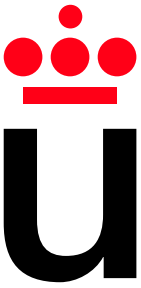

Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería Informática

Curso 2024-2025

Trabajo Fin de Grado

APLICACIÓN DE METODOLOGÍAS DE CALIDAD SOFTWARE EN EL DESARROLLO DE UN JUEGO WEB ONLINE

Autor: Miguel Serralta Jamard
Tutor: Michel Maes Bermejo

Agradecimientos

Quiero agradecer a mi tutor, Michel, por su dedicación, orientación y por ayudarme a definir unos criterios sólidos que me han permitido desarrollar un Trabajo de Fin de Grado con un enfoque profesional y de calidad.

También quiero expresar mi más sincero agradecimiento a mi padre, por su apoyo incondicional durante todo este proceso. Su presencia constante y sus palabras de ánimo en los momentos más difíciles, han sido un pilar fundamental para alcanzar esta meta.

Finalmente, agradezco a mi amigo Fredy, compañero de estudios en el ámbito de la ingeniería informática, por su cercanía, motivación y apoyo a lo largo de este camino compartido que ha sido estudiar la misma carrera.

Resumen

En este Trabajo de Fin de Grado se documenta el desarrollo completo de WordleApp, una aplicación web basada en el popular juego Wordle. El proyecto se ha abordado con una doble perspectiva: como desarrollador y como ingeniero de calidad, priorizando desde el inicio las buenas prácticas de desarrollo, la validación mediante pruebas automatizadas y el cumplimiento de estándares de calidad del software.

La aplicación ha sido desarrollada utilizando tecnologías ampliamente reconocidas en la industria, como Java 21, Spring Boot, MySQL y Selenium, siguiendo una metodología de desarrollo iterativa e incremental con enfoque TDD (Test Driven Development). A lo largo del proyecto se ha aplicado control de versiones con GitHub, integración continua con GitHub Actions y análisis estático del código con SonarCloud, cumpliendo así con los requisitos exigidos para garantizar la mantenibilidad, la extensibilidad y la fiabilidad de la aplicación.

Un aspecto clave del trabajo ha sido la cobertura de pruebas, incluyendo pruebas unitarias, de integración y de interfaz de usuario, además del uso de herramientas como Jacoco para medir cobertura y asegurar que el código cumple con las métricas de calidad. Asimismo, se ha utilizado Docker y Docker Compose para facilitar el despliegue de la aplicación, asegurando su portabilidad y reproducibilidad en distintos entornos.

Este proyecto demuestra cómo el enfoque sistemático en la calidad del software, junto con una arquitectura modular y bien estructurada, permite construir una aplicación funcional, robusta y lista para ser desplegada en producción.

Palabras clave:

- Wordle
- Java
- Javascript
- MySQL
- Docker
- GitHub
- SonarCloud
- Selenium
- TDD
- QA
- Pruebas automatizadas
- REST

Índice de contenidos

Índice de figuras

Índice de códigos

1 Introducción

1.1. Contexto y alcance

En el contexto actual del desarrollo de aplicaciones web, cada vez es más relevante no solo construir soluciones funcionales, sino también garantizar su calidad, fiabilidad y facilidad de mantenimiento.

El presente Trabajo de Fin de Grado (TFG) tiene como objetivo desarrollar una aplicación web basada en el juego Wordle, integrando en todo el ciclo de vida del software un enfoque orientado a la calidad del código y la validación exhaustiva mediante pruebas automatizadas.

El desarrollo de *WordleApp* se ha llevado a cabo de manera individual, y ha permitido aplicar de forma práctica los conocimientos adquiridos a lo largo del Grado en Ingeniería Informática, incluyendo el diseño de arquitecturas web, el uso de herramientas modernas de desarrollo, así como la implementación de pruebas y métricas de calidad del software.

El proyecto también ha servido como una experiencia de aprendizaje sobre buenas prácticas de ingeniería del software, desde el diseño modular, hasta la entrega e integración continua y el análisis de calidad estática del código.

1.2. WordleApp: contexto del juego

Wordle es un juego de palabras que alcanzó gran popularidad en 2022. Su dinámica consiste en adivinar una palabra secreta de cinco letras en un máximo de seis intentos, recibiendo como pista la posición correcta e incorrecta de cada letra mediante un sistema de colores.

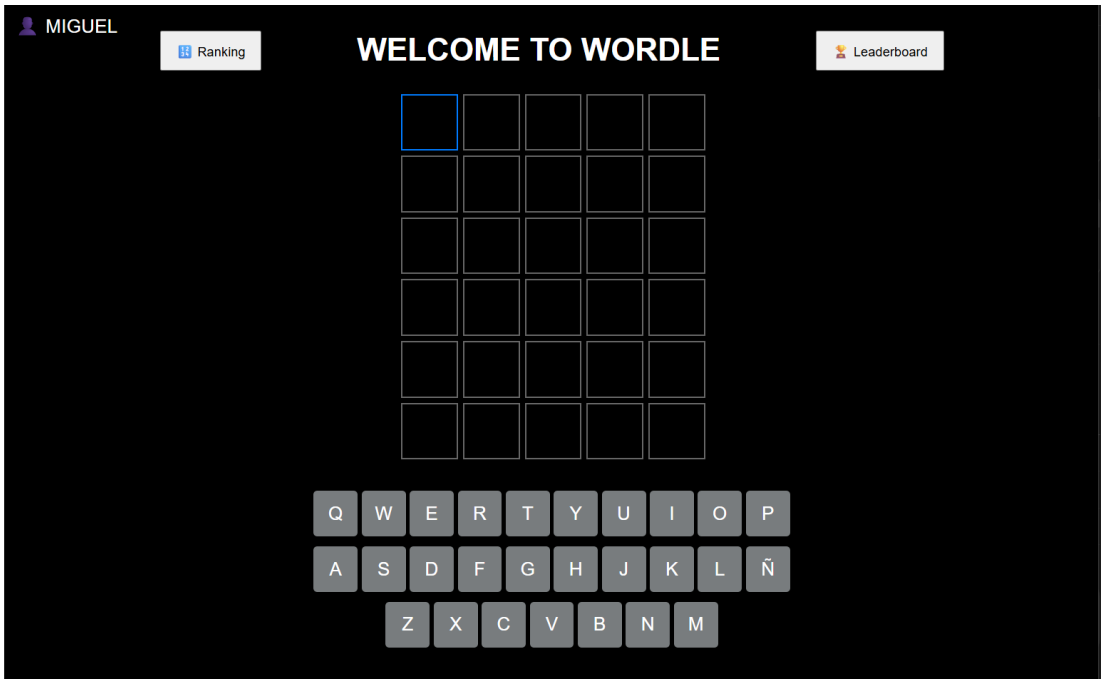

Figura 1.1: Tablero principal del juego WordleApp.

El éxito del juego ha propiciado múltiples adaptaciones y clones, lo que lo convierte en una opción atractiva como base para un proyecto académico.

WordleApp toma como referencia la versión clásica del juego, pero la adapta para funcionar como aplicación web personalizada, con un sistema de puntuación, rankings y validación contra diccionario.

1.3. Motivación del proyecto

La motivación principal de este proyecto reside en combinar la figura del desarrollador y del ingeniero de calidad en un entorno realista.

A lo largo del desarrollo, se ha puesto especial énfasis en aplicar principios de Test Driven Development, pruebas de regresión, medición de cobertura y análisis de deuda técnica, buscando no solo un resultado funcional, sino un producto técnicamente sólido.

Figura 1.2: Wordle original.

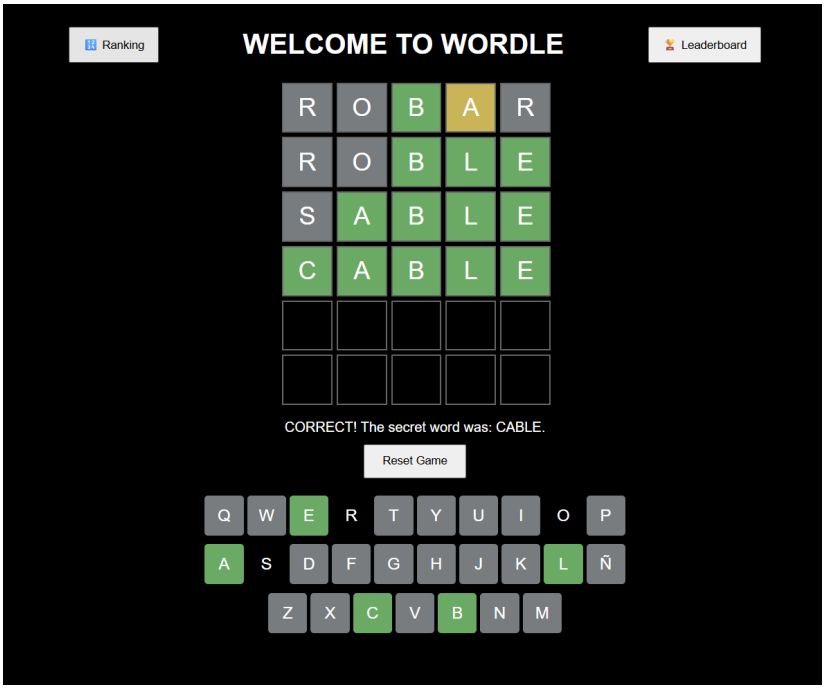

Figura 1.3: Palabra acertada en WordleApp.

Además, se ha integrado el despliegue de la aplicación usando Docker, se ha configurado un flujo de integración y entrega continua con GitHub Actions, y se ha realizado un análisis de calidad con SonarCloud.

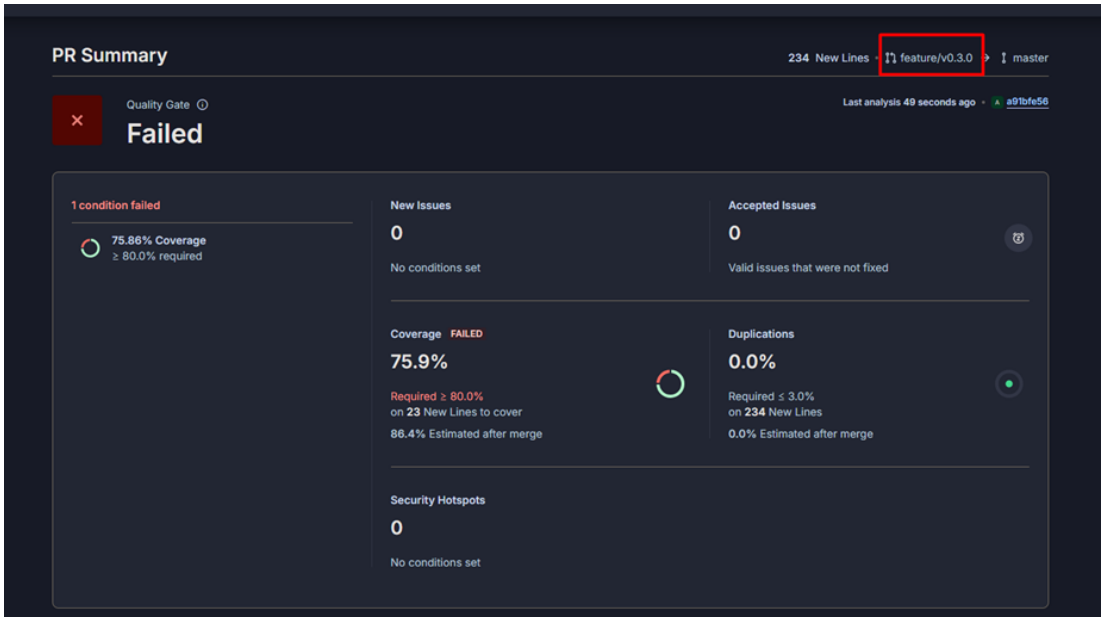

Figura 1.4: Panel de SonarCloud mostrando una cobertura de pruebas incompleta.

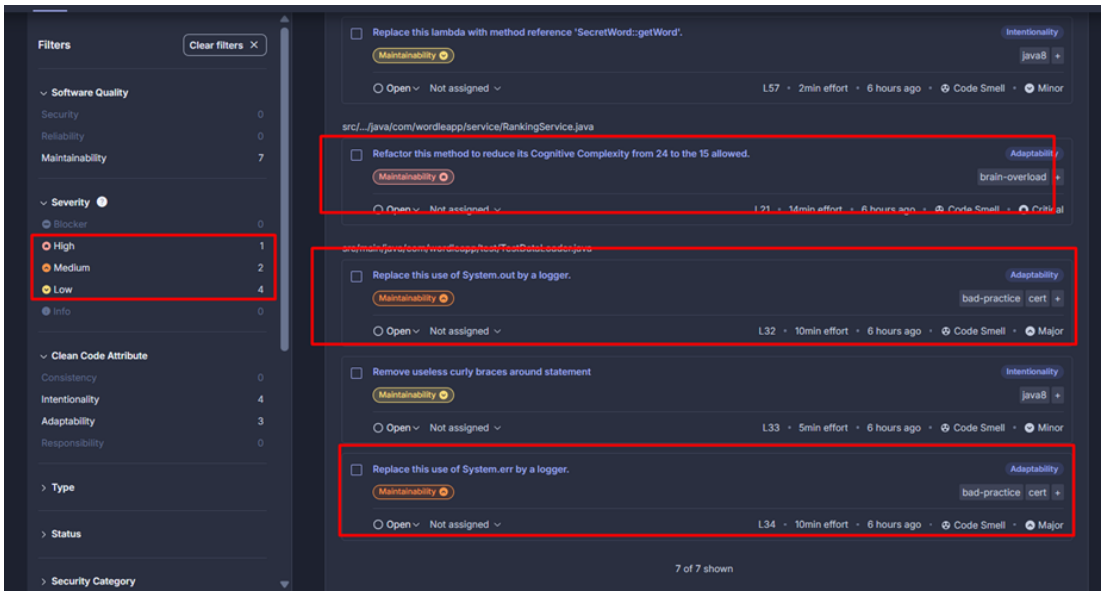

Figura 1.5: Indicadores de deuda técnica en SonarCloud.

Todo ello ha contribuido a simular un entorno de desarrollo profesional, donde cada cambio de código debe cumplir con ciertos estándares para poder ser integrado.

