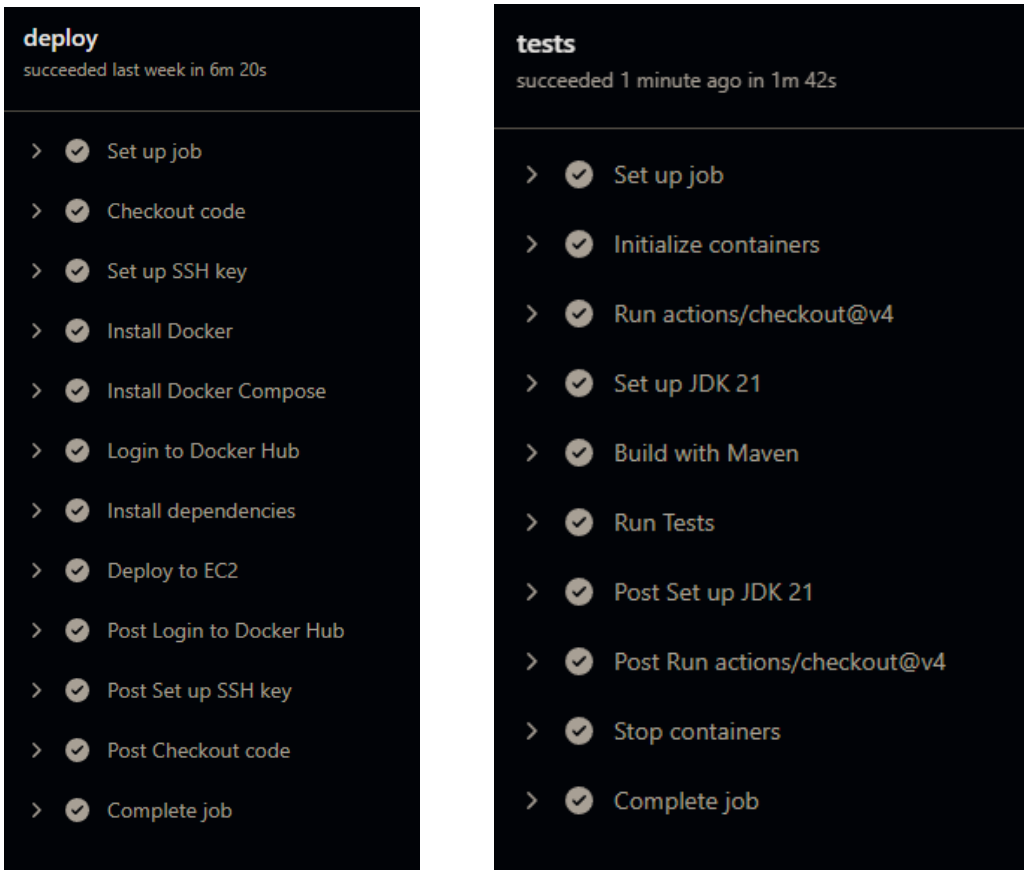
```
        sudo curl -L   https  ://github.com/docker/compose/releases/download/v2
    .3.3/docker-compose-$(uname -s)-$(uname -m)    -o /usr/local/bin/docker-
    compose
        sudo chmod +x /usr/local/bin/docker-compose
        docker-compose --version

      - name: Login to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Install dependencies
        run: npm install

      - name: Deploy to EC2
        run: |
          git fetch origin main
          git pull origin main
          echo   $  {{ secrets.DOCKER_COMPOSE_BUILD_FILE }}    > docker-compose.
    yml
          docker-compose build
          docker push atom52025/quantum-library-backend:latest
          docker push atom52025/quantum-library-frontend:latest
          ssh -o StrictHostKeyChecking=no ${{ secrets.EC2_USER }}@${{ secrets.
    EC2_HOST }} << 'EOF'
            echo   $  {{ secrets.DOCKER_COMPOSE_START_FILE }}    > docker-compose
    .yml

            docker-compose down

            docker pull atom52025/quantum-library-frontend:latest
            docker pull atom52025/quantum-library-backend:latest

            docker image prune -f
            docker-compose up -d
          EOF
        env:
          SSH_PRIVATE_KEY: ${{ secrets.EC2_SSH_KEY }}
```

Código 4.18: Action CD with EC2

4.5.3. AWS

En cuanto a los servicios contratados en `AWS` (*Amazon Web Services*), actualmente contamos con una máquina virtual `EC2` y una base de datos relacional `RDS`. Ambos servicios operan bajo la capa gratuita de `AWS` para minimizar costos, aunque ha sido necesario asumir el pago de la IP flexible que ofrece Amazon. Debido a esto, la capacidad de procesamiento de la máquina es limitada, lo que la hace vulnerable a caídas en caso de un tráfico simultáneo inesperadamente alto.

