**1. Introducción**

El objetivo de la aplicación es proporcionar al usuario una experiencia que le permita ver información actual de los pilotos y circuitos de la temporada 2025 de la Formula 1, comprar cartas de sus pilotos y circuitos favoritos para luego poder competir en una tabla general por ver que usuario consigue coleccionar más cartas. Además, también pueden crear su propia carta y que todos los usuarios puedan ver las cartas personalizadas de los demás. Aparte de poder ver las cartas personalizadas del resto de usuarios también pueden ver las colecciones de los demás.

El público objetivo de la aplicación son usuarios normales, con interés en el deporte de motor y sobre todo el gran circo de la Formula 1.

Las tecnologías utilizadas para este proyecto son:

1. Spring Boot con Java.
2. Angular con Typescript, HTML, y CSS.
3. MySQL.
4. Railway y Vercel.
5. Github.

**2. Arquitectura del Sistema**

Frontend:

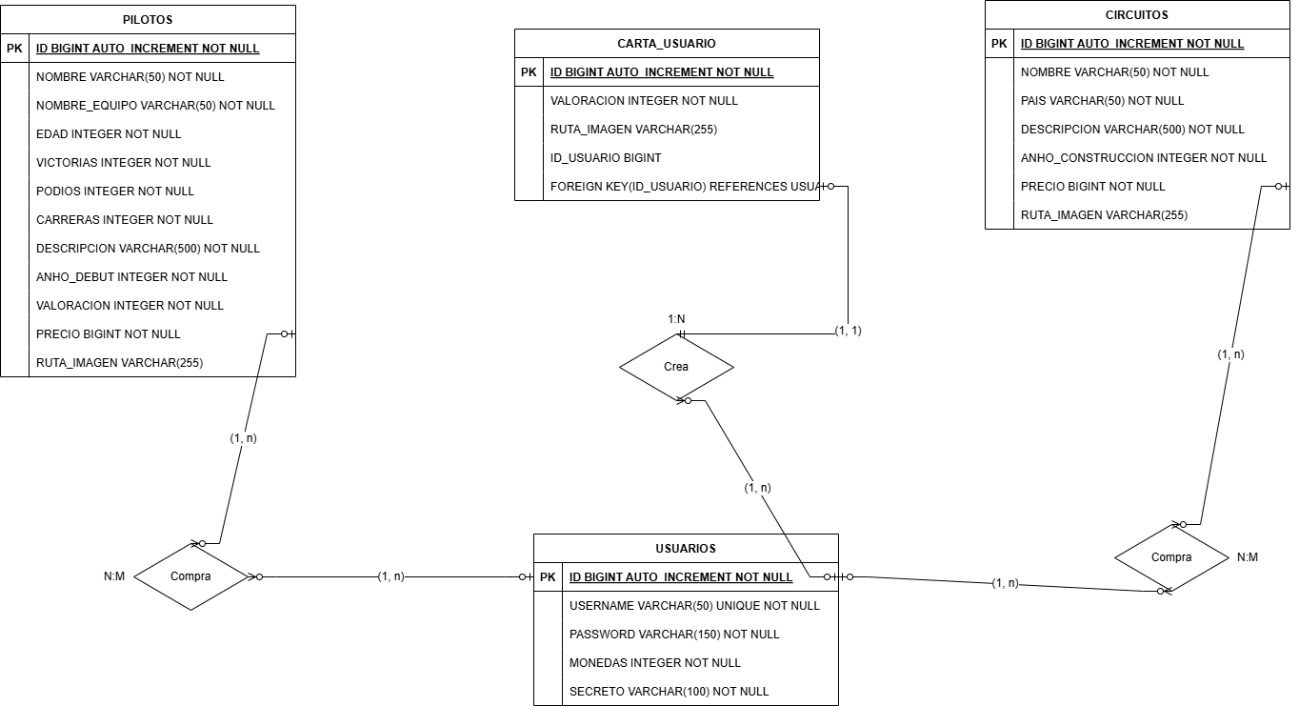
* Angular: es un Framework web de código abierto creado por Google. El gran punto fuerte respecto a su competencia como React o Vue es que es una solución completa, te ofrece todo lo necesario, por lo que, aunque existen herramientas externas esas funcionalidades ya las cubre Angular. Cabe destacar que no todo son ventajas, si Angular tiene algo malo es su curva de aprendizaje, mucho mayor que la de su competencia. En el proyecto si se ha hecho uso de librerías externas como:
  + PrimeNG: es una biblioteca de componentes para la interfaz de usuario. Específicamente en el proyecto se ha implementado para mostrar las validaciones de las contraseñas, filtros para bloquear ciertos caracteres y espacios en algunos inputs, iconos a lo largo de toda la aplicación, o botones entre otros.
  + SweetAlert: es una biblioteca para mostrar ventanas emergentes. Específicamente en el proyecto se ha implementado en el momento de registrar una cuenta, iniciar sesión, y crear tu carta personalizada, tanto para mostrar que ha funcionado correctamente o el error que ha sucedido.
* Typescript: se puede denominar la versión avanzada de JavaScript, desarrollado por Microsoft. Su éxito radica en que mejora a JavaScript con múltiples funcionalidades extra, pero destacando sobre todo el tipado estático, lo que mejora la seguridad y calidad de código, detectando errores antes de la ejecución y no causándolos durante la misma. Typescript es culpable en gran medida de lo bueno y malo que tiene Angular como su curva de aprendizaje, que es mayor a la de JavaScript.
* HTML: es el lenguaje de marca estándar en el desarrollo web, está conformado por etiquetas que describen la estructura de la página web, y destaca por su gran facilidad. Pero por si solo se le puede denominar como “inútil” ya que carece de estilos e interactividad, el corazón de una página web.
* CSS: lenguaje de estilos utilizado para definir la apariencia del contenido HTML de una página web, resolviendo esa carencia de estilos. Destaca por su flexibilidad e infinidad de posibilidades para estilizar una página web. Aunque para aplicar estilos básicos parece inofensivo a nivel avanzado su curva de aprendizaje se puede volver muy grande. Cabe destacar que es en gran parte el limitante en la tabla de compatibilidad detallada en el manual del usuario, muchas características solo funcionan en navegadores muy actuales.
* Visual Studio Code: entorno de desarrollo, diseñado por Microsoft. Destaca por su gran compatibilidad, no solo a nivel de Sistemas Operativos, también a nivel de lenguajes de programación, dispone de extensiones para cualquier lenguaje de programación medianamente conocido.