El proyecto también ha sido una oportunidad para reforzar competencias personales como la autonomía, la planificación por iteraciones, la gestión de versiones, y la documentación técnica del trabajo realizado.

1.4. Justificación del TFG

Este proyecto se ajusta a los objetivos del TFG propuestos por la Escuela Técnica Superior de Ingeniería Informática, en tanto que combina el desarrollo de una aplicación funcional (WordleApp) con el uso de tecnologías actuales (Spring Boot, MySQL, Selenium, Rest Assured, Docker, etc.) y la aplicación de herramientas de calidad de software.

Además, se ha seguido una metodología iterativa en la que cada versión publicada ha incorporado nuevas funcionalidades siguiendo un control de versiones semántico (Semantic Versioning), permitiendo trazar claramente la evolución del proyecto y los hitos técnicos superados.

Figura 1.6: Histórico de versiones etiquetadas en Github.

2

Objetivos

Este proyecto tiene como propósito principal no solo replicar la experiencia del juego Wordle, sino ampliarla, aplicar técnicas modernas de desarrollo software y validar cada fase del proceso mediante metodologías de calidad. Para ello, se han definido una serie de objetivos, divididos entre **principales**, orientados al desarrollo funcional y técnico esencial, y **secundarios**, centrados en automatización, entrega y documentación del producto.

2.1. Objetivos principales

- Desarrollar una aplicación web funcional y usable que permita jugar a Wordle desde el navegador.
- Ampliar la funcionalidad original del juego con un sistema de Leaderboard y Ranking por palabra acertada, permitiendo al usuario consultar partidas ganadas y compararse con otros jugadores.
- Diseñar una arquitectura modular, mantenible y escalable, utilizando Java 21, Spring Boot y MySQL como tecnologías base.
- Aplicar una metodología de desarrollo iterativo basada en TDD, guiando la implementación mediante pruebas automatizadas.
- Validar todas las funcionalidades implementadas a través de pruebas unitarias, de integración y end-to-end.

- Garantizar la calidad del código fuente mediante análisis estático con SonarCloud en cada iteración menor.

2.2. Objetivos secundarios

- Establecer un flujo de trabajo con GitHub Flow, utilizando integración continua (CI) con GitHub Actions para verificar automáticamente cada Pull Request (PR).
- Automatizar la entrega continua (CD) mediante la publicación de imágenes Docker versionadas en DockerHub tras cada merge a `master`.
- Gestionar el versionado del proyecto utilizando la convención SemVer, diferenciando versiones mayores, funcionalidades nuevas y correcciones técnicas.
- Documentar el proyecto de forma profesional, incluyendo capturas y trazabilidad completa del desarrollo y calidad aplicada.

2.3. Contribuciones del proyecto

El desarrollo de WordleApp busca demostrar cómo una aplicación sencilla desde el punto de vista funcional puede ser el vehículo ideal para aplicar prácticas avanzadas de calidad del software, simulando un entorno profesional de trabajo.

Este TFG también sirve como ejemplo de integración entre desarrollo, validación, automatización y control de versiones, con énfasis en la mejora continua.

3

Tecnologías, Herramientas y Metodologías

3.1. Tecnologías

3.1.1. Java 21

El lenguaje principal del proyecto ha sido **Java** en su versión 21 [1], que permite aprovechar características modernas como records, mejoras en las estructuras de control y un rendimiento optimizado en aplicaciones backend. Se ha utilizado como lenguaje base del sistema y como base para el motor de validación de palabras, control de partidas e integración con la base de datos.

3.1.2. Spring Boot 3

El framework elegido para el backend ha sido **Spring Boot** [2] por su enfoque modular, integración con Spring Data JPA y facilidad de configuración. La aplicación expone una API REST que gestiona el flujo del juego, la validación de palabras, la persistencia de partidas y el ranking de resultados.

3.1.3. MySQL

Para el almacenamiento de datos persistentes se ha utilizado **MySQL** [3]. En la base de datos se guarda la colección de palabras válidas para ingresar por el usuario, las palabras candidatas a ser la palabra a adivinar, las partidas ganadas y las estadísticas necesarias para construir los rankings.

3.1.4. HTML, JavaScript y Mustache

El frontend de WordleApp se ha desarrollado con **HTML** [4], **JavaScript** [5] y el motor de plantillas **Mustache** [6], incluido mediante Spring Boot. Mustache permite insertar dinámicamente el contenido HTML desde el backend con una separación clara entre lógica y presentación. Se ha usado para renderizar el tablero de juego, mostrar mensajes de error y resultados de pistas, controlando la estructura del DOM de forma sencilla pero efectiva.

3.2. Herramientas

3.2.1. Rest Assured

La herramienta **Rest Assured** [7] se ha empleado para validar el comportamiento de la API REST. Se han creado tests para los endpoint, verificando los distintos comportamientos de la aplicación, y asegurando que las respuestas sean correctas tanto en formato como en contenido.

3.2.2. JUnit y Selenium

Se ha utilizado **JUnit 5** [8] para las pruebas unitarias y de integración. En concreto, la extensión de **Mockito** [9] para las unitarias, y **Spring Test** [10] con **MockMvc** [11] para las de integración. Se ha usado **Selenium WebDriver** [12] junto con WebDriverManager para ejecutar pruebas E2E de la interfaz de usuario, validando la interacción desde el punto de vista del usuario.

3.2.3. SonarCloud

Se ha configurado **SonarCloud** [13] para analizar cada iteración del proyecto. En cada versión menor (0.1.0, 0.2.0, etc.) se ejecuta un análisis estático del código desde GitHub Actions. El análisis incluye cobertura de código (con **Jacoco**) [14],

complejidad ciclomática, deuda técnica y duplicación. Cuando se detectan violaciones o métricas inaceptables, se aplican correcciones en versiones patch (0.1.1, 0.2.3, etc.).

3.2.4. Git y GitHub

Todo el proyecto ha sido versionado mediante **Git** [15] y alojado en **GitHub** [16]. Se ha seguido un flujo de trabajo basado en GitHub Flow, trabajando en ramas independientes con pull requests, que se integraban en master tras superar los tests y los análisis de calidad.

3.2.5. Docker y DockerHub

La aplicación se ha dockerizado por completo, incluyendo un **DockerFile** [17] para el backend y un docker-compose.yml que lanza la base de datos y el backend como servicios independientes. Las imágenes se publican en **DockerHub** [18] con una etiqueta correspondiente a la versión semántica (1.5.2, etc.), permitiendo despliegue inmediato en otros entornos.

3.3. Metodologías

3.3.1. Iterativa e incremental

La metodología seguida ha sido iterativa e incremental, organizando el desarrollo por versiones semánticas **SemVer** [19] (v0.x.y, v1.x.y...), cada una con objetivos funcionales y técnicos concretos. Esto ha permitido entregar versiones funcionales progresivas, con validación continua del avance.

3.3.2. TDD (Test Driven Development)

A lo largo del desarrollo de *WordleApp*, se ha aplicado la metodología **Test Driven Development (TDD)** [20] en varias funcionalidades clave del sistema. Esta metodología consiste en escribir primero una prueba automatizada que falla, implementar el código mínimo necesario para que dicha prueba pase, y posteriormente refactorizar el código, para mejorar su calidad sin alterar el comportamiento.

Este enfoque ha sido especialmente útil en funcionalidades sensibles del backend como la lógica de validación de pistas, la gestión del número de intentos o la

verificación del ranking. Gracias a TDD se ha asegurado una mayor estabilidad, una cobertura de código más robusta, y se ha evitado la aparición de regresiones durante las sucesivas iteraciones del proyecto.

En la subsección **Aplicación práctica del ciclo TDD** 4.4.2, dentro del capítulo de **Pruebas**, se incluye un ejemplo detallado del ciclo TDD aplicado al desarrollo de la lógica de pistas del juego.

3.3.3. GitHub Flow y CI/CD

El proyecto ha seguido una metodología iterativa e incremental en espiral. En cada iteración se definieron funcionalidades concretas y se desarrollaron de forma aislada en ramas propias, siguiendo el flujo **GitHub Flow** [21]. Cada cambio fue validado mediante Integración Continua (CI) antes de ser integrado en la rama principal.

Además, el uso de **GitHub Actions** [22] como sistema de Integración y Entrega Continua (CI/CD) ha permitido mantener un flujo de trabajo automatizado, donde cada entrega ha sido testeada y verificada antes de su publicación, reforzando el enfoque de calidad adoptado desde el inicio del proyecto.

4

Descripción Informática

4.1. Requisitos

4.1.1. Requisitos Funcionales

Los requisitos funcionales definen el comportamiento esperado de la aplicación WordleApp desde el punto de vista de las acciones que el sistema debe permitir al usuario realizar. A continuación se organizan en subtipos para mejorar la trazabilidad y claridad.

Requisitos Funcionales de Entrada y Validación

- **RF01.** El sistema debe permitir al usuario introducir su nombre al iniciar la aplicación para identificarlo durante la sesión de juego.
- **RF02.** El sistema debe mostrar un mensaje de error si el usuario pulsa el botón "Start" sin haber introducido un nombre válido.
- **RF03.** El sistema debe mostrar el nombre del usuario en la esquina superior izquierda durante toda la partida.
- **RF06.** El sistema debe permitir al usuario introducir palabras utilizando el teclado físico o un teclado virtual en pantalla.

- **RF07.** El sistema debe validar una palabra únicamente cuando el usuario pulse el botón "Enter".
- **RF10.** El sistema debe mostrar un mensaje de error y una animación si la palabra introducida no pertenece al diccionario de palabras válidas.

Requisitos Funcionales Generales

- **RF04.** El sistema debe seleccionar aleatoriamente una palabra de cinco letras de un diccionario precargado para iniciar cada nueva partida.
- **RF05.** El sistema debe permitir al usuario realizar un máximo de seis intentos por partida para adivinar la palabra secreta.
- **RF08.** El sistema debe generar pistas visuales tras cada intento, indicando con colores: verde (letra correcta y bien ubicada), naranja (letra correcta pero mal ubicada) y gris (letra incorrecta).
- **RF09.** El sistema debe mostrar al usuario el número de intentos restantes después de enviar cada palabra.
- **RF11.** El sistema debe mostrar la palabra secreta en pantalla si el usuario agota los seis intentos sin adivinarla.
- **RF12.** El sistema debe mostrar un mensaje de victoria si el usuario adivina la palabra secreta correctamente en uno de los seis intentos.
- **RF13.** El sistema debe mostrar un botón de "Reset" tras finalizar una partida, independientemente de si el usuario ha ganado o perdido.
- **RF14.** El sistema debe reiniciar el juego y seleccionar una nueva palabra secreta al pulsar el botón "Reset", manteniendo el mismo nombre de usuario.
- **RF16.** El sistema debe permitir al usuario acceder a una pantalla "LeaderBoard" para consultar las partidas ganadas registradas en el sistema.
- **RF17.** El sistema debe permitir ordenar el leaderboard por número de intentos o por fecha.
- **RF18.** El sistema debe mostrar un botón "Back" en la pantalla "LeaderBoard" para regresar a la pantalla principal del juego.
- **RF19.** El sistema debe permitir al usuario acceder a una pantalla "Ranking" donde podrá consultar rankings por palabra adivinada.
- **RF20.** El sistema debe permitir al usuario introducir una palabra en un input de búsqueda para visualizar los mejores jugadores ordenados por número de intentos.

- **RF21.** El sistema debe mostrar un botón "Back" en la pantalla "Ranking" para regresar a la pantalla principal del juego.

Requisitos Funcionales de Persistencia

- **RF15.** El sistema debe guardar en base de datos todas las partidas ganadas junto al nombre del usuario, número de intentos, palabra adivinada y fecha.
- **RF22.** El sistema debe permitir almacenar múltiples partidas ganadas por el mismo usuario sobre una misma palabra.
- **RF23.** El sistema debe mostrar empates en el ranking cuando varios usuarios hayan adivinado una misma palabra con el mismo número de intentos.

Requisitos Funcionales de Interfaz

- **RF24.** El tablero del juego debe tener una estructura de 6 filas y 5 columnas que represente los intentos del jugador.
- **RF25.** El sistema debe destacar la casilla activa con un borde azul para mejorar la visibilidad durante la introducción de letras.
- **RF26.** El sistema debe actualizar los colores del teclado virtual conforme a las pistas visuales del intento actual.

4.1.2. Requisitos no funcionales

Los requisitos no funcionales establecen criterios sobre la calidad, el rendimiento, la seguridad y la usabilidad del sistema WordleApp. No describen funciones específicas, sino condiciones bajo las cuales debe operar.

- **RNF01.** El sistema debe ser accesible desde navegador web moderno sin requerir instalación adicional.
- **RNF02.** La interfaz debe ser intuitiva y permitir que un nuevo usuario comprenda el funcionamiento en menos de 5 minutos.
- **RNF03.** El sistema debe mostrar mensajes de error comprensibles, orientados a usuario final.
- **RNF04.** La interfaz debe ser responsive y adaptarse a ordenadores, tablets y móviles.

- **RNF05.** La aplicación debe responder a cualquier acción del usuario en menos de 2 segundos bajo condiciones normales.
- **RNF06.** El sistema debe estar dockerizado y preparado para su despliegue automatizado en distintos entornos.
- **RNF07.** La aplicación debe contar con pruebas automatizadas que cubran al menos un 80 % de las funcionalidades implementadas.
- **RNF08.** El código fuente debe superar los quality gates definidos por SonarCloud en cada entrega (mantenibilidad, fiabilidad, duplicación...).
- **RNF09.** El proyecto debe aplicar control de versiones semántico (SemVer) en cada release.
- **RNF10.** La imagen Docker de la aplicación debe ser publicada automáticamente en DockerHub tras cada merge en la rama principal.

4.2. Arquitectura y análisis

La aplicación WordleApp sigue una arquitectura monolítica de tipo cliente-servidor basada en una API REST, con el backend desarrollado en Spring Boot y el frontend generado dinámicamente mediante el motor de plantillas Mustache.

Estructura general

El servidor gestiona los intentos de los usuarios, el estado de la partida, el control de sesiones y la persistencia en una base de datos MySQL. El cliente se ejecuta en el navegador del usuario y permite interactuar con la aplicación a través de una interfaz HTML y JavaScript, enriquecida con CSS y animaciones para ofrecer una experiencia fluida.