Para mitigar este riesgo, se han aplicado diversas optimizaciones mencionadas anteriormente. Bajo pruebas de carga con un uso razonable, la web ha demostrado ser estable, soportando sin problemas hasta cinco usuarios simultáneos. No obstante, algunos servicios automáticos externos a nuestro control de *scraping*



(a) Jobs de CD con MySql (b) Jobs de CI con MySql

Figura 4.21: Jobs ejecutados durante la pipeline de Github

web han llegado a generar desde 500 hasta 1000 peticiones simultáneas que provocan la saturación de la instancia EC2, impidiendo que se recupere por sí misma y siendo necesario un reinicio manual de la instancia.

El impacto de este problema puede analizarse a través de las estadísticas proporcionadas por `Cloudflare` en la figura 4.22 y en 4.23. Para reducir los efectos de estos ataques, se ha implementado temporalmente el "modo bajo ataque" de `Cloudflare`, que verifica la conexión de los usuarios antes de permitir el acceso. No obstante, esta es una solución provisional, y se continuará trabajando en estrategias más efectivas para gestionar y filtrar este tipo de peticiones masivas.

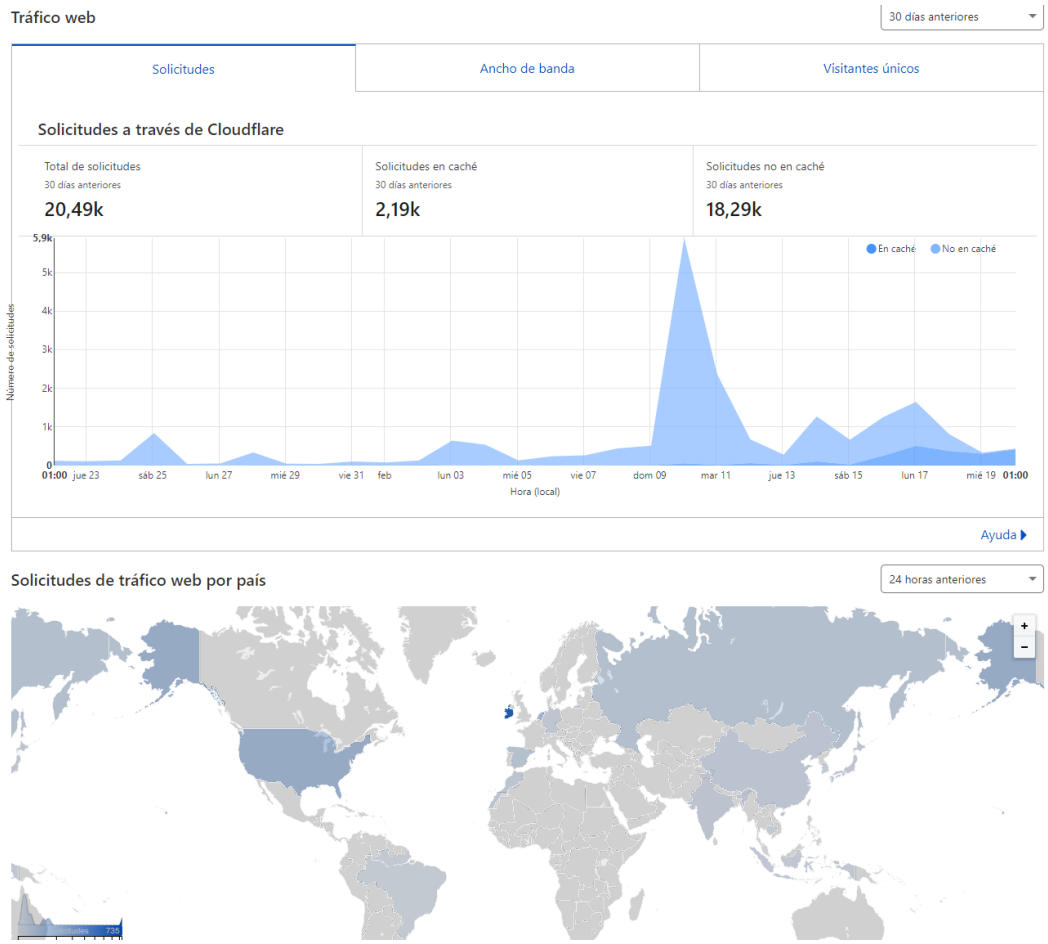


Figura 4.22: Estadísticas de tráfico ofrecidas por Cloudfare