Backend:

* Spring Boot: es un framework de código abierto para desarrollo de APIS y páginas web. Es una extensión de Spring Framework pensada para facilitar la configuración inicial de la aplicación. Es un framework muy potente, pero por lo tanto pesado. Cuenta con un gran ecosistema de funcionalidades como Spring Data, Spring Security o Spring Cloud, que, si bien a la larga son muy útiles, para un desarrollador nuevo al no estar familiarizado con las peculiaridades de Spring supone una gran curva de aprendizaje.
* Java: lenguaje de programación orientado a objetos, propiedad de Oracle. Destaca por la famosa JVM la cual hace que los programas Java puedan ejecutarse de forma nativa en cualquier plataforma gracias a que ejecuta el bytecode generado por el compilador, el cual es un formato independiente de la plataforma. Si bien la JVM está excepcionalmente optimizada al límite de sus posibilidades sigue haciendo a Java un lenguaje más lento y pesado que aquellos compilados directamente a código máquina.
* MySQL: sistema de gestión de bases de datos relacionales, también propiedad de Oracle. Utiliza el lenguaje de consulta estructurado (SQL) que organiza los datos en tablas compuestas de filas y columnas. Se caracteriza por su facilidad de uso al contar con herramientas gráficas, buen rendimiento y compatibilidad multiplataforma. El esquema fijo en tablas de SQL ofrece ventajas, pero en temas de consultas complejas se queda corto comparado con las bases de datos NoSQL gracias a su flexibilidad. Además, comparado con otros sistemas de gestión relacionales carece de funciones avanzadas, mientras que en MySQL para la generación de claves automáticas hay que conformarse con el AUTO\_INCREMENT en otros sistemas como PostgreSQL existen las secuencias, que aumentan la personalización al definir saltos específicos, valores mínimos y máximos, reinicios automáticos y un mejor rendimiento.
* JWT: es un estándar que define una forma segura de transmitir información haciendo uso de tokens, que están firmados digitalmente para garantizar su integridad y autenticidad, y contienen los datos en forma de JSON. Los tokens contienen toda la información del usuario como correo, nombre de usuario o sus permisos lo que facilita el flujo de la aplicación, y gracias a que son compactos se pueden transmitir fácilmente. Si cabe destacar algo es que los tokens JWT no están cifrados, tan solo firmados para verificar su integridad, lo que significa que si cualquier persona se hace con un token que no sea suyo puede ver todo su contenido como el correo o los permisos del usuario. Además, estos tokens son válidos hasta que llegue su fecha de expiración, para conseguir revocarlos sería necesario almacenarlos en el servidor por ejemplo en una base de datos, lo cual desvirtúa en gran parte la naturaleza y el propósito de JWT.
* Eclipse: es un entorno de desarrollo, principalmente pensado para el desarrollo de aplicaciones en C, C++ y Java. Cuenta con una gran comunidad, gran cantidad de extensiones y compatibilidad multiplataforma. El gran obstáculo de Eclipse es su consumo de recursos y su interfaz más tradicional. La elección de Eclipse por ejemplo sobre otros entornos como IntelliJ ha sido por su facilidad para integrarlo con Spring Boot, con una extensión prácticamente “Plug & Play”, mientras que IntelliJ presentaba más problemas.