La organización del código en paquetes sigue el principio de separación de responsabilidades:

- **Controller**: gestiona las peticiones HTTP y conecta la interfaz con el modelo.
- **Dto**: contiene estructuras de datos de transferencia como `LeaderboardEntry`.
- **Model**: agrupa las entidades persistentes como `Game`, `SecretWord`.
- **Repository**: interfaces que extienden `JpaRepository` para acceder a la base de datos.

- **Service**: encapsula la lógica del negocio del juego (gestión de intentos, ranking, palabras).
- **Templates y css**: interfaz web generada con Mustache y estilizada con archivos `.css`.

4.2.1. Arquitectura

La figura 4.1 muestra el diagrama de arquitectura general de la aplicación WordleApp. Este proyecto se ha organizado siguiendo un modelo en capas, donde cada componente cumple una función específica dentro del flujo de ejecución, respetando el principio de separación de responsabilidades. Las capas, de izquierda a derecha, representan el flujo de interacción: presentación, backend (que incluye los controladores y la lógica de negocio), y la base de datos.

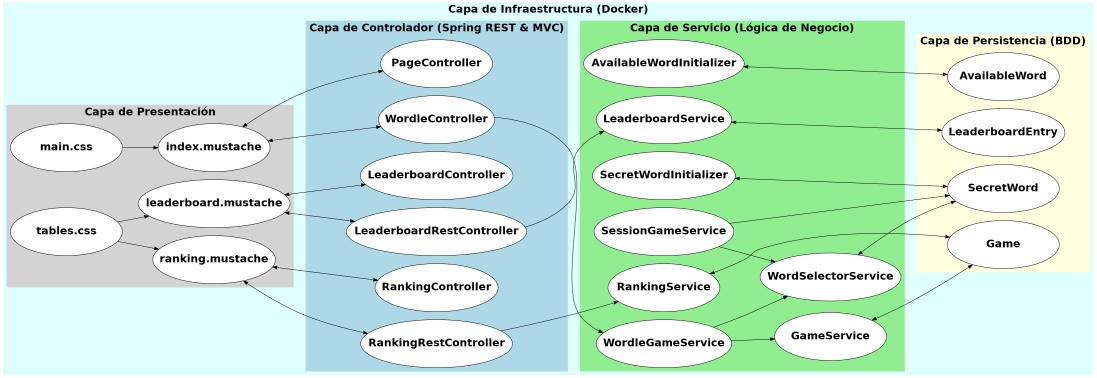

Figura 4.1: Diagrama de arquitectura de capas de WordleApp.

- **Capa de presentación (FRONTEND)**: compuesta por las vistas (`index.mustache`, `leaderboard.mustache` y `ranking.mustache`, los archivos de estilos `main.css` y `tables.css`, y el código JavaScript embebido que gestiona dinámicamente las interacciones del usuario con el backend mediante peticiones `fetch()` a los controladores REST. Esta capa es responsable de la experiencia de usuario y la presentación visual del juego, mostrando el tablero, los rankings y gestionando la interacción mediante teclado físico o virtual.
- **Capa de controladores (BACKEND)**: incluye controladores MVC (`PageController`, `LeaderboardController`, `RankingController`) orientados a devolver vistas renderizadas, y controladores REST (`WordleController`, `LeaderboardRestController`, `RankingRestController`) que exponen endpoints para el acceso a datos en formato JSON. Esta capa traduce las peticiones del frontend a acciones sobre la lógica de negocio o recuperación de datos.

- **Capa de servicios (BACKEND)**: concentra la lógica de negocio. El servicio principal es `WordleGameService`, que orquesta el flujo del juego, apoyándose en `GameService` y `WordSelectorService` para operaciones de partida y selección de palabras. `SessionGameService` gestiona el estado de la sesión HTTP y se comunica con `WordSelectorService` para establecer nuevas palabras objetivo. Los servicios `RankingService` y `LeaderboardService` gestionan la obtención y cálculo de clasificaciones y estadísticas. Además, los servicios auxiliares `AvailableWordInitializer` y `SecretWordInitializer` se encargan de cargar las palabras válidas y secretas desde ficheros al arrancar la aplicación.
- **Capa de persistencia (BDD)**: contiene las entidades JPA (`Game`, (`SecretWord`, `AvailableWord`, `LeaderboardEntry`) que se almacenan en la base de datos MySQL. Estas entidades son utilizadas por los servicios para persistir y recuperar datos de las partidas, palabras secretas y rankings.
- **Capa de infraestructura**: hace referencia al entorno de despliegue y ejecución de la aplicación. Se emplea `Docker` como tecnología de contenedorización, orquestando tanto la base de datos MySQL como la aplicación Java (Spring Boot). En caso de utilizar archivos como `Dockerfile` o `docker-compose.yml`, estos forman parte esencial de esta capa, garantizando portabilidad, consistencia del entorno y facilidad de despliegue en cualquier sistema compatible.

4.2.2. Comunicación y flujo de datos entre capas

El diseño modular y en capas de la aplicación WordleApp no sólo facilita el mantenimiento, la escalabilidad y el testing, sino que también asegura la trazabilidad de los componentes y de los flujos de datos.

Aunque el diseño sigue un enfoque por capas con responsabilidades bien delimitadas, la comunicación entre ellas no es estrictamente unidireccional. Existen flujos de información bidireccionales en varios puntos clave del sistema:

- **Capa de presentación ↔ Controladores REST:** La interfaz de usuario (frontend) se comunica con los controladores REST del backend mediante peticiones asíncronas `fetch()`, intercambiando datos en formato JSON. Esto permite que la vista no sólo envíe información al servidor (por ejemplo, intentos de palabras, solicitud de rankings), sino que también actualice dinámicamente su estado con las respuestas recibidas.
- **Servicios ↔ Persistencia:** La capa de servicios interactúa tanto para consultar información como para persistir nuevos datos en la base de datos. Este flujo bidireccional es esencial para reflejar en tiempo real los resultados de la partida, la gestión de usuarios y la actualización de rankings.

Nota sobre controladores y obtención de datos: En la arquitectura de WordleApp, los controladores MVC (por ejemplo, `RankingController` y `LeaderboardController`) se encargan exclusivamente de servir las vistas iniciales (`.mustache`), sin interactuar con servicios ni modelos de datos. La obtención y actualización de datos dinámicos se realiza en el frontend mediante código JavaScript, que utiliza peticiones `fetch()` a los controladores REST (`RankingRestController`, `LeaderboardRestController`, etc.).

Estos controladores REST, a su vez, sí delegan en los servicios de la aplicación para recuperar la información necesaria y devolverla en formato JSON. Así, se establece una separación clara entre la entrega de la interfaz (MVC) y la gestión de la lógica de negocio y datos (REST + servicios), lo que responde a patrones modernos de aplicaciones web SPA (Single Page Application).

Esta estructura de comunicación flexible responde a la naturaleza interactiva de las aplicaciones web modernas y refuerza la modularidad, mantenibilidad y extensibilidad del sistema.

4.2.3. Casos de uso de la aplicación

En la Figura 4.2 se representa el diagrama de casos de uso de la aplicación WordleApp. El único actor identificado es el **Usuario**, que puede interactuar con el sistema a través de distintas funcionalidades ofrecidas por la interfaz web.

Las principales acciones disponibles son:

- **Introducir nombre**: acción inicial que identifica al usuario y habilita el inicio de partida.
- **Jugar partida**: intenta adivinar la palabra oculta introduciendo palabras y recibiendo pistas visuales en cada intento.
- **Guardar partida**: si el usuario gana, la partida se guarda automáticamente junto con su nombre, palabra y número de intentos.
- **Resetear partida**: una vez finalizada una partida (ganada o perdida), puede reiniciar sin introducir de nuevo el nombre.
- **Ver leaderboard**: permite acceder a una pantalla con el ranking general de partidas ganadas, ordenable por fecha o por intentos.
- **Consultar ranking**: permite buscar los mejores jugadores por palabra adivinada y limitar los resultados por top N.

Este conjunto de casos de uso refleja todas las interacciones previstas entre el usuario y la aplicación, tanto durante el desarrollo de la partida como en las acciones complementarias de consulta y reinicio.

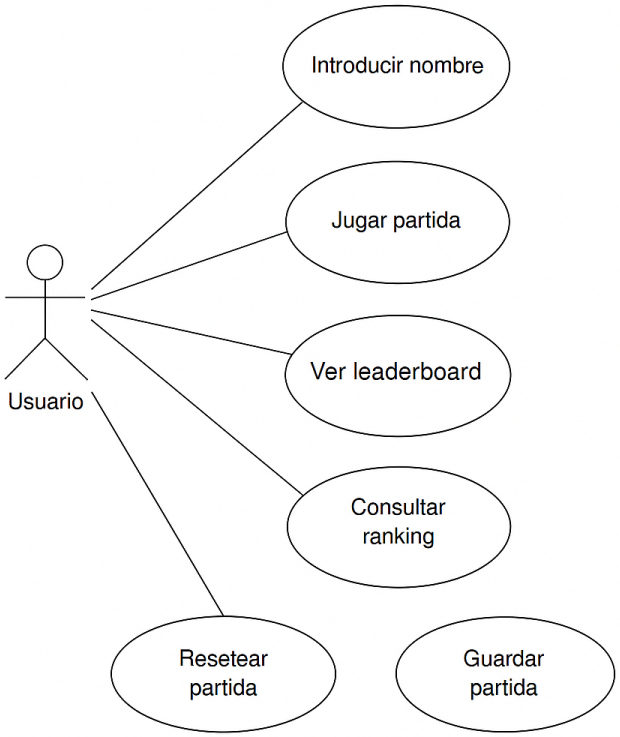

Figura 4.2: Diagrama de casos de uso de la aplicación WordleApp

4.3. Diseño e implementación

En este apartado se destacan algunos aspectos técnicos relevantes del desarrollo del sistema, tanto desde el punto de vista funcional como desde el punto de vista de calidad y diseño.

Diseño modular y arquitectura RESTful

Cada funcionalidad clave del juego se encapsuló en un servicio independiente. Por ejemplo:

- `WordleGameService`: lógica principal del juego (pistas, intentos, reinicio).
- `WordSelectorService`: selección y persistencia de la palabra secreta.
- `GameService` y `LeaderboardService`: persistencia de partidas y generación del ranking.

El diseño sigue el patrón clásico de controlador-servicio-repositorio, desacoplando la lógica del negocio del acceso a datos y de la gestión HTTP.

4.3.1. Implementaciones críticas

Algoritmo de generación de pistas

Uno de los aspectos más críticos fue la lógica que determina las pistas tras cada intento. Esta funcionalidad debía evaluar letra a letra, si cada letra coincide exactamente con la de la palabra secreta, si está contenida pero no en la misma posición, o si no esta contenida.

Esta lógica fue implementada en los métodos `markCorrectPositions()` y `markMisplacedLetters()` dentro del servicio `WordleGameService`.(Ver Código 4.1)

Código 4.1: Función implementa la lógica de pistas.

```
1  private void markCorrectPositions(String guess, char[] hint,
       boolean[] matched, boolean[] used) {
2          for (int i = 0; i < Constants.WORD_LENGTH; i++) {
3              if (guess.charAt(i) ==
                   wordSelectorService.getCurrentWord().charAt(i)) {
4                  hint[i] = guess.charAt(i);
5                  matched[i] = true;
6                  used[i] = true;
7              }
8          }
9      }
10 private void markMisplacedLetters(String guess, char[] hint,
       boolean[] matched, boolean[] used) {
11     for (int i = 0; i < Constants.WORD_LENGTH; i++) {
12         if (hint[i] == '_') {
13             for (int j = 0; j < Constants.WORD_LENGTH; j++) {
14                 if (!matched[j] && !used[j] && guess.charAt(i)
                       ==
                       wordSelectorService.getCurrentWord().charAt(j))
                       {
15                     hint[i] = '?';
16                     used[j] = true;
17                     break;
18                 }
19             }
20         }
21     }
22 }
```

Gestión de sesión entre partidas

Para mantener el estado del juego entre peticiones HTTP, se ha utilizado la interfaz `HttpSession` proporcionada por Spring. Esta decisión permite persistir datos específicos del usuario durante toda la duración de su sesión activa, sin

necesidad de implementar almacenamiento adicional.

Entre los datos almacenados en la sesión se encuentran:

- El número de intentos realizados (`attempts`).
- El estado de la partida (`gameWon`).
- La pista más reciente (`lastHint`).
- El nombre del usuario (`username`).

La clase `SessionGameService` encapsula esta lógica mediante el método `resetGame(String username, HttpSession session)`, que reinicia los valores de la sesión al comienzo de una nueva partida. Este método se invoca desde `WordleController` cuando se recibe una petición al endpoint `/api/wordle/reset`.

Desde el lado del cliente, el archivo `index.mustache` contiene lógica en JavaScript para controlar el almacenamiento temporal en `sessionStorage`, restaurar intentos previos y sincronizar el estado visual del juego tras un reinicio.

Código 4.2: Método resetGame en SessionGameService.

```
1 public void resetGame(String username, HttpSession session) {
2     session.removeAttribute("attempts");
3     session.removeAttribute("gameWon");
4     session.removeAttribute("lastHint");
5     wordSelectorService.selectRandomWord();
6     session.setAttribute("lastHint",
          wordSelectorService.getCurrentWord());
7     session.setAttribute("username", username);
8     }
9 }
```

Esta combinación entre estado de sesión del servidor y almacenamiento local en el navegador ofrece una solución eficaz para gestionar múltiples partidas sin necesidad de autenticación persistente.

Funcionalidades de Leaderboard y Ranking

El sistema cuenta con dos mecanismos de consulta de puntuaciones: el **leaderboard global** y el **ranking por palabra**.

- En la vista de `RankingService` 4.3 permite consultar los jugadores con mejor puntuación para una palabra secreta concreta, filtrando por top 1, top 3 o top 5.

- En la vista de `LeaderboardService` 4.4 se gestiona la recuperación de todas las partidas ganadas, y las ordena por fecha o número de intentos.

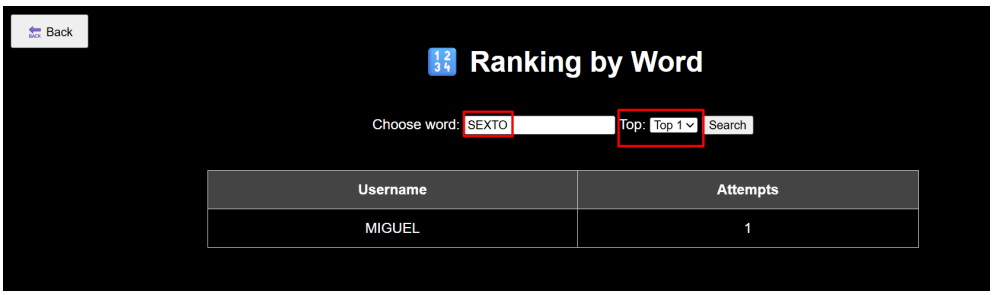

Figura 4.3: Vista del ranking por palabra: selección dinámica y filtrado.