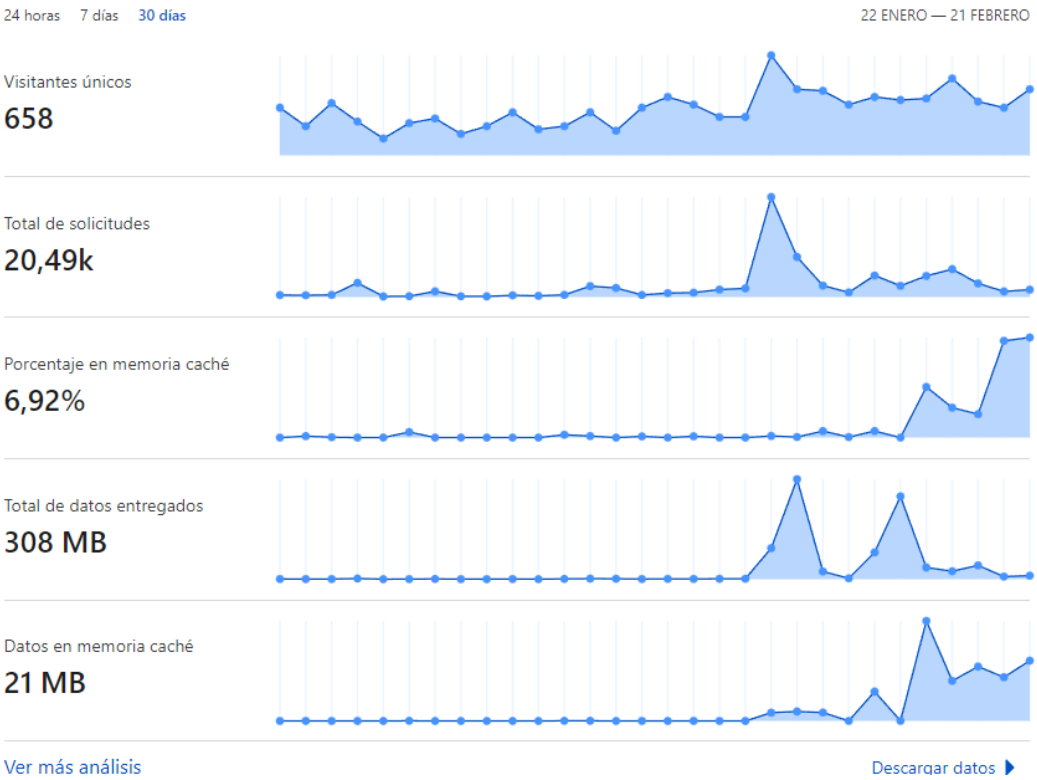


Figura 4.23: Estadísticas de solicitudes ofrecidas por Cloudfare

5

Conclusiones y trabajos futuros

5.1. Reflexión sobre el trabajo realizado

El desarrollo de esta aplicación ha sido extenso debido a la multiplicidad de decisiones de diseño e implementación que se han tomado a lo largo del proceso. Además, ha sido necesario un constante ciclo de evolución y refinamiento, lo que ha permitido mejorar y ajustar cada componente según los requisitos del proyecto y las necesidades de los usuarios. Esto ha supuesto que los objetivos definidos al principio del proyecto, aunque cumplidos en su totalidad, no supusieran la fotografía completa de la aplicación.

Finalmente, la primera versión de nuestra aplicación ha alcanzado todos los objetivos clave marcados al principio del desarrollo y los que se marcaron a lo largo del desarrollo, permitiendo así un uso completo de la misma. Pese a esto, la visión futura de nuestra aplicación abarca mucho más terreno del cubierto. A continuación, se detallan los principales avances conseguidos hasta el momento, así como los aspectos pendientes de implementación y ampliación en el futuro.