**3. Fases de desarrollo**

Análisis y diseño:

* La base de datos está compuesta de 6 tablas
  + Usuarios: representa los usuarios de la aplicación. Su Primary Key es el id, un bigint auto\_increment. Además, contiene el email, número de monedas, secreto (código para obtener los OTP), y la contraseña la cual está cifrada para aumentar la seguridad.
  + Pilotos: representan los pilotos. Su Primary Key es el id, un bigint auto\_increment. Contiene además su nombre, nombre de su equipo, edad, victorias, podios, carreras, una descripción corta, su año de debut, valoración, precio y la ruta donde se almacena su imagen.
  + Circuitos: representan los circuitos. Su Primary Key es el id, un bigint auto\_increment. Contiene además su nombre, país, una descripción corta, su año de construcción, precio y la ruta donde se almacena su imagen.
  + Carta\_Usuario: representan las cartas customizadas que han creado los usuarios. Su Primary Key es el id, un bigint auto\_increment. Contiene además su valoración, ruta donde se almacena su imagen y el id del usuario al que pertenece.
  + Usuario\_Pilotos: representan las cartas de los pilotos que ha comprado el usuario. Su Primary Key son la combinación del id del piloto y el id del usuario, siendo ambos todo el contenido de la tabla.
  + Usuario\_Circuitos: representan las cartas de los circuitos que ha comprado el usuario. Su Primary Key son la combinación del id del circuito y el id del usuario, siendo ambos todo el contenido de la tabla.
* Las relaciones que conforman la base de datos son las siguientes:
  + Usuarios y Carta\_Usuario: un Usuario puede tener de una a muchas Carta\_Usuario (1, n) mientras que una Carta\_Usuario pertenece solo a un Usuario (1, 1) lo que conforma una relación uno a muchos 1:N.
  + Usuarios y Pilotos: un usuario puede comprar de una a muchos Pilotos (1, n) y un piloto puede ser propiedad de uno a muchos Usuarios (1, n) lo que conforma una relación muchos a muchos N:M que provoca la creación de una tercera tabla Usuario\_Pilotos para representar la relación correctamente.
  + Usuarios y Circuitos: un usuario puede comprar de uno a muchos Circuitos (1, n) y un circuito puede ser propiedad de uno a muchos Usuarios (1, n) lo que conforma una relación muchos a muchos N:M que provoca la creación de una tercera tabla Usuario\_Circuitos para representar la relación correctamente.

Desarrollo: los endpoints disponibles son los siguientes

**AUTENTICACIÓN**

|  |  |  |
| --- | --- | --- |
| URL | /autentificación/login | |
| Método | POST | |
| Descripción | Permite el inicio de sesión de un usuario | |
| Variable de ruta | *~~No aplica~~* | |
| Parámetros de consulta | username | Correo del usuario |
| password | Contraseña |
| otp | Código OTP |
| Cuerpo de la solicitud | *~~No aplica~~* | |
| Códigos de respuesta | 200 | Correcto |
| 401 | OTP incorrecto |
| 403 | Credenciales incorrectas |
| Cuerpo de respuesta | {  "token": "string"  } | |
| Autenticación | Código OTP (Autenticación de 2 Factores) | |

|  |  |  |
| --- | --- | --- |
| URL | /autentificación/registro | |
| Método | POST | |
| Descripción | Permite la creación de una nueva cuenta | |
| Variable de ruta | *~~No aplica~~* | |
| Parámetros de consulta | *~~No aplica~~* | |
| Cuerpo de la solicitud | {  "username": "string",  "password": "string"  } | |
| Códigos de respuesta | 200 | Correcto |
| 404 | Usuario ya existente |
| Cuerpo de respuesta | {  "qr": "String",  "secreto": "String"  } | |
| Autenticación | *~~No aplica~~* | |