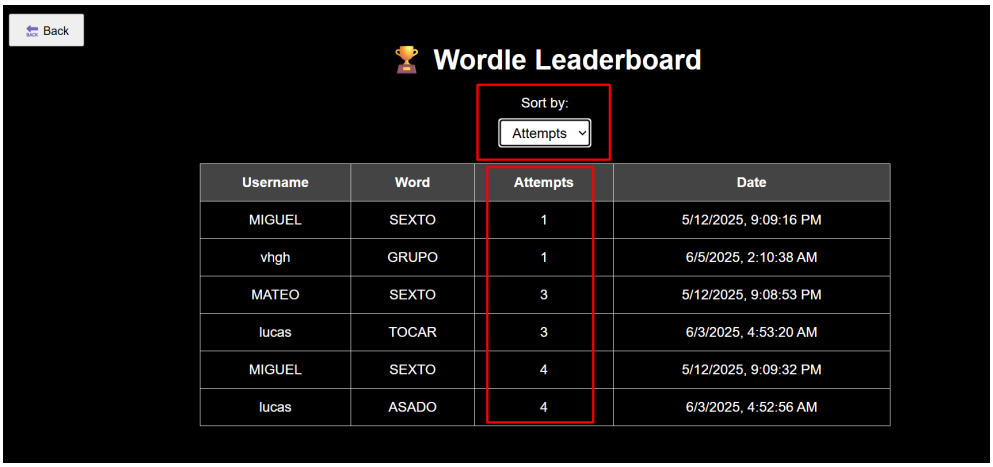

Figura 4.4: Vista del leaderboard con partidas ordenadas por intentos.

Estas funcionalidades han sido validadas con datos reales de prueba, verificando que los jugadores aparecen correctamente ordenados por sus resultados.

Carga de palabras desde fichero: diseño y mejora progresiva

Inicialmente, el sistema cargaba todas las palabras de cinco letras del fichero `dictionary.txt` directamente como candidatas a ser la palabra secreta. Este enfoque presentó un problema importante de usabilidad: muchas de las palabras eran inusuales, técnicas o poco intuitivas, lo que dificultaba la jugabilidad para un usuario básico.

Para resolverlo, se decidió dividir la lógica en dos repositorios:

- `AvailableWord`: contiene todas las palabras válidas de 5 letras en español (unas 10.000), cargadas desde `dictionary.txt` mediante la clase `AvailableWordInitializer`.
- `SecretWord`: contiene solo aquellas palabras que son razonables como candidatas a ser adivinadas (verbos, sustantivos, adjetivos comunes), cargadas desde `words.txt` mediante `SecretWordInitializer`.

De esta forma, cuando el jugador introduce una palabra:

- Si pertenece a `AvailableWord`, y NO es la palabra secreta, se valida como intento fallido.
- Si no pertenece a `AvailableWord`, se rechaza sin contar como intento, mostrando un mensaje de error.
- La palabra secreta siempre se selecciona de forma aleatoria desde `SecretWord`.

Esta decisión mejora significativamente la experiencia de usuario, manteniendo el desafío razonable.

4.3.2. Defectos y refactorizaciones

Error de experiencia de usuario

Durante el desarrollo se detectó un error en el frontend: al introducir una palabra no válida, se mostraba un mensaje de error, pero la fila del tablero no se limpiaba ni el cursor se reiniciaba. Esto se resolvió implementando la función `animateInvalidGuess()` en JavaScript.

Código 4.3: Reseteo visual tras palabra inválida.

```
function animateInvalidGuess() {
    for (let col = 0; col < 5; col++) {
        const tile =
            document.getElementById(`tile-${currentRow}-${col}`);
        if (tile) {
            tile.textContent = "";
            tile.classList.add("shake");

            // Remove the animation after finishing (so that it
                can be repeated if it fails again).
            tile.addEventListener("animationend", () => {
                tile.classList.remove("shake");
            }, { once: true });
        }
    }
        // Restart the visual and logical cursor, and clean row
            in memory
        guessRows[currentRow] = ["", "", "", "", ""];
        currentTile = 0;
        currentCol = 0;
        updateActiveCell(); // already makes the cursor visual
            set correctly
    }
```

Esta corrección mejoró la experiencia de usuario y aseguró el reinicio completo del intento tras una entrada inválida.

Refactorización del ordenamiento del leaderboard

Esta mejora no fue producto de un error funcional, sino del análisis reflexivo del diseño aplicado durante el desarrollo incremental. Tras completar las pruebas unitarias de `LeaderboardService`, se observó que el servicio asumía la responsabilidad de ordenar los resultados recuperados de la base de datos, a pesar de que esta lógica podía delegarse directamente al repositorio gracias a las capacidades de `Spring Data JPA`.

Código 4.4: Implementación inicial con ordenación manual en el servicio.

```
public List<LeaderboardEntry> getLeaderboard(String orderBy) {
    Comparator<Game> comparator =
        Comparator.comparing(Game::getCreatedAt).reversed();
    if ("attempts".equalsIgnoreCase(orderBy)) {
        comparator = Comparator.comparingInt(Game::getAttempts);
    }
    return gameRepository.findAll().stream()
            .sorted(comparator)
            .map(game -> new LeaderboardEntry(...))
            .toList();
}
```

Desde el punto de vista de la calidad del software, este patrón se identifica como una violación del principio de responsabilidad única (SRP). La lógica de ordenación no debería residir en el servicio si puede resolverse eficientemente desde la capa de persistencia. Además, mantener esta lógica en el servicio implicaba mayor acoplamiento, mayor dificultad de mantenimiento y menor claridad del código.

Por tanto, se tomó la decisión de refactorizar el código, delegando la ordenación a métodos derivados del repositorio como `findAllByOrderByAttemptsAsc()` o `findAllByOrderByCreatedAtDesc()`. Este tipo de decisiones ejemplifica el enfoque adoptado en el proyecto: priorizar la calidad y el diseño orientado a buenas prácticas, incluso en aspectos no funcionales del desarrollo.

Código 4.5: Implementación refactorizada delegando ordenación al repositorio.

```
public List<LeaderboardEntry> getLeaderboard(String orderBy) {
    List<Game> games = orderBy.equals("date")
        ? gameRepository.findAllByOrderByCreatedAtDesc()
        : gameRepository.findAllByOrderByAttemptsAsc();

    return games.stream()
             .map(game -> new LeaderboardEntry(...))
             .toList();
}
```

Métodos añadidos al repositorio:

```
List<Game> findAllByOrderByAttemptsAsc();
List<Game> findAllByOrderByCreatedAtDesc();
```

Esta refactorización también impacta directamente en las pruebas unitarias existentes. En concreto, el test `LeaderboardServiceTest`, que inicialmente validaba la lógica de ordenación como parte de la lógica de negocio del servicio, pierde su sentido original. Al delegar la ordenación al repositorio, el test deja de verificar una responsabilidad propia del servicio y pasa a convertirse en una validación indirecta de que el repositorio devuelve los datos ordenados correctamente, lo cual debería abordarse en pruebas de integración o de repositorio, no en una prueba unitaria del servicio.

4.4. Pruebas

Desde el inicio del proyecto, las pruebas han sido un pilar central en el desarrollo de WordleApp. Siguiendo una perspectiva de calidad de software, se han implementado pruebas automatizadas a diferentes niveles: pruebas unitarias, de integración y E2E (end-to-end). Esta estrategia ha permitido validar la funcionalidad de la aplicación desde múltiples ángulos, detectar errores de forma temprana,

y garantizar que la aplicación es robusta, mantenible y confiable.

Se ha aplicado TDD en varias fases del desarrollo, guiando la implementación a partir de pruebas previas, y se ha integrado un sistema de Integración Continua (CI) con GitHub Actions, que ejecuta automáticamente los tests y el análisis de calidad en cada pull request.

Además, se ha realizado análisis estático de código en cada versión menor del proyecto usando SonarCloud, corrigiendo la deuda técnica en versiones patch (ej. 0.3.1, 1.1.2, etc.), y registrando su evolución mediante capturas que acompañan esta memoria.

Con el fin de evitar duplicación de anotaciones de configuración (@ActiveProfiles, @Sql, etc.) entre los distintos tipos de pruebas, se ha definido una clase base llamada **BaseTestConfiguration**. Esta clase se encuentra en un paquete común y es utilizada por los tests funcionales, de UI, de API REST y de integración, garantizando la carga del entorno de prueba correcto y del diccionario de palabras antes de cada test.

De este modo, se asegura que los datos empleados durante la ejecución de las pruebas sean siempre independientes de los datos reales de la aplicación, reduciendo el riesgo de efectos no deseados y facilitando la reproducibilidad y aislamiento de los resultados de testing.

Código 4.6: Clase Java abstracta de la que se nutren algunos tests.

```
1 @ActiveProfiles("test")
2 @Sql(scripts = "/sql/available-words.sql", executionPhase =
      Sql.ExecutionPhase.BEFORE_TEST_METHOD)
3 public abstract class BaseTestConfiguration {
4 }
```

Se han implementado tres niveles de pruebas:

4.4.1. Pruebas unitarias (JUnit)

Aunque estos tests no acceden a una base de datos real, cumplen una función crítica: verifican cómo se comportan las clases de servicio ante diferentes respuestas del repositorio, que es precisamente lo que puede fallar en producción. Simular estos escenarios con Mockito permite garantizar que la lógica de negocio es robusta frente a entradas inesperadas o casos límite (por ejemplo, que no se seleccionen palabras, o que ya existan todas).

En este TFG se prioriza la separación de responsabilidades, y los tests unitarios reflejan esa arquitectura desacoplada. Además, al ser rápidos y auto-contenidos, permiten una ejecución eficiente en pipelines de integración continua, como GitHub Actions.

- Comportamiento predecible: Validan salidas esperadas para entradas dadas, por ejemplo, lanzar una excepción si no hay palabras disponibles en la BBDD.
- Detección temprana de errores: Permiten identificar errores lógicos desde fases iniciales del desarrollo (principio fail-fast).
- Rapidez de ejecución: No requieren levantar el contexto de Spring ni realizar operaciones de entrada/salida.
- Documentación del código: Comunican de forma clara la intención de la lógica y cómo se gestionan los casos límite.
- Resistencia al cambio: Aseguran que las refactorizaciones no alteren el comportamiento esperado de cada servicio.

Algunas de las clases probadas:

- **GameServiceTest**: guarda partidas ganadas y valida entidades.
- **WordSelectorServiceTest**: controla la palabra actual.
- **AvailableWordInitializerTest**: carga las palabras válidas al iniciar.
- **LeaderboardServiceTest**: gestiona el ranking global.
- **SecretWordInitializerTest**: inserta palabras objetivo en la base de datos.

Código 4.7: Test unitario para validar palabra devuelta por repositorio.

```
class WordSelectorServiceTest {
    private SecretWordRepository secretWordRepository;
    private WordSelectorService wordSelectorService;

    @BeforeEach
    void setUp() {
        secretWordRepository = mock(SecretWordRepository.class);
        wordSelectorService = new
           WordSelectorService(secretWordRepository);
    }
@Test
void shouldSetCurrentWordWhenRandomWordExists() {
    // given
    SecretWord mockWord = new SecretWord("GRAPE");
    when(secretWordRepository.findRandomWord()
    .thenReturn(Optional.of((mockWord)));
    // when
    wordSelectorService.selectRandomWord();
    // then
    assertEquals("GRAPE", wordSelectorService.getCurrentWord());
}
```

Código 4.8: Test unitario para validar error al no tener palabras en repositorio.

```
1  @Test
2  void shouldThrowExceptionWhenRandomWordIsNull() {
3      when(secretWordRepository.findRandomWord()).
4      thenReturn(Optional.empty());
5
6      IllegalStateException exception =
           assertThrows(IllegalStateException.class, () ->
7              wordSelectorService.selectRandomWord());
8
9      assertEquals("Random word not found.",
           exception.getMessage());
10 }
```

La clase `WordSelectorServiceTest` contiene pruebas unitarias que validan el comportamiento del servicio responsable de seleccionar la palabra secreta aleatoria al inicio de cada partida. Esta funcionalidad es esencial para garantizar el correcto funcionamiento del juego desde su primer paso y para evitar errores en la experiencia del usuario.

- **Caso exitoso** (Ver Código 4.7): se comprueba que, si el repositorio devuelve una palabra válida (por ejemplo, `GRAPE`), el servicio la asigna correctamente como palabra secreta actual mediante el método `getCurrentWord()`.
- **Caso de error controlado** (Ver Código 4.8): se valida que, si no hay palabras disponibles en la base de datos (respuesta vacía del repositorio), se lanza una excepción controlada de tipo `IllegalStateException` con el mensaje esperado.

Estas pruebas, diseñadas bajo el principio de aislamiento (utilizando mocks para las dependencias), permiten detectar errores de inicialización de forma temprana y asegurar un comportamiento robusto del sistema incluso en situaciones límite, como una base de datos vacía. Además, contribuyen a alcanzar los objetivos de calidad del TFG, ya que:

- Facilitan el desarrollo iterativo y la aplicación de TDD (desarrollo dirigido por pruebas).
- Permiten una rápida detección de regresiones cuando se refactoriza o amplía la lógica del servicio.
- Mejoran la mantenibilidad, ya que los tests documentan explícitamente los comportamientos esperados y los casos límite.
- Incrementan la confianza en la fiabilidad del código, al validar que la lógica principal responde correctamente ante cualquier escenario.

En definitiva, el uso sistemático de pruebas unitarias es un pilar fundamental para el desarrollo de código de calidad y para garantizar la estabilidad del software en el tiempo.

4.4.2. Aplicación práctica del ciclo TDD

Una de las metodologías clave seguidas en este proyecto ha sido el **Desarrollo Guiado por Pruebas** (TDD). Esta técnica consiste en crear primero los tests que describen el comportamiento esperado de una funcionalidad antes de implementarla, asegurando así que el desarrollo esté completamente orientado a los requisitos y reduzca las posibles regresiones.