5.1.1. Objetivos cumplidos

El principal objetivo de este proyecto ha sido la creación de una plataforma de gestión de contenido multimedia, con un enfoque inicial en videojuegos. A través de un desarrollo modular y escalable, se ha logrado implementar una estructura que facilita la integración de nuevas categorías de contenido en el futuro, como

música, películas, animes y series. Además, se ha logrado una gran optimización de los recursos, tanto en `back` como en `front`, facilitando la escalabilidad de la aplicación, tanto horizontal, añadiendo nuevas secciones, como vertical, añadiendo aún más información en cada sección.

En el aspecto general de la aplicación, se ha logrado implementar un perfil personalizable para cada usuario, así como la funcionalidad de creación y gestión de grupos.

Dentro de la sección de videojuegos, se han desarrollado diversas características clave, como la biblioteca global y personal, la gestión de logros, la categorización y etiquetado de juegos, la personalización de las carátulas de cada título, el registro del tiempo de juego, la selección aleatoria de un título y la generación de estadísticas globales de la biblioteca personal. Además, se ha integrado la posibilidad de visualizar bibliotecas compartidas dentro de los grupos, lo que permite a los usuarios explorar y comparar sus colecciones con otros miembros de la comunidad.

En cuanto a la seguridad, aunque en un estado inicial, ha logrado la autenticación segura de usuarios mediante tokens `JWT`, así como la protección de rutas protegidas por autenticación. Además, esta seguridad se ve reforzada gracias a la implementación de seguridad `SSL`, tanto en el entorno web gracias a `Cloudflare` como en nuestra máquina gracias a `Certbot`.

Por último, es destacable la implementación *responsive* de nuestro `front-end`, permitiendo su correcta visualización independientemente del dispositivo que usemos para acceder a la web.

5.1.2. Aspectos pendientes y mejoras futuras

Sección de Videojuegos

Aunque la sección de videojuegos está operativa en esta versión, existen mejoras previstas para próximas iteraciones. Una de las principales es la integración con una *API* externa que recopile datos sobre los logros de cada videojuego, incluyendo el nombre, imagen, requisito y porcentaje de desbloqueo global en plataformas como *Steam*. Además, se añadirá la posibilidad de que los usuarios retiren logros de su biblioteca en caso de que no puedan desbloquearlos por alguna razón. Asimismo, se incluirá la opción de seleccionar logros específicos para mostrar en las estadísticas del usuario, creando así una especie de "estantería de trofeos".

En el futuro se pretende mejorar la descripción y la información mostrada sobre cada videojuego. En versiones futuras, se podrá visualizar información más detallada sobre cada juego, incluyendo el año de lanzamiento, la desarrolladora

y una descripción general del juego. Esta ampliación de la información permitirá ofrecer a los usuarios una visión más completa y rica de los juegos que tienen en su biblioteca.

Además, esta información será utilizada como base para el desarrollo de un sistema de recomendación de juegos. Este sistema analizará las métricas más relevantes, como los `tags` más jugados por el usuario y una pequeña encuesta previa para conocer sus preferencias, a fin de recomendar juegos que podrían interesarle en el futuro. De esta forma, se proporcionará un servicio personalizado que facilitará a los jugadores la selección de nuevos títulos para agregar a su colección, mejorando la experiencia global del usuario y fomentando el descubrimiento de nuevos juegos según sus gustos y hábitos de juego previos.

Se planea también añadir más métricas y estadísticas que el usuario podrá elegir mostrar según sus preferencias, permitiendo una personalización más detallada de la visualización de los logros y el rendimiento del jugador.

Nuevas integraciones

- **Metacritic:** *Metacritic* se presenta como una de las plataformas más influyentes en la recolección de puntuaciones y críticas sobre videojuegos, tanto de medios especializados como de usuarios. Su sistema de puntuación ofrece una visión general del rendimiento y la recepción de un título. La integración de *Metacritic* en nuestra plataforma proporcionaría a los usuarios acceso directo a calificaciones objetivas y detalladas, facilitando la comparación entre distintos títulos y enriqueciendo la experiencia de selección de nuevos juegos.