**CARTAS**

|  |  |  |
| --- | --- | --- |
| URL | /cartas/uploads/{id} | |
| Método | POST | |
| Descripción | Permite crear una nueva carta | |
| Variable de ruta | Id: Identificador del usuario que crea la carta | |
| Parámetros de consulta | imagen | Archivo con la imagen |
| valoracion | Valoración (1-99) de la carta |
| Cuerpo de la solicitud | *~~No aplica~~* | |
| Códigos de respuesta | 200 | Correcto |
| 400 | El usuario ya tiene una carta |
| Cuerpo de respuesta | *~~No aplica~~* | |
| Autenticación | Token JWT | |

|  |  |  |
| --- | --- | --- |
| URL | /cartas/{id} | |
| Método | GET | |
| Descripción | Obtiene las cartas compradas de un usuario | |
| Variable de ruta | Id: Identificador del usuario | |
| Parámetros de consulta | *~~No aplica~~* | |
| Cuerpo de la solicitud | *~~No aplica~~* | |
| Códigos de respuesta | 200 | Correcto |
| 404 | El usuario no existe |
| Cuerpo de respuesta | {  "pilotos": [  {  "id": int,  "nombre": "string",  "nombreEquipo": "string",  "edad": int,  "victorias": int,  "podios": int,  "carreras": int,  "descripcion": "string",  "anhoDebut": int,  "valoracion": int,  "precio": int,  "rutaImagen": "string"  }  ],  "circuitos": [  {  "id": int,  "nombre": "string",  "pais": "string",  "descripcion": "string",  "anhoConstruccion": int,  "precio": int,  "rutaImagen": "string"  }  ]  } | |
| Autenticación | Token JWT | |

|  |  |  |
| --- | --- | --- |
| URL | /cartas/{id} | |
| Método | PATCH | |
| Descripción | Actualiza las cartas y/o monedas de un usuario | |
| Variable de ruta | Id: Identificador del usuario | |
| Parámetros de consulta | monedas | Monedas del usuario |
| idPiloto | Id del piloto (si aplica) |
| idCircuito | Id del circuito (si aplica) |
| Cuerpo de la solicitud | *~~No aplica~~* | |
| Códigos de respuesta | 200 | Correcto |
| 404 | El usuario no existe |
| Cuerpo de respuesta | *~~No aplica~~* | |
| Autenticación | Token JWT | |

|  |  |  |
| --- | --- | --- |
| URL | /cartas/usuarios | |
| Método | GET | |
| Descripción | Obtiene las cartas personalizadas de todos | |
| Variable de ruta | *~~No aplica~~* | |
| Parámetros de consulta | *~~No aplica~~* | |
| Cuerpo de la solicitud | *~~No aplica~~* | |
| Códigos de respuesta | 200 | Correcto |
| 500 | Error inesperado |
| Cuerpo de respuesta | [  {  "valoracion": int,  "rutaImagen": "string",  "username": "string"  }  ] | |
| Autenticación | Token JWT | |

|  |  |  |
| --- | --- | --- |
| URL | /cartas/usuarios/{id} | |
| Método | GET | |
| Descripción | Obtiene la carta personalizada de un solo usuario | |
| Variable de ruta | Id: Identificador del usuario | |
| Parámetros de consulta | *~~No aplica~~* | |
| Cuerpo de la solicitud | *~~No aplica~~* | |
| Códigos de respuesta | 200 | Correcto |
| 404 | El usuario no existe |
| Cuerpo de respuesta | {  "valoracion": int,  "rutaImagen": "string",  "username": "string"  } | |
| Autenticación | Token JWT | |