En este caso, uno de los comportamientos centrales del juego es la generación de pistas después de cada intento del jugador. Este requisito se puede formular como: "El sistema debe proporcionar pistas visuales indicando las letras correctas y sus posiciones tras cada intento del usuario". A continuación se muestra un ejemplo real de cómo se aplicó el ciclo TDD para implementar esta funcionalidad, garantizando la trazabilidad directa entre requisito funcional, test y código.

Fase 1: Test fallido (rojo)

Se redactó el siguiente test de integración utilizando Rest Assured para validar el comportamiento esperado cuando el usuario introduce una palabra parcialmente correcta:

Código 4.9: Test inicial TDD para lógica de pistas.

```
1  private final String secretTestWord = "sexto".toUpperCase();
2  @BeforeEach
3  void setUp() {
4      RestAssured.port = port;
5      RestAssured.baseURI = "http://localhost:" + port +
           "/api/wordle";
6      Mockito.when(wordSelectorService.getCurrentWord())
7          .thenReturn(secretTestWord);
8      sessionFilter = new SessionFilter();
9  }
10 @Test
11 void testMixedCorrectAndIncorrectPositions() {
12     given()
13         .contentType("application/json")
14         .when()
15         .post("/guess?guess=SESGO")
16         .then()
17         .statusCode(HttpStatus.OK.value())
18         .body(containsString("S E _ _ O"));
19 }
```

Este test esperaba que la respuesta de la API incluyera la pista `S E _ _ O`, indicando que las letras `S`, `E` y `O` estaban en la posición correcta. Dado que en ese momento no existía aún la lógica que devolvía pistas, el test falló como se esperaba.

Fase 2: Implementación mínima (verde)

Se desarrolló una implementación básica que comparaba las letras del intento con la palabra secreta, posición por posición, y generaba una pista con letras correctas o espacios vacíos. El test pasó con esta primera versión funcional.

Código 4.10: Implementación inicial de pistas.

```
for (int i = 0; i < attempt.length(); i++) {
    if (attempt.charAt(i) == secret.charAt(i)) {
        clue[i] = attempt.charAt(i);
    } else {
        clue[i] = "_";
    }
}
```

Fase 3: Refactorización y casos adicionales

Una vez superado el test inicial, se añadieron nuevas pruebas que cubrían escenarios más complejos: letras presentes en posiciones incorrectas, letras repetidas, intentos inválidos, y persistencia de sesión. Este proceso ayudó a identificar posibles *casos límite* y ambigüedades en los requisitos funcionales, permitiendo clarificar y robustecer la lógica de negocio. Se refactorizó la lógica del servicio para mantener coherencia en todos los casos y mejorar la expresividad del código.

En este fragmento 4.1 se visualiza la refactorización del desarrollo de la lógica de la generación de pistas a partir de la palabra que introduce el usuario.

Valor añadido del enfoque TDD

Este ciclo permitió:

- Validar de forma temprana el comportamiento deseado del sistema, con trazabilidad directa entre requisitos, tests y código.
- Asegurar regresión cero al modificar la lógica de pistas, sirviendo los tests como red de seguridad.
- Servir de documentación viva para entender cómo debe responder la API REST.

- Clarificar requisitos y descubrir casos límite o ambigüedades, ya que escribir los tests fuerza a pensar en todos los posibles escenarios que el usuario puede provocar.
- Mejorar la colaboración en entornos de trabajo en equipo, ya que los tests documentan los contratos de comportamiento y permiten que cualquier desarrollador comprenda rápidamente las expectativas del sistema.

Aunque el test es de integración, representa fielmente el enfoque TDD aplicado a nivel funcional, especialmente útil en el desarrollo guiado por comportamiento y pruebas orientadas a REST. Esta metodología ha facilitado un desarrollo más controlado, predecible y alineado con los principios de calidad software.

4.4.3. Pruebas de integración

Las pruebas de integración validan el comportamiento del sistema cuando interactúan múltiples componentes entre sí, permitiendo detectar errores que sólo se manifiestan en escenarios reales de uso. En este proyecto, Spring Boot junto con la base de datos en memoria H2 permite ejecutar los repositorios JPA y simular una ejecución real de SQL, mientras que los controladores REST se verifican mediante peticiones HTTP auténticas con Rest Assured y MockMvc.

MockMvc

MockMvc posibilita realizar pruebas sobre los controladores de Spring Boot simulando peticiones HTTP sin necesidad de levantar el servidor real. Esto permite validar el comportamiento de endpoints individuales, sus códigos de respuesta y el contenido generado por el controlador, así como comprobar la serialización y el formato del JSON devuelto.

- **WordleControllerTest**: prueba los endpoints `/api/wordle/guess` y `/api/wordle/reset`, validando la lógica de envío de palabras, reinicio de partida y control de intentos.
- **GlobalExceptionHandlerTest**: valida el tratamiento de errores personalizados, asegurando que se responda con códigos y mensajes adecuados ante excepciones conocidas.
- **PageControllerTest**: comprueba que las vistas HTML se entregan correctamente mediante Mustache, asegurando la renderización del frontend.
- **LeaderboardEndpointTest** (Ver Código 4.11): verifica que el endpoint REST `/api/wordle/leaderboard` responde correctamente con una lista

ordenada por número de intentos y que el formato del JSON devuelto es coherente con el modelo esperado.

Código 4.11: Test del endpoint REST de ranking ordenado por intentos.

```
@Test
@DisplayName("Should return a valid leaderboard ordered by attempts")
void shouldReturnLeaderboardByAttempts() throws Exception {
    List<LeaderboardEntry> mockList = List.of(
            new LeaderboardEntry("Ana", "CASAS", 2, LocalDateTime.of(2025, 4, 24, 15, 30)),
            new LeaderboardEntry("Luis", "RATON", 4, LocalDateTime.of(2025, 4, 24, 12, 0))
    );
    when(leaderboardService.getLeaderboard("attempts"))
    .thenReturn(mockList);

    mockMvc.perform(get("/api/wordle/leaderboard?orderBy=attempts"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.length()").value(2))
            .andExpect(jsonPath("$[0].username").value("Ana"))
            .andExpect(jsonPath("$[0].word").value("CASAS"))
            .andExpect(jsonPath("$[0].attempts").value(2))
            .andExpect(jsonPath("$[0].date").exists())
            .andExpect(jsonPath("$[1].username").value("Luis"))
            .andExpect(jsonPath("$[1].word").value("RATON"))
            .andExpect(jsonPath("$[1].attempts").value(4));
    }
```

El test muestra cómo se comprueba que el endpoint REST `/api/wordle/leaderboard` responde correctamente, devolviendo una lista ordenada y todos los campos relevantes. En este test, se utiliza MockMvc para simular peticiones HTTP y Mockito para inyectar un resultado controlado en el servicio, validando tanto la lógica de negocio como el formato de la respuesta.

Rest Assured

Rest Assured se ha utilizado para realizar pruebas más completas sobre los endpoints REST, validando peticiones y respuestas reales contra el servidor embebido de Spring. Estas pruebas actúan sobre el sistema completo y son especialmente útiles para comprobar los flujos del juego Wordle de principio a fin.

- **WordleAttemptsRestTest** (Ver Código 4.12): valida que los intentos enviados por los usuarios se contabilicen correctamente, que se limite a un máximo de 6 intentos y que se devuelva el estado de la partida.

- **WordleCluesTest** (Ver Código 4.13): comprueba que la lógica de generación de pistas (letras correctas, mal colocadas e incorrectas) funciona según las reglas del juego Wordle.
- **WordleIntegrationTest**: integra el flujo completo del juego, desde la selección de palabra secreta hasta el fin de partida, actuando como una validación funcional del sistema.

Código 4.12: Test que valida el número de intentos para adivinar la palabra con Rest Assured.

```
1  @Test
2  void shouldAllowUpToFiveIncorrectAttempts() {
3      for (int i = 1; i <= 5; i++) {
4          given()
5          .contentType("application/json")
6          .filter(sessionFilter)
7          .when()
8          .post("/guess?guess=WRONG")
9          .then()
10         .statusCode(HttpStatus.OK.value())
11         .body(containsString("Try again! Attempts left: " + (6 -
               i)));
12     }
13     given()
14         .contentType("application/json")
15         .filter(sessionFilter)
16         .when()
17         .post("/guess?guess=WRONG")
18         .then()
19         .statusCode(HttpStatus.OK.value())
20         .body(containsString("GAME OVER! The secret word was: "
               + wordSelectorService.getCurrentWord()));
21 }
```

Código 4.13: Test que valida la lógica de la comprobación de palabra.

```
void testAllIncorrectLetters() {
    given()
            .contentType("application/json")
            .when()
            .post("/guess?guess=CLIMA")
            .then()
            .statusCode(HttpStatus.OK.value())
            .body(containsString("_ _ _ _ _"));
}
void testCorrectLetterPositions() {
    given()
            .contentType("application/json")
            .when()
            .post("/guess?guess=SEXTO")
            .then()
            .statusCode(HttpStatus.OK.value())
            .body(containsString("S E X T O"));
}
void testFirstWordPositions() {
    given()
            .contentType("application/json").when()
            .post("/guess?guess=EUROS")
            .then()
            .statusCode(HttpStatus.OK.value())
            .body(containsString("? _ _ ? ?"));
}
```

Pruebas de integración de servicios y repositorios

Además de los tests sobre endpoints REST, se han implementado pruebas de integración directa sobre los servicios de negocio, empleando Spring Boot Test para levantar el contexto de la aplicación y trabajar con la base de datos embebida en un entorno controlado. Un ejemplo representativo es:

- **RankingServiceTest**: valida la integración entre el servicio de ranking y la base de datos, asegurando que el cálculo de clasificaciones, ordenaciones y empates se realiza correctamente sobre datos reales, sin necesidad de exponer endpoints web ni simular peticiones HTTP.

Valor añadido de las pruebas de integración en este contexto:

Estas pruebas resultan clave desde la perspectiva de la calidad de software porque:

- Simulan escenarios de usuario reales, permitiendo validar no sólo la lógica aislada sino el funcionamiento conjunto del sistema.

- Detectan errores en la integración entre capas (controlador, servicio, repositorio, persistencia), incluyendo errores de mapeo, serialización, gestión de sesión, validación de entradas y formato de las respuestas.
- Actúan como "contrato" que debe cumplir el sistema, ayudando a documentar la API y a garantizar la compatibilidad ante futuros cambios.
- Permiten detectar regresiones si se modifica el comportamiento de los servicios asociados, especialmente en funcionalidades críticas como el ranking o la lógica de intentos.
- Refuerzan la robustez y mantenibilidad del software, facilitando un ciclo de vida más controlado y seguro para la evolución de la aplicación.

Además, en el caso concreto de WordleApp, la experiencia desarrollando estas pruebas ha permitido descubrir y resolver incidencias reales, como problemas en la validación de palabras, gestión de sesiones y consistencia de las respuestas REST, lo que demuestra el valor tangible de integrar las pruebas de integración en el ciclo de desarrollo.

4.4.4. Pruebas End-to-End

Las pruebas E2E (End-to-End) validan el comportamiento del sistema desde la perspectiva del usuario final. En *WordleApp*, estas pruebas automatizan la interacción con la interfaz gráfica a través de un navegador, comprobando que los flujos completos del juego (inicio, intento, victoria, derrota, reinicio, etc.) se comportan de forma correcta y coherente.

Para ello, se ha utilizado **Selenium WebDriver**, en modo *headless*, lo que permite ejecutar pruebas sin mostrar el navegador gráfico.

Estructura Page Object

El diseño de las pruebas E2E sigue el patrón **Page Object Model (POM)**, una buena práctica que encapsula la lógica de interacción con la interfaz en una clase separada. En este caso, la clase `WordlePage` define todos los métodos reutilizables para interactuar con la aplicación web, como enviar palabras, leer mensajes de estado, validar el color de las casillas, o pulsar el botón de reinicio de partida. Esto permite que las pruebas sean más limpias, reutilizables y mantenibles.

La clase `WordleUITest` contiene las pruebas funcionales completas, y utiliza internamente la clase `WordlePage` para interactuar con la interfaz, simulando la experiencia real de un usuario.

Código 4.14: Funciones pertenecientes al Page Object.

```
1  public String getResultMessage() {
2      wait.until(ExpectedConditions.presenceOfElementLocated
3      (resultMessage));
4      WebElement resultElement =
           wait.until(ExpectedConditions.visibilityOfElementLocated
5      (resultMessage));
6      return resultElement.getText();
7  }
8  public void makeGuessUI(String guess) {
9      WebElement body =
           wait.until(ExpectedConditions.visibilityOfElementLocated
10     (By.tagName("body")));
11     body.sendKeys(guess + Keys.ENTER);
12     }
13 public boolean isGameOver() {
14     try {
15         if (driver.findElements(resultMessage).isEmpty()) {
16             return false; // If the result has not yet appeared,
                   the game is not over.
17         }
18         String resultText = getResultMessage();
19         return resultText.contains("GAME OVER!") ||
               resultText.contains("CORRECT!");
20     } catch (Exception e) {
21         return false; // If there are any errors, we assume that
               the game is still in progress.
22     }
23 }
```

Pruebas realizadas

Entre las pruebas más destacadas realizadas con Selenium:

- Validación de una partida ganada en menos de 6 intentos, comprobando los colores verdes en todas las casillas de una fila.
- Comprobación del mensaje de `GAME OVER!` tras agotar los 6 intentos, e imposibilidad de continuar.
- Verificación del funcionamiento del botón `Reset`, asegurando que todas las casillas se vacían y el estado del juego se reinicia.
- Comprobación de que introducir una palabra inválida muestra un mensaje de error y borra la palabra insertada en la fila automáticamente.

Fragmento representativo de prueba

Código 4.15: Validación de intento correcto en Wordle con todas las letras acertadas.

```
1 wordlePage.makeGuessUI("NOBLE");
2 String result = wordlePage.getResultMessage();
3 List<String> rowClasses = wordlePage.getTileClasses(2);
4 List<String> expectedColors = List.of("green", "green", "green",
      "green", "green");
5
6 for (int i = 0; i < 5; i++) {
7     assertTrue(rowClasses.get(i).contains(expectedColors.get(i)));
8 }
9 assertTrue(result.contains("CORRECT"));
```

Este fragmento corresponde a un test de victoria en el tercer intento, donde se comprueba tanto la respuesta visual (color verde en todas las letras) como el mensaje informativo final.