- **How Long to Beat:** `HLTB` (*How Long to Beat*) es una plataforma que proporciona información sobre el tiempo estimado necesario para completar un videojuego en sus diferentes niveles de finalización. Estos incluyen la historia principal, la historia junto con misiones secundarias y la finalización completa con todos los coleccionables. Esta es una de las métricas que más valor pueden ofrecer a nuestra plataforma, permitiendo a los usuarios filtrar juegos según el tiempo que requieren para completarlos, facilitando así la planificación de sus próximas partidas.

- **Nexus Mods:** *Nexus Mods* es la página líder en publicación de *Mods* creados por la comunidad. Una integración como esta permitiría al usuario ver una métrica de los *mods* disponibles para los juegos de su biblioteca, ofreciendo así una nueva oportunidad de rejugabilidad para títulos ya completados. Además, se planea habilitar un vínculo directo a la página donde el usuario haya creado su propio paquete de *mods*, en caso de que así lo haya hecho.

- **Epic Games:** *Epic Games* ofrece una *API* pública, aunque su complejidad y extensión requieren un análisis detallado para evaluar su viabilidad. Será necesario estudiar los datos proporcionados para determinar si la información disponible resulta útil y aplicable a nuestra plataforma.

- **Web Scraping:** Muchos de los servicios y plataformas actuales para la compra de videojuegos no disponen de una *API* que permita extraer información sobre los juegos disponibles o la biblioteca de cada usuario. Para solucionar esta limitación, en futuras versiones se implementará la técnica de *Web Scraping*, un método automatizado para la recolección de datos desde páginas web. Un ejemplo de plataforma donde podría aplicarse esta técnica es *PlayStation*.

Sección General

Una de las características más importantes de la aplicación es su capacidad para gestionar distintos tipos de contenido multimedia. Aunque en esta versión solo se ha trabajado en la sección de videojuegos, gracias a la implementación modular del sistema, será más sencillo añadir otras categorías, como música, películas, animes y series. En un principio, cada categoría será creada manualmente y estará disponible para cada usuario. Sin embargo, en fases posteriores se incluirá la opción de crear una biblioteca personalizada, donde el usuario podrá agrupar todo el contenido que desee, utilizando todas las funcionalidades previas y creando su propia biblioteca multimedia.

Al igual que en la actual sección de videojuegos, se añadirán distintas *APIs* externas que recojan información variada sobre cada contenido, aumentando así nuestra capacidad propia como *API*, que en un futuro se podrá distribuir como tal para su uso por otras aplicaciones.

Sección Social

Una de las áreas que más se potenciará en el futuro será la interactividad social. Se tiene la intención de permitir a los usuarios ver los perfiles de otros, siempre que estos lo deseen, y visualizar el contenido que cada uno quiera compartir. Por ejemplo, en la sección de videojuegos, los usuarios podrán mostrar sus estadísticas, así como sus estanterías con trofeos destacados. También se implementará la opción de sugerir contenidos que no estén presentes en un grupo, para que los usuarios puedan decidir si desean añadir ese contenido a sus bibliotecas personales.

Sección de Seguridad

La seguridad implementada en esta primera versión es básica, pero se han planificado varias medidas de mejora que se incluirán en versiones posteriores. Estas son algunas de las mejoras previstas:

- **Seguridad de contraseñas:** Se implementará un sistema para ayudar al usuario a elegir contraseñas seguras, mediante la regulación de los requisitos de complejidad.
- **Verificación por correo:** Se enviará un correo de confirmación al usuario al crear una cuenta, para garantizar la validez de la dirección de correo proporcionada.
- **Recuperación de credenciales:** Se habilitará una funcionalidad para recuperar las credenciales mediante correo electrónico, asegurando así el acceso a las cuentas en caso de olvido de la contraseña.

5.1.3. Conclusiones personales

Este proyecto ha sido un gran reto para su desarrollo en solitario debido a su amplitud y complejidad a largo plazo. Sin embargo, esto ha supuesto una gran mejora personal al resolver muchos de los problemas y casuísticas que han surgido durante el desarrollo. Esto ha supuesto una oportunidad para aplicar los conocimientos adquiridos durante mi formación académica y ampliar mis conocimientos y comprensión de las tecnologías y metodologías empleadas durante el desarrollo.