**USUARIOS**

|  |  |  |
| --- | --- | --- |
| URL | /usuarios/{id}/monedas | |
| Método | GET | |
| Descripción | Obtiene las monedas de un usuario | |
| Variable de ruta | Id: Identificador del usuario | |
| Parámetros de consulta | *~~No aplica~~* | |
| Cuerpo de la solicitud | *~~No aplica~~* | |
| Códigos de respuesta | 200 | Correcto |
| 404 | El usuario no existe |
| Cuerpo de respuesta | {  "monedas": int  } | |
| Autenticación | Token JWT | |

|  |  |  |
| --- | --- | --- |
| URL | /usuarios/ranking | |
| Método | GET | |
| Descripción | Obtiene el ranking de los usuarios | |
| Variable de ruta | *~~No aplica~~* | |
| Parámetros de consulta | *~~No aplica~~* | |
| Cuerpo de la solicitud | *~~No aplica~~* | |
| Códigos de respuesta | 200 | Correcto |
| 500 | Error del servidor |
| Cuerpo de respuesta | [  {  "username": "string",  "cantidadPilotos": int,  "cantidadCircuitos": int,  "cantidadTotal": int,  "pilotos": [  {  "id": int,  "nombre": "string",  "nombreEquipo": "string",  "edad": int,  "victorias": int,  "podios": int,  "carreras": int,  "descripcion": "string",  "anhoDebut": int,  "valoracion": int,  "precio": int,  "rutaImagen": "string"  }  ],  "circuitos": [  {  "id": int,  "nombre": "string",  "pais": "string",  "descripcion": "string",  "anhoConstruccion": int,  "precio": int,  "rutaImagen": "string"  }  ]  }  ] | |
| Autenticación | Token JWT | |

**PILOTOS**

|  |  |  |
| --- | --- | --- |
| URL | /pilotos | |
| Método | GET | |
| Descripción | Obtiene todos los pilotos | |
| Variable de ruta | *~~No aplica~~* | |
| Parámetros de consulta | *~~No aplica~~* | |
| Cuerpo de la solicitud | *~~No aplica~~* | |
| Códigos de respuesta | 200 | Correcto |
| 500 | Error del servidor |
| Cuerpo de respuesta | [  {  "id": int,  "nombre": "string",  "nombreEquipo": "string",  "edad": int,  "victorias": int,  "podios": int,  "carreras": int,  "descripcion": "string",  "anhoDebut": int,  "valoracion": int,  "precio": int,  "rutaImagen": "string"  }  ] | |
| Autenticación | Token JWT | |

**CIRCUITOS**

|  |  |  |
| --- | --- | --- |
| URL | /circuitos | |
| Método | GET | |
| Descripción | Obtiene todos los circuitos | |
| Variable de ruta | *~~No aplica~~* | |
| Parámetros de consulta | *~~No aplica~~* | |
| Cuerpo de la solicitud | *~~No aplica~~* | |
| Códigos de respuesta | 200 | Correcto |
| 500 | Error del servidor |
| Cuerpo de respuesta | [  {  "id": int,  "nombre": "string",  "pais": "string",  "descripcion": "string",  "anhoConstruccion": int,  "precio": int,  "rutaImagen": "string"  }  ] | |
| Autenticación | Token JWT | |

La aplicación sigue una arquitectura general de cliente-servidor y además desacoplada, ya que por una parte está el Frontend desarrollado como una SPA mediante Angular, y por otro lado tenemos el Backend desarrollado como una API REST mediante Spring Boot, las cuales se comunican constantemente para el funcionamiento de la aplicación. Si desglosamos la arquitectura de la aplicación diferenciando entre ambas capas:

* Frontend: Angular sigue el patrón MVVM:
  + Model: interfaces y clases que representan los datos para interactuar con el Backend, ya sean datos recibidos o mandados.
  + View: formado por todo lo que el usuario ve, en este caso son el HTML, CSS, e imágenes.
  + ViewModel: conecta la vista con los datos de la aplicación, es el encargado de manejar toda la lógica de la aplicación, en este caso son los archivos Typescript.
* Backend: Angular sigue el patrón MVC:
  + Model: en este caso representa los datos como por ejemplo DTOs o las entidades recogidas de base de datos mediante un ORM, pero también toda la lógica de negocio dentro de los servicios.
  + View: en este caso debido a Angular la vista no se genera desde Spring Boot, pero en caso de que no fuese así el servidor se encargaría de generar el HTML con Thymeleaf o JSP.
  + Controller: es el encargado de atender las peticiones del Frontend, recibe datos del mismo y llama a los servicios para que los procesen correctamente y luego se encarga de devolver la respuesta correspondiente y los datos necesarios al Frontend, ya sea correcto o no. El controlador simplemente actúa como un intermediario, solo se encarga de llamar al servicio con la información que ha recibido y de interpretar el resultado del servicio para comunicarlo al Frontend.