Importancia desde la perspectiva de calidad

Estas pruebas son fundamentales para garantizar que la experiencia de usuario no se vea afectada por errores introducidos en futuras versiones. Además, su ejecución automática dentro del pipeline de CI refuerza el enfoque de calidad continua del proyecto, permitiendo detectar fallos visuales o funcionales antes de que lleguen a producción.

4.4.5. Análisis estático de código

En el desarrollo de *WordleApp*, el análisis estático de código ha sido un componente esencial para garantizar la calidad técnica y mantenibilidad del proyecto. Se ha realizado mediante el uso combinado de **SonarCloud** (en la nube, como paso del pipeline de CI/CD) y **SonarLint** (a nivel local, integrado en el entorno de desarrollo).

SonarCloud: análisis continuo en cada versión

Se ha configurado SonarCloud como paso obligatorio en el sistema de integración continua mediante GitHub Actions. Cada vez que se abre un *pull request* o se publica una nueva versión menor, el workflow ejecuta el comando `mvn verify sonar:sonar`, generando un análisis completo del código fuente. Entre los aspectos analizados se encuentran:

- Cobertura de pruebas (medida con Jacoco)

- Complejidad ciclomática.
- Deuda técnica (tiempo estimado de corrección)
- Bugs, code smells y duplicación de código.
- Cumplimiento de quality gates.

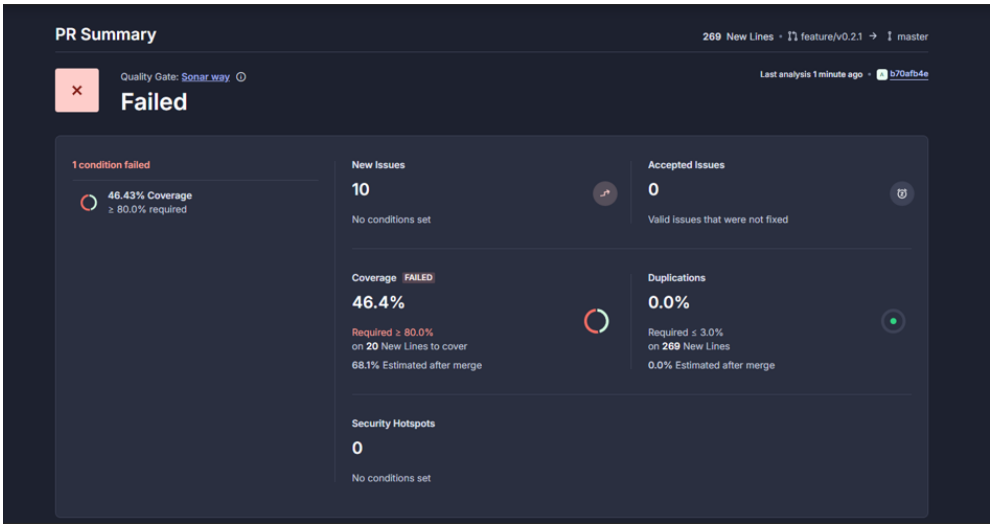

Figura 4.5: Análisis fallido en SonarCloud que originó una versión patch.

Cuando alguna versión menor (por ejemplo, `v1.2.0`) no superaba los **quality gates** (Ver Figura 4.6), se abría una nueva versión *patch* (por ejemplo, `v1.2.1`) dedicada exclusivamente a corregir los errores encontrados. En estos casos se revisaban los métodos con mayor complejidad, se extraían funciones auxiliares y se eliminaban duplicidades o estructuras no seguras.

Ejemplo de corrección en versión patch

En la versión `1.2.0`, SonarCloud detectó una deuda técnica crítica en la clase `AvailableWordInitializer`, donde se mezclaban responsabilidades de lectura de fichero, validación y carga a base de datos. Esta deuda se resolvió en la versión `1.2.1` separando las responsabilidades en los métodos `CreateReaderFromDcitionary()` e `insertValidWords()`, y aplicando el principio de responsabilidad única.

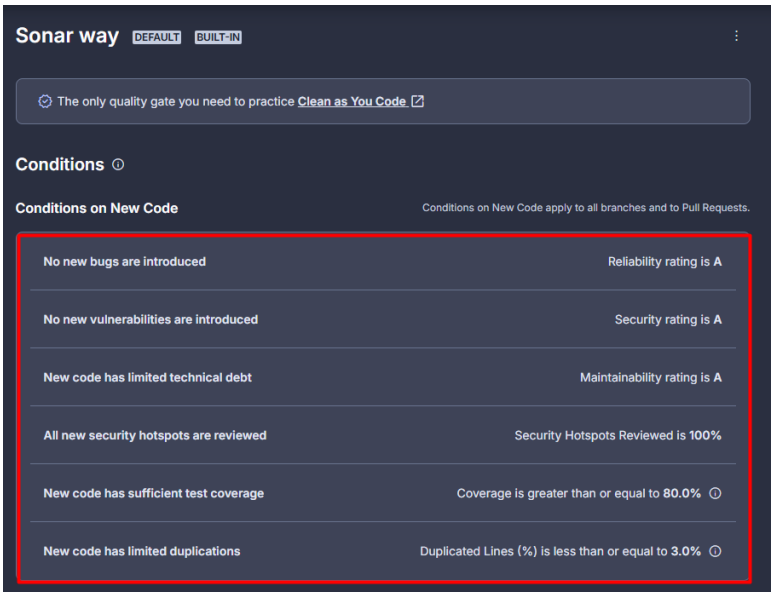

Figura 4.6: Quality Gates de SonarCloud.

```
public BufferedReader createReaderFromDictionary() {  2 usages  miguel.serralta
    InputStreamReader inputStreamReader = new InputStreamReader(
            Objects.requireNonNull(getClass().getClassLoader().getResourceAsStream( name: "dictionary.txt"))
    );
    return new BufferedReader(inputStreamReader);
}

private int insertValidWords(BufferedReader reader) {  1 usage  miguel.serralta
    return reader.lines()  Stream<String>
            .map(String::trim)
            .map(String::toUpperCase)
            .filter(word -> word.length() == 5)
            .distinct()
            .filter(word -> !availableWordRepository.existsByWord(word))
            .map(word -> {
                try {
                    availableWordRepository.save(new AvailableWord(word));
                    return 1;
                } catch (Exception e) {
                    log.warn("⚠️ Could not insert '{}': {}", word, e.getMessage());
                    return 0;
                }
            })  Stream<Integer>
            .mapToInt(Integer::intValue)  IntStream
            .sum();
}
```

Figura 4.7: Código refactorizado a raíz de análisis SonarCloud.

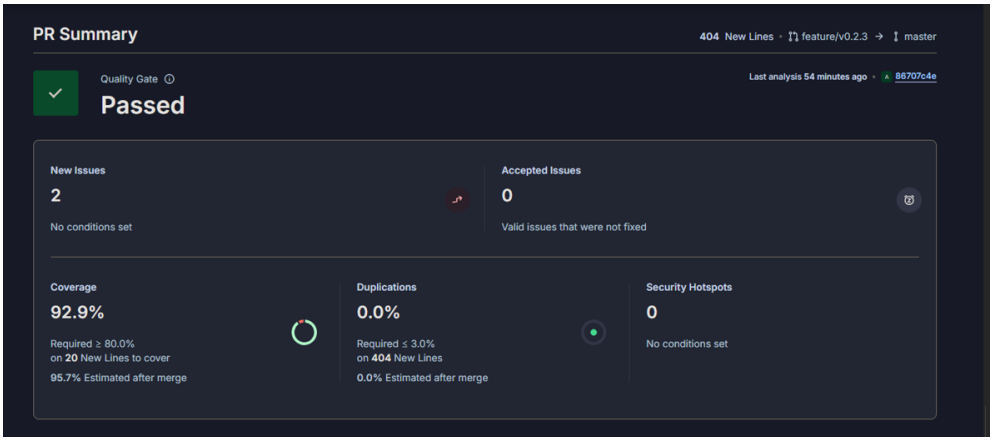

Figura 4.8: Análisis superado tras aplicar correcciones en la versión patch.

Esta práctica de lanzar versiones específicas para solucionar errores técnicos refuerza el enfoque de calidad incremental y la trazabilidad del estado del proyecto.

Impacto en la calidad del proyecto

La integración continua ha tenido un papel clave para:

- Garantizar que ninguna funcionalidad comprometa la estabilidad del sistema.
- Detectar y corregir errores de manera temprana.
- Mantener bajo control la evolución de la calidad del código.
- Automatizar la validación técnica de cada entrega.

Este enfoque ha sido esencial para aplicar de manera efectiva las metodologías de calidad software propuestas en este TFG, combinando buenas prácticas de desarrollo con herramientas reales del sector.

SonarLint: control local antes del análisis en remoto

Antes de realizar el commit, el entorno de desarrollo (IntelliJ IDEA) contaba con la extensión `SonarLint` conectada a SonarCloud. Esta herramienta permitía detectar errores de estilo, duplicación o potenciales bugs incluso antes de lanzar el análisis remoto con la herramienta SonarCloud.

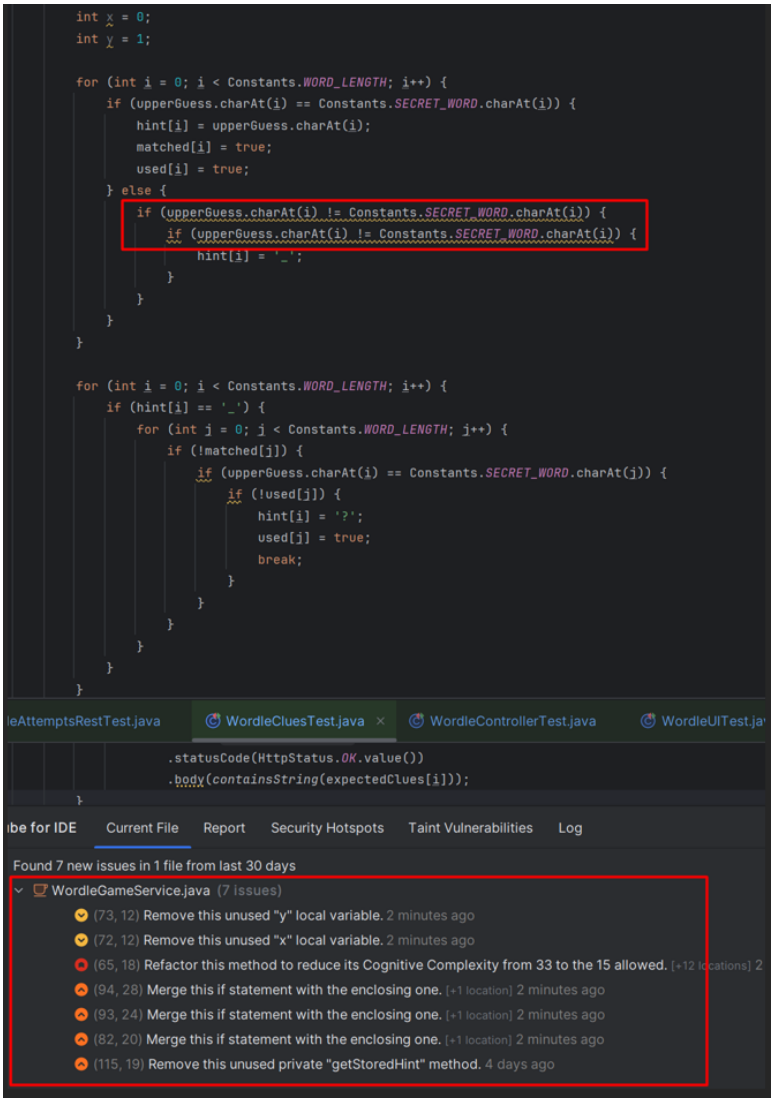

Figura 4.9: Detección local de problemas mediante SonarLint antes del commit.

Gracias a esta integración, se reducían considerablemente los errores detectados en el pipeline, y se favorecía una programación preventiva basada en estándares. (Ver Figura 4.9)

4.4.6. Integración continua de las pruebas y control de regresiones

Desde el inicio del proyecto, se ha utilizado un sistema de **Integración Continua (CI)** basado en **GitHub Actions**, con el objetivo de asegurar que cada nueva funcionalidad o corrección introducida en el proyecto `WordleApp` cumple con los estándares de calidad definidos. Esta práctica ha permitido reducir el

riesgo de regresiones y mantener un control constante sobre el estado funcional y técnico del software.

Flujo GitHub Flow y CI como base de control de calidad

El desarrollo ha seguido el flujo *GitHub Flow*: cada nueva funcionalidad o mejora se ha implementado en una rama independiente creada desde `master`, y al finalizar, se ha abierto un *Pull Request*. Gracias a GitHub Actions, cada PR activa automáticamente la ejecución de:

- Compilación y construcción del proyecto.
- Ejecución de todas las pruebas automatizadas (unitarias, de integración, E2E).
- Análisis estático de código con **SonarCloud**.

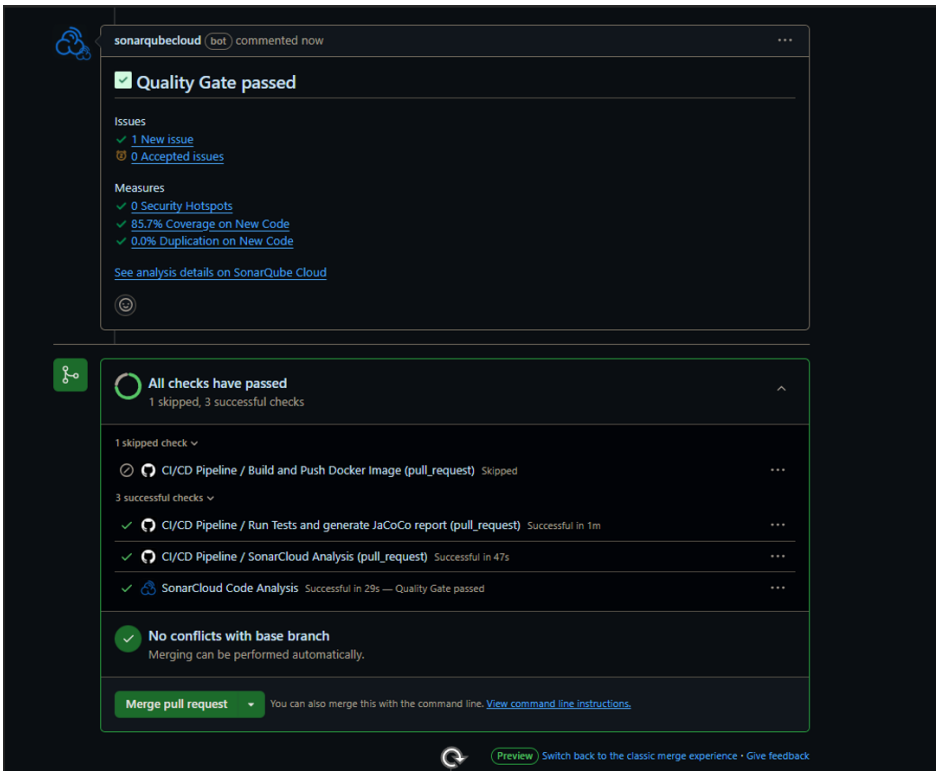

Figura 4.10: Ejecución automática de pruebas y análisis estático en GitHub Actions al crear un Pull Request.

El **merge** solo es posible si todas las pruebas y análisis se completan correctamente, lo que garantiza que no se introduzcan errores ni se degrade la calidad del código. Esta estrategia ha servido como mecanismo de **prueba de regresión automática** en cada cambio, clave para un desarrollo sostenible y profesional.

4.4.7. Pruebas manuales complementarias

Durante el desarrollo también se realizaron pruebas manuales, especialmente para:

- Validar animaciones en la UI (como el coloreo de casillas).
- Detectar errores de UX (posicionamiento de cursor tras error).
- Verificar comportamiento responsive y accesibilidad en distintos navegadores.

Estas pruebas se usaron para detectar errores funcionales no detectables automáticamente (como fallos visuales) y se tradujeron en pruebas automatizadas siempre que fue posible.

Durante la fase de pruebas manuales se identificaron dos defectos funcionales en la interfaz del juego *WordleApp*, que no habían sido detectados mediante pruebas automatizadas.

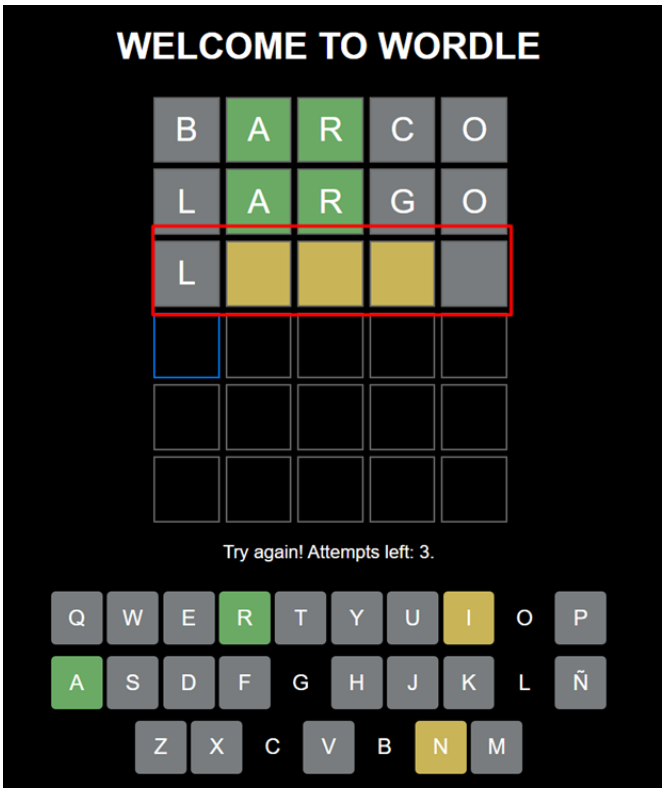

Figura 4.11: Resultado incorrecto tras introducir primero “WIANA” (no contenida en el diccionario) y luego “LIANA” (válida).

Primer fallo: inconsistencias visuales. Si el usuario introducía primero una palabra de longitud válida pero no contenida en el diccionario precargado (como “WIANA”), y a continuación una palabra válida y aceptada (como “LIANA”), el

sistema mostraba un estado visual incorrecto, como se observa en la Figura 4.11. Esto sucedía porque no se limpiaba correctamente el estado interno tras un intento inválido.

Segundo fallo: mensaje de error incorrecto. Cuando se introducía una palabra no contenida en el diccionario, y justo después, luego una palabra inválida por tener menos de 5 caracteres, el sistema seguía mostrando el mensaje:

```
Not in the list of valid words.
```

cuando en realidad el error correcto era:

```
Invalid input: The word must be 5 letters long.
```

Corrección aplicada. Se resolvieron ambos fallos:

- Se limpió correctamente el estado visual y lógico tras un intento inválido.
- Se implementó una validación jerárquica, priorizando primero la longitud y luego la verificación contra el diccionario.

Importancia de las pruebas exploratorias Este caso demuestra el valor de las pruebas manuales exploratorias para detectar errores que pueden pasar desapercibidos en la automatización. Resulta especialmente útil en fases donde interviene la experiencia de usuario, gestión de errores y validaciones visuales, permitiendo garantizar un comportamiento más robusto y coherente de la aplicación.

Para facilitar la validación de la funcionalidad de *Ranking* y *Leaderboard* sin necesidad de realizar partidas completas manualmente, se creó un componente auxiliar **TestDataLoader** junto a un controlador **TestDataController** expuesto únicamente bajo perfiles de desarrollo o prueba. Esto permitió insertar partidas predefinidas contra palabras secretas que se escogieran manualmente, para validar filtros, ordenaciones y la persistencia de datos.

Esta estrategia mejora la productividad durante la fase de testing, mantiene la base de datos controlada y no interfiere en el entorno de producción, gracias a la activación selectiva mediante perfiles de Spring Boot.

4.5. Distribución y despliegue

Una vez desarrollada la aplicación, ha sido necesario preparar su distribución y despliegue para poder ejecutarla en diferentes entornos de forma automatizada, reproducible y profesional. En este TFG se ha optado por utilizar **Docker**, una herramienta ampliamente adoptada en la industria para encapsular aplicaciones junto a sus dependencias en contenedores.

4.5.1. Contenerización con Docker

Se ha creado un `Dockerfile` que construye una imagen de la aplicación, basada en Java 21, que permite ejecutarla como un contenedor independiente. Esta imagen se compila automáticamente al hacer `merge` en la rama `master`, una vez que todas las pruebas han pasado y el análisis estático ha aprobado los criterios de calidad establecidos por SonarCloud.

Además, se han creado dos archivos `docker-compose` [23]:

- `docker-compose.yml`: orientado al entorno local y desarrollo. Incluye la aplicación junto a una base de datos MySQL, con configuración accesible para debugging.
- `docker-compose.prod.yml`: destinado a entornos de producción. Utiliza la imagen publicada en DockerHub con la etiqueta `latest`, lo que permite desplegar siempre la versión más reciente sin necesidad de recompilación.

Esta separación permite cumplir con buenas prácticas DevOps: desacoplar los entornos de desarrollo y producción, y asegurar reproducibilidad del entorno desplegado.

Publicación automática en DockerHub

Se ha configurado un workflow de GitHub Actions para implementar el principio de **Entrega Continua (CD)**. Este workflow se activa automáticamente cuando se hace push sobre `master`, lo cual ocurre únicamente tras haber superado todos los tests y análisis de calidad en las Pull Requests.

El siguiente fragmento de código muestra parte del workflow definido en el archivo `ci-cd.yml` que construye la imagen y la publica en DockerHub:

Código 4.16: Publicación en DockerHub

```
- name: Build and push Docker image
  run: |
    docker build -t mserral/wordleapp:${{ env.VERSION }}-${{
       env.GIT_SHA }} .
    docker tag mserral/wordleapp:${{ env.VERSION }}-${{
       env.GIT_SHA }} mserral/wordleapp:latest
    docker push mserral/wordleapp:${{ env.VERSION }}-${{
       env.GIT_SHA }}
    docker push mserral/wordleapp:latest
```

Esto garantiza que en cada nueva versión funcional de la aplicación, se genera automáticamente una imagen actualizada, versionada semánticamente (usando SemVer, y extrayendo la versión de la versión indicada en el pom.xml), y disponible públicamente para su ejecución.

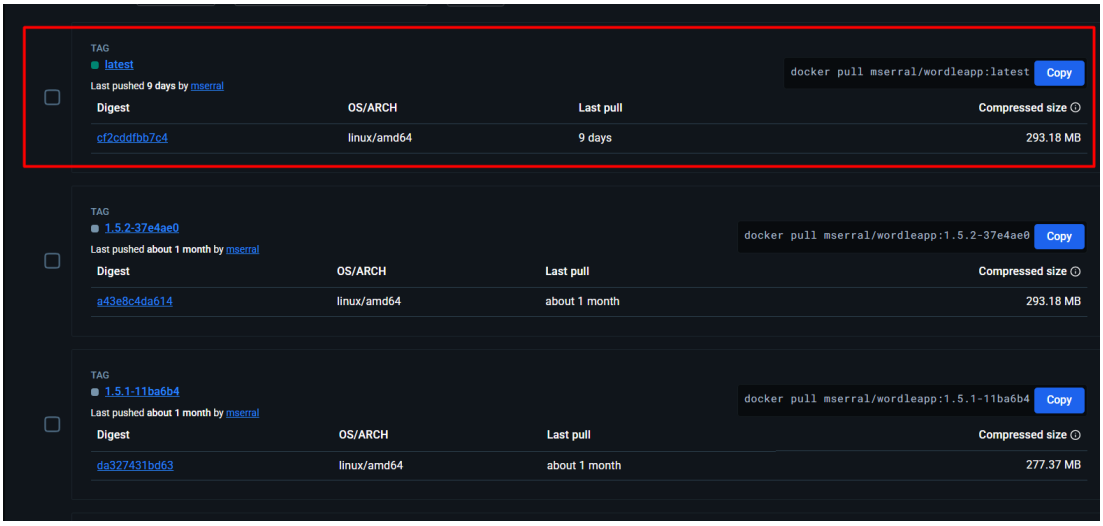

Figura 4.12: Imágenes generadas en DockerHub.

Despliegue mediante Docker

Gracias a la automatización anterior, el despliegue de la aplicación es inmediato. Los pasos están documentados en el `README.md`, y son los siguientes:

1. Descargar el archivo `docker-compose.prod.yml`.
2. Ejecutar:

```
docker pull mserral/wordleapp:latest
docker-compose -f docker-compose.prod.yml up
```

3. Acceder a `http://localhost:8080`

Descripción del archivo `docker-compose.prod.yml`

El archivo `docker-compose.prod.yml` define la configuración de despliegue en producción para la aplicación WordleApp. Está diseñado para levantar dos contenedores de forma orquestada:

- **wordle-mysql**: una base de datos MySQL persistente configurada con volúmenes y credenciales propias, que se ejecuta en el puerto `3307` del host.
- **wordle-app**: la aplicación WordleApp, descargada directamente desde DockerHub en su versión `latest`, que se ejecuta en el puerto `8080`.

Este fichero está orientado al despliegue en entornos finales (producción) y está preparado para ejecutarse con un único comando, sin necesidad de compilar ni instalar manualmente ninguna dependencia.

Código 4.17: Archivo docker-compose.prod.yml para despliegue en producción

```
1  version: "3.8"
2  services:
3    wordle-mysql:
4      image: mysql:8.3
5      container_name: wordle-mysql
6      restart: always
7      environment:
8        MYSQL_ROOT_PASSWORD: wordle2025
9        MYSQL_DATABASE: wordle
10     ports:
11       - "3307:3306"
12     volumes:
13       - mysql-data:/var/lib/mysql
14     healthcheck:
15       test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
16       interval: 5s
17       timeout: 5s
18       retries: 10
19   wordle-app:
20     image: mserral/wordleapp:latest
21     container_name: wordle-app
22     restart: on-failure
23     depends_on:
24       wordle-mysql:
25         condition: service_healthy
26     ports:
27       - "8080:8080"
28     environment:
29       SPRING_PROFILES_ACTIVE: docker
30       SPRING_DATASOURCE_URL:
            jdbc:mysql://wordle-mysql:3306/wordle?
31       serverTimezone=Europe/Madrid
32       SPRING_DATASOURCE_USERNAME: root
33       SPRING_DATASOURCE_PASSWORD: wordle2025
34       SPRING_JPA_HIBERNATE_DDL_AUTO: update
```

Este archivo asegura que la base de datos esté completamente disponible y saludable antes de lanzar la aplicación, gracias al bloque `depends_on` con condición `service_healthy`. La persistencia de datos se garantiza mediante el volumen `mysql-data`, lo cual es fundamental para conservar información entre reinicios.

El uso de variables de entorno permite a Spring Boot activar el perfil `docker` y conectar automáticamente a la base de datos sin necesidad de modificación manual del fichero de propiedades.

El archivo de producción se mantiene siempre actualizado, ya que la imagen con etiqueta `latest` apunta automáticamente a la última versión desplegada correctamente.

Relación con Integración Continua

Aunque la CI se trata en otra sección específica, es importante destacar que el sistema de CI/CD definido se alinea con la estrategia de control de calidad del proyecto. Las pruebas de regresión (tests unitarios, de integración y de interfaz gráfica) se ejecutan automáticamente en las Pull Requests. Solo si todas ellas son exitosas, se permite el `merge` y posterior despliegue.

Esta arquitectura garantiza que el producto publicado en DockerHub ha pasado por todas las validaciones necesarias, siguiendo la filosofía de calidad continua y prevención de errores en entornos productivos.

4.5.2. Despliegue alternativo desde el repositorio GitHub

Como alternativa para entornos de desarrollo o análisis más profundo, también es posible clonar el repositorio y ejecutar la aplicación directamente mediante Maven:

Código 4.18: Ejecución manual desde el repositorio GitHub.

```
1 git clone https://github.com/M-Serral/WordleApp.git
2 cd WordleApp
3 mvn spring-boot:run
```

Esto puede ser útil para depuración o desarrollo de nuevas funcionalidades, pero no se recomienda para despliegues en producción.

4.5.3. Publicación automatizada de releases en GitHub

Uno de los aspectos más relevantes del flujo de trabajo del proyecto es la automatización completa de la publicación de nuevas versiones funcionales mediante

GitHub Actions. Tras realizar un merge en la rama `master`, se activa un workflow que realiza los siguientes pasos de forma totalmente desatendida:

1. **Construcción de imagen Docker** con una etiqueta única basada en la versión del proyecto y el SHA del commit. Usar el SHA es útil, por si nos olvidamos de cambiar el número de versión en el pom.xml, y mergeamos a master una versión llamada de la misma manera que la anterior. Esto hace que en DockerHub, se publique una imagen nueva, en vez de borrar la anterior que ya existia y sustituirla por la nueva.
2. **Login en DockerHub** utilizando credenciales cifradas del repositorio.
3. **Publicación de la imagen** tanto con su versión concreta (`X.Y.Z-sha`) como con la etiqueta `latest`.
4. **Creación de un nuevo tag en Git** (Ver Código 4.19) a través de la API de GitHub, con el mismo nombre que la versión (‘vX.Y.Z‘) y usando como título el último Pull Request mergeado en `master`.
5. **Lectura del archivo** `RELEASE_NOTES.md` (Ver Código 4.19), el cuerpo de la release se genera a partir del contenido de este archivo, que se incluye en el repositorio y se actualiza manualmente antes de realizar el merge final. Este archivo documenta los cambios funcionales, correcciones, mejoras técnicas y nuevos requisitos introducidos de cada nueva versión.
6. **Publicación de una nueva Release en GitHub** (Ver Código 4.13), vinculada al tag, con los artefactos necesarios para el despliegue (`Dockerfile`, `docker-compose.yml`, etc.).

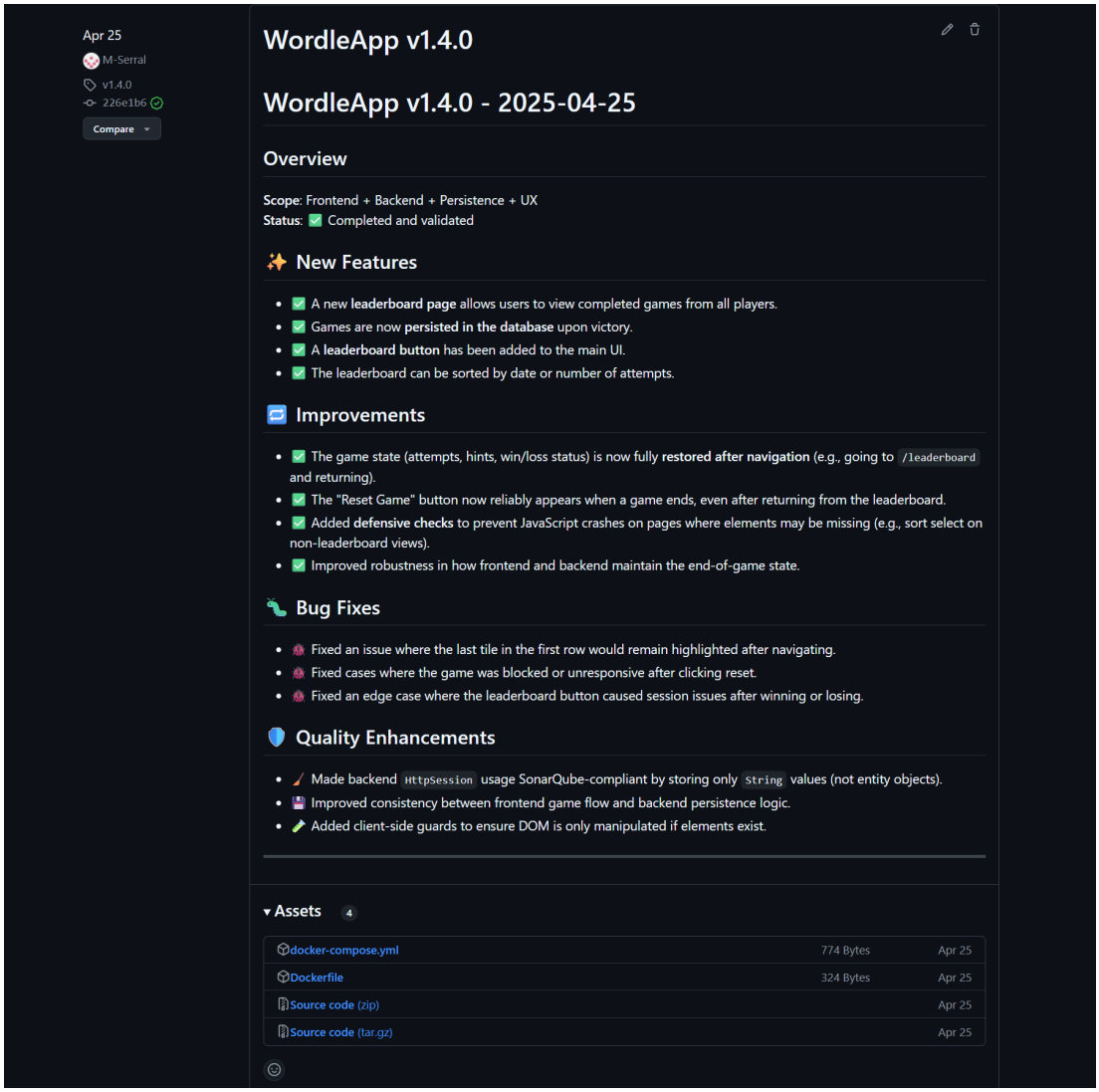

Figura 4.13: Publicación automática de una release en GitHub tras hacer push en master, usando el título del PR y el contenido del archivo RELEASE_NOTES.md.

El siguiente fragmento muestra parte de la configuración del workflow definida en el archivo `ci-cd.yml`, para automatizar este proceso de etiquetado de la versión, y publicación de la release:

Código 4.19: Automatización del título de la nueva release y su contenido.

```
- name: Create and publish Git Tag via GitHub API with PR title
# Get the latest merged PR title targeting master
          PR_TITLE=$(curl -s -H "Authorization: token $GH_TOKEN"
              \
            "https://api.github.com/repos/M-Serral/WordleApp/
            pulls?state=closed&base=master&sort=updated&direction
            =desc" \
            | jq -r '[.[] | select(.merged_at !=
                null)][0].title' | tr -d '\n')

          if [ -z "$PR_TITLE" ] || [ "$PR_TITLE" = "null" ]; then
            PR_TITLE="Release $TAG_NAME"
          fi
- name: Read release notes
id: release_notes
run: |
  RELEASE_BODY="$(cat RELEASE_NOTES.md)"
  echo "RELEASE_BODY<<EOF" >> $GITHUB_ENV
  echo "$RELEASE_BODY" >> $GITHUB_ENV
  echo "EOF" >> $GITHUB_ENV
```

Este proceso refuerza el enfoque de calidad del proyecto, permitiendo:

- Trazabilidad completa de versiones mediante etiquetado SemVer.
- Publicación inmediata de una release tras validación.
- Distribución automatizada del artefacto a través de DockerHub.

Esta estrategia garantiza que el software entregado no solo ha pasado todas las pruebas y análisis de calidad, sino que además está correctamente versionado, documentado y publicado, cerrando así el ciclo completo de integración y entrega continua.

5

Conclusiones y trabajos futuros

5.1. Evaluación del trabajo realizado

El desarrollo de WordleApp ha sido una experiencia completa y enriquecedora, tanto desde el punto de vista técnico como desde la perspectiva de calidad del software. El proyecto se ha llevado a cabo en distintas fases planificadas previamente, siguiendo una metodología iterativa e incremental. Gracias a esta planificación, ha sido posible cumplir todos los objetivos funcionales y no funcionales establecidos al inicio.

Desde las primeras versiones, que implementaban una funcionalidad básica del juego Wordle, hasta las últimas iteraciones con rankings, validaciones por diccionario y almacenamiento en base de datos, se ha mantenido un enfoque riguroso basado en buenas prácticas de ingeniería del software: uso de pruebas automatizadas, análisis estático de código con SonarCloud, control de versiones semántico, y despliegue continuo mediante DockerHub y GitHub Actions.

Además, se ha conseguido integrar exitosamente pruebas a tres niveles (unitarias, de integración y E2E), aplicando desarrollo guiado por pruebas (TDD) en varios módulos. Este enfoque ha permitido mantener una elevada cobertura de código y facilitar la detección temprana de errores.

5.2. Consecución de objetivos

Los objetivos definidos en el capítulo 2 han sido cumplidos en su totalidad:

- Desarrollo de una aplicación web funcional que reproduce el comportamiento del juego Wordle.
- Diseño e implementación de una arquitectura modular con Spring Boot y MySQL.
- Implementación de una interfaz HTML dinámica con JavaScript y Mustache.
- Integración de un sistema completo de rankings y validación de palabras con diccionario.
- Incorporación de pruebas automatizadas (JUnit, Selenium, Rest Assured).
- Configuración de una pipeline CI/CD efectiva mediante GitHub Actions.
- Dockerización del backend/frontend y publicación de imágenes versionadas en DockerHub.
- Análisis de calidad del código con SonarCloud aplicado a cada iteración menor.

Además, se han introducido mejoras técnicas progresivas, como la optimización del feedback visual en el frontend, la implementación de lógica compleja para ranking por palabra y el control de deuda técnica en cada release.

5.3. Trabajos futuros

Aunque el proyecto se encuentra en una versión madura y plenamente funcional, existen varias líneas de mejora que podrían abordarse en fases futuras:

- Gestión de usuarios: se podría añadir un sistema de autenticación (registro/login) para guardar las partidas de forma personalizada y permitir estadísticas por usuario.
- Internacionalización: soporte multilingüe tanto para el frontend como para el conjunto de palabras válidas.
- Visualización de estadísticas: incorporación de gráficos (bar charts, line charts) para mostrar evolución de resultados, rankings globales, etc.

- Sistema de comentarios o valoraciones: permitiendo interacción entre usuarios, especialmente en rankings.
- Modo multijugador por turnos: como ampliación lúdica, podría desarrollarse una versión competitiva en la que dos jugadores intentan adivinar la misma palabra alternativamente.
- Despliegue en la nube (IaaS/PaaS): el sistema actual está listo para producción, pero podría integrarse en servicios como Railway, Heroku o AWS.

Estas posibles ampliaciones permitirían no solo seguir explorando nuevas tecnologías, sino también mejorar la experiencia de usuario y ofrecer una aplicación más rica y completa.

5.4. Conclusiones personales

Este Trabajo de Fin de Grado ha supuesto una oportunidad extraordinaria para consolidar conocimientos adquiridos durante el grado, pero también para afrontar retos reales que habitualmente se encuentran en proyectos profesionales: organización modular del código, diseño desde una perspectiva de calidad, automatización de pruebas, control de calidad, integración continua, gestión de versiones, etc.

Uno de los mayores aprendizajes ha sido adquirir una disciplina de trabajo centrada en la calidad desde la primera línea de código. La integración de SonarCloud como parte del pipeline, el uso sistemático de TDD en las funcionalidades críticas, y el análisis de métricas en cada versión menor me han permitido entender de forma práctica el impacto de la deuda técnica y la importancia del mantenimiento del software a largo plazo.

Otro aspecto especialmente valioso ha sido el contacto profundo con herramientas reales de la industria como Docker, GitHub Actions, Selenium o Rest Assured, todas ellas esenciales hoy día en cualquier entorno de desarrollo profesional como ingeniero de calidad.

A nivel personal, ha sido un reto que me ha permitido demostrar que puedo planificar, diseñar, implementar, validar y documentar una solución técnica de principio a fin, manteniendo en todo momento el foco en la calidad del producto. Esto no solo me ha hecho crecer como ingeniero, sino también como profesional responsable y autónomo.

En definitiva, WordleApp no ha sido solo un proyecto académico, sino una simulación real de cómo abordar el desarrollo y la validación de un producto de software robusto, útil y mantenible.

Bibliografía

[1] Oracle. Java 21 documentation. [Online]. Available: https://docs.oracle.com/en/java/javase/21/.

[2] S. Team. Spring boot documentation. [Online]. Available: https://spring.io/projects/spring-boot.

[3] Oracle. Mysql reference manual. [Online]. Available: https://dev.mysql.com/doc/refman/8.3/en/.

[4] M. W. Docs. Html: Hypertext markup language. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML.

[5] ——. Javascript guide. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide.

[6] Mustache. Mustache templates documentation. [Online]. Available: https://mustache.github.io/.

[7] R. A. Project. Rest assured - testing and validating rest services in java. [Online]. Available: https://rest-assured.io/.

[8] J. Team. Junit 5 user guide. [Online]. Available: https://junit.org/junit5/docs/current/user-guide/.

[9] M. Contributors. Mockito documentation. [Online]. Available: https://site.mockito.org/.

[10] S. Team. Testing in spring boot. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/html/features.htmlfeatures.testing.

[11] ——. Mockmvc — spring framework reference documentation. [Online]. Available: https://docs.spring.io/spring-framework/reference/testing/web/webmvc.html.

[12] S. HQ. Selenium webdriver documentation. [Online]. Available: https://www.selenium.dev/documentation/webdriver/.

[13] SonarSource. Sonarcloud documentation. [Online]. Available: https://docs.sonarcloud.io/.

[14] S. Team. Test coverage execution with jacoco — sonarcloud docs. [Online]. Available: https://sonarcloud.io/documentation/enriching/test-coverage/java/.

[15] G. SCM. Git - distributed version control system. [Online]. Available: https://git-scm.com/.

[16] GitHub. Github docs - collaborative development. [Online]. Available: https://docs.github.com/.

[17] D. Inc. Docker documentation. [Online]. Available: https://docs.docker.com/.

[18] ——. Docker hub. [Online]. Available: https://hub.docker.com/.

[19] T. Preston-Werner. Semantic versioning 2.0.0. [Online]. Available: https://semver.org/.

[20] M. Fowler. Test-driven development (tdd). [Online]. Available: https://martinfowler.com/bliki/TestDrivenDevelopment.html.

[21] Atlassian. Git flow workflow. [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

[22] GitHub. Github actions documentation. [Online]. Available: https://docs.github.com/en/actions.

[23] D. Inc. Docker compose command-line reference. [Online]. Available: https://docs.docker.com/engine/reference/commandline/compose$_up$/.

Apéndices

A

Repositorio GitHub

A.1. Enlace al repositorio

El proyecto ha sido alojado en la plataforma GitHub.

El enlace es el siguiente: `https://github.com/M-Serral/WordleApp`