A lo largo del proceso, he comprendido la complejidad de desarrollar una aplicación dirigida a ser un producto completo, además de entender mejor las fases de dicho desarrollo y la satisfacción que supone construir una aplicación que, aunque en su versión inicial, tiene un gran potencial para crecer y evolucionar. Además, la experiencia de contar con usuarios dispuestos a ofrecer retroalimentación constante me ha permitido comprender mejor la importancia de la usabilidad y la experiencia del usuario para el éxito de una aplicación.

En conclusión, este proyecto no solo ha sido una valiosa experiencia educativa, sino también un reto personal que ha fortalecido mis habilidades técnicas y que, sin duda, ha contribuido a mi desarrollo profesional.

Bibliografía

[1] Oracle, "Java, lenguaje de programación," http://www.oracle.com/technetwork/java/index.html, 1995.

[2] Mozilla Contributors, "Javascript - programming language for the web," https://developer.mozilla.org/en-US/docs/Web/JavaScript, 2025, accessed: 2025-02-13.

[3] Spring Team, "Spring boot - build anything with java," https://spring.io/projects/spring-boot, 2025, accessed: 2025-02-13.

[4] Auth0, "Json web tokens - jwt.io," https://jwt.io/, 2025, accessed: 2025-02-13.

[5] Oracle Corporation, "Mysql - the world's most popular open source database," https://www.mysql.com/, 2025, accessed: 2025-02-13.

[6] MapStruct Contributors, "Mapstruct - java bean mapping, the easy way!" https://mapstruct.org/, 2025, accessed: 2025-02-13.

[7] Project Lombok, "Lombok - java library to reduce boilerplate code," https://projectlombok.org/, 2025, accessed: 2025-02-13.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[9] JUnit Team, "Junit 5 - a modern, powerful testing framework for java," https://junit.org/junit5/, 2025, accessed: 2025-02-13.

[10] Spring Team, "Webclient - non-blocking, reactive web client," https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webclient, 2025, accessed: 2025-02-13.

[11] Vercel, "Next.js - the react framework for production," https://nextjs.org/, 2025, accessed: 2025-02-13.

[12] Meta Open Source, "React - a javascript library for building user interfaces," https://react.dev/, 2024, accessed: 2025-02-12.

[13] MDN Web Docs, "Single page application (spa) - web development concepts," https://developer.mozilla.org/en-US/docs/Glossary/SPA, 2025, accessed: 2025-02-13.

[14] Vercel, "Image optimization in next.js," https://nextjs.org/docs/pages/building-your-application/optimizing/images, 2025, accessed: 2025-02-13.

[15] Google, "Largest contentful paint (lcp)," https://web.dev/lcp/, 2025, accedido: 2025-02-11.

[16] Lovell Fuller, "Sharp - high performance node.js image processing," https://sharp.pixelplumbing.com/, 2025, accessed: 2025-02-13.

[17] Tailwind Labs, "Tailwind css - a utility-first css framework," https://tailwindcss.com/, 2025, accessed: 2025-02-13.

[18] NextUI Contributors, “Nextui - beautiful, fast and modern ui library for react,” https://nextui.org/, 2025, accessed: 2025-02-13.

[19] Framer, “Framer motion - a production-ready motion library for react,” https://www.framer.com/motion/, 2025, accessed: 2025-02-13.

[20] React Icons Contributors, “React icons - popular react icons library,” https://react-icons.github.io/react-icons/, 2025, accessed: 2025-02-13.

[21] Chart.js Contributors, “Chart.js - simple html5 charts,” https://www.chartjs.org/, 2025, accessed: 2025-02-13.

[22] John C. May, “React chart.js 2 - react wrapper for chart.js,” https://github.com/reactchartjs/react-chartjs-2, 2025, accessed: 2025-02-13.

[23] NextAuth.js Contributors, “Nextauth.js - authentication for next.js applications,” https://next-auth.js.org/, 2025, accessed: 2025-02-13.

[24] Docker Contributors, “Docker - open source platform for developing, shipping, and running applications,” https://www.docker.com/, 2025, accessed: 2025-02-13.

[25] ——, “Docker compose - define and run multi-container docker applications,” https://docs.docker.com/compose/, 2025, accessed: 2025-02-13.

[26] Nginx, Inc., “Nginx - high-performance web server and reverse proxy,” https://www.nginx.com/, 2025, accessed: 2025-02-13.