La organización de paquetes es la siguiente:

com.arjona.f1gameinfo/

├── F1GameInfoApplication.java # Clase principal de arranque de la aplicación

├── business/

│ ├── model/ # Modelo y DTOs

│ │ ├── CartaCompradaDTO.java

│ │ ├── CartaUsuario.java

│ │ ├── CartaUsuarioDTO.java

│ │ ├── Circuito.java

│ │ ├── Piloto.java

│ │ ├── RankingDTO.java

│ │ └── UsuarioDTO.java

│ │

│ ├── repositories/ # Interfaces de acceso a datos (JPA)

│ │ ├── CartaUsuarioRepository.java

│ │ ├── CircuitoRepository.java

│ │ └── PilotoRepository.java

│ │

│ ├── services/ # Interfaces de servicios de negocio

│ │ ├── CartaServices.java

│ │ ├── CircuitoServices.java

│ │ ├── PilotoServices.java

│ │ └── UsuarioNoAuthServices.java

│ │

│ └── services/impl/ # Implementaciones de los servicios

│ ├── CartaServicesImpl.java

│ ├── CircuitoServicesImpl.java

│ ├── PilotoServicesImpl.java

│ └── UsuarioNoAuthServicesImpl.java

├── presentation/

│ ├── WebConfig.java # Configuración general del contexto web

│ │

│ ├── config/ # Gestión de errores y excepciones personalizadas

│ │ ├── GestorExcepciones.java

│ │ ├── HttpCustomError.java

│ │ └── StatusException.java

│ │

│ └── restcontrollers/ # Controladores REST que exponen la API

│ ├── CartaController.java

│ ├── CircuitoController.java

│ ├── PilotoController.java

│ └── UsuarioNoAuthController.java

├── security/

│ ├── FiltroAutentificacionJWT.java # Filtro de autenticación con JWT

│ ├── FiltroAutentificacionOTP.java # Filtro de autenticación con OTP

│ ├── SeguridadConfig.java # Configuración de seguridad (Spring Security)

│ ├── UsuarioDetails.java # Implementación de UserDetails

│ ├── UsuarioDetailsService.java # Servicio de carga de usuarios para autenticación

│ ├── UtilsJWT.java # Utilidades para manejo de JWT

│ └── UtilsOTP.java # Utilidades para manejo de OTP

│

├── security/integration/

│ ├── model/ # Entidad de usuario para autenticación

│ │ └── Usuario.java

│ │

│ ├── repositories/ # Repositorio JPA para autenticación

│ │ └── UsuarioRepository.java

│ │

│ ├── services/ # Servicio de autenticación

│ │ └── UsuarioServices.java

│ │

│ └── services/impl/ # Implementación del servicio de autenticación

│ └── UsuarioServicesImpl.java

├── security/payloads/ # Clases para peticiones y respuestas de autenticación

│ ├── JwtResponse.java

│ ├── LoginRequest.java

│ ├── RegisterRequest.java

│ └── RegisterResponse.java

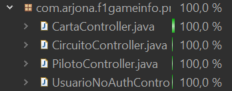
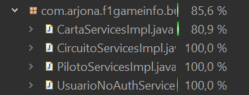
└── security/presentation/restcontrollers/

└── AutentificacionController.java # Controlador REST para login y registro

Pruebas: para el testing se ha hecho uso de:

* JUnit: es un marco de pruebas unitarias. Permite la creación de métodos que verifican el comportamiento del código de la aplicación en diferentes escenarios, ya sea cuando todo va bien o cuando ocurren errores. Su mayor ventaja es la simplicidad.
* Mockito: es utilizado para crear objetos simulados en el testing. Digamos que quieres probar el comportamiento de los servicios de tu aplicación que dependen a su vez del repositorio, con Mockito lo que haces es simular el comportamiento del repositorio, y así verificas el comportamiento del servicio dependiendo de lo que reciba del repositorio, y, además, reduces las pruebas solo al servicio, teniendo que crear otras para el repositorio lo que mejora la calidad de las mismas.
* TestContainers: es una biblioteca de pruebas que permite hacer uso de contenedores de Docker a la hora de probar la aplicación. A la hora de hacer pruebas de los controladores Spring Boot intenta arrancar el contexto entero de la aplicación, lo que hace que intenten conectarse a la base de datos de la aplicación la cuál no está en marcha, con esta biblioteca se crea un entorno aislado que aloja la base de datos solo durante el tiempo de ejecución de las pruebas, facilitando la configuración de las mismas.

La cantidad del código de la aplicación que las pruebas cubren, es decir, la cantidad de cobertura de la aplicación se sitúa en un 84,1%, por encima del recomendado 80%. 

Con lo mas importante siendo los servicios y controladores de la aplicación que contienen prácticamente toda la lógica situandose en un 85,6% y 100% respectivamente, por encima del resto de la aplicación.

**4. Despliegue**

Para el despliegue de la aplicación se ha hecho uso de dos plataformas distintas, una para la base de datos y el Backend, y otra para el Frontend:

1. Railway
   1. ¿Qué es?: es una plataforma de despliegue de aplicaciones que permite desarrollar, desplegar y escalar aplicaciones de manera sencilla, especializada en bases de datos y Backend. Aventaja a toda su competición debido a las facilidades que ofrece mediante Nixpacks. Nixpacks es una herramienta propia de Railway que automatiza el proceso de creación, empaquetamiento y despliegue de aplicaciones que permite desplegar una aplicación sin ningún tipo de configuración avanzada, esto lo consigue mediante la creación de una imagen OCI, que integra con Docker y un plan de construcción de manera automática.
   2. Proceso de despliegue: el despliegue de la base de datos ha sido bastante rápido, simplemente consiste en crear una imagen Docker de MySQL desde la interfaz de Railway y crear tu esquema dentro.

El despliegue del Backend tiene más dificultad, pero sigue sin ser demasiado complicado. Lo primero es dentro del mismo proyecto donde está la base de datos crear un nuevo servicio, en mi caso desde un repositorio de Github, seleccionar la carpeta dentro del repositorio donde se encuentra ubicado mi proyecto Spring Boot y luego conectar la rama principal para que cada vez que subo un nuevo cambio a la misma, redespliegue la aplicación automáticamente. Gracias a que está dentro del mismo proyecto puedes conectar el Backend a la base de datos desde la red interna del proyecto reduciendo costes de uso y usando las variables de entorno que te da Railway como la url, contraseña y usuario de la base de datos.

1. Vercel
   1. ¿Qué es?: es una plataforma de desarrollo y despliegue de aplicaciones web, que soporta múltiples Frameworks web incluyendo Angular, React o Vue.js.
   2. Proceso de despliegue: lo primero de todo es seleccionar el Framework, en este caso Angular. Al igual que el Backend, es necesario conectar el repositorio de Github, escoger la rama, y la carpeta donde está ubicado el proyecto. El mayor problema fue que al construir la aplicación no se creaba el index.html como punto de entrada para que Vercel pudiese desplegar la aplicación automáticamente, esto es debido a la arquitectura Angular Universal, para conseguir que crease el punto de entrada es necesario pasar a la arquitectura SPA (aplicación de una sola página) retocando los archivos de configuración dentro del proyecto de Angular.

**5. Futuras mejoras**

1. SonarQube: es una herramienta que permite identificar errores y vulnerabilidades en el código. Lo consigue mediante:
   1. Cobertura de pruebas: evalúa la cobertura de las pruebas unitarias, para asegurar que el código esté correctamente probado.
   2. Code Smells: son problemas dentro del código que pueden derivar en problemas de mantenibilidad y seguridad a lo largo del tiempo.
   3. Deuda técnica: Es el coste acumulado de malas prácticas dentro del código incluyendo los Code Smells, falta de documentación, tecnología obsoleta. Representado en forma de tiempo estimado para resolverlo.
   4. Valoración final: mediante lo explicado anteriormente SonarQube incluye en el reporte una valoración final que indica si el código cumple el estándar necesario o no.
2. Convertir el proyecto en microservicios: gracias a dividir el proyecto en múltiples servicios se vería beneficiado en:
   1. Resistencia: en un monolito