[27] Valve Corporation, “Steam - digital distribution platform for video games,” https://store.steampowered.com/, 2025, accessed: 2025-02-13.

[28] ——, “Steam web api - access to steam community data,” https://steamcommunity.com/dev, 2025, accessed: 2025-02-13.

[29] SteamGridDB, “Steamgriddb api - search and retrieve game artwork and metadata,” https://www.steamgriddb.com/api, 2025, accessed: 2025-02-13.

[30] SteamSpy, “Steamspy - video game statistics and market data,” https://steamspy.com/, 2025, accessed: 2025-02-13.

[31] Amazon Web Services, “Aws - cloud computing services,” https://aws.amazon.com/, 2025, accessed: 2025-02-13.

[32] ——, “Amazon rds - relational database service,” https://aws.amazon.com/rds/, 2025, accessed: 2025-02-13.

[33] ——, “Amazon ec2 - scalable cloud computing service,” https://aws.amazon.com/ec2/, 2025, accessed: 2025-02-13.

[34] Cloudflare, Inc., “Cloudflare - web performance and security company,” https://www.cloudflare.com/, 2025, accessed: 2025-02-13.

[35] Electronic Frontier Foundation (EFF), “Certbot - easy automated certificate management for https,” https://certbot.eff.org/, 2025, accessed: 2025-02-13.

[36] Let’s Encrypt, “Let’s encrypt - free, automated, and open certificate authority,” https://letsencrypt.org/, 2025, accessed: 2025-02-13.

[37] GitHub, Inc., “Github - the platform for version control and collaboration,” https://github.com/, 2025, accessed: 2025-02-13.

[38] Axosoft, “Gitkraken - git gui for windows, mac linux,” https://www.gitkraken.com/, 2025, accessed: 2025-02-13.

[39] GitHub, Inc., “Github actions - automate, customize, and execute software development workflows,” https://github.com/features/actions, 2025, accessed: 2025-02-13.

[40] JetBrains, “Intellij idea - the most intelligent java ide,” https://www.jetbrains.com/idea/, 2025, accessed: 2025-02-13.

[41] The Apache Software Foundation, “Apache maven - build automation and dependency management tool,” https://maven.apache.org/, 2025, accessed: 2025-02-13.

[42] SmartBear Software, “Swagger - api design and documentation tool,” https://swagger.io/, 2025, accessed: 2025-02-13.

[43] Oracle Corporation, “Javadocs - official java documentation,” https://docs.oracle.com/en/java/, 2025, accessed: 2025-02-13.

[44] Prettier Contributors, “Prettier - an opinionated code formatter,” https://prettier.io/, 2025, accessed: 2025-02-13.

[45] Trivago, “Trivago prettier sort imports - prettier plugin for sorting imports,” https://github.com/trivago/prettier-plugin-sort-imports, 2025, accessed: 2025-02-13.

[46] SonarSource, “Sonarlint - ide extension for real-time code analysis,” https://www.sonarsource.com/products/sonarlint/, 2025, accessed: 2025-02-13.

[47] JetBrains, “Qodana - continuous code quality monitoring by jetbrains,” https://www.jetbrains.com/qodana/, 2025, accessed: 2025-02-13.

[48] DBeaver Team, “Dbeaver - universal database tool,” https://dbeaver.io/, 2025, accessed: 2025-02-13.

[49] Postman, Inc., “Postman - api development environment,” https://www.postman.com/, 2025, accessed: 2025-02-13.

[50] Adobe Inc., “Adobe photoshop - image editing and graphic design software,” https://www.adobe.com/products/photoshop.html, 2025, accessed: 2025-02-13.

[51] R. C. Martin, “Design principles and design patterns,” PDF disponible en línea, p. 12, 2000, accedido: 14 de noviembre de 2024. [Online]. Available: https://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf

[52] T. J. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976. [Online]. Available: http://www.literateprogramming.com/mccabe.pdf

[53] F. B. Abreu, “The mood metrics set,” in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 1995)*, 1995, p. 267.

Apéndices

A

Repositorio de GitHub

A.1. Enlace al repositorio de GitHub

Todo el código fuente puede ser accedido en el siguiente enlace:

https://github.com/Atom52025/QuantumLibrary.git

La pagina web puede ser visitada en:

https://quantum-library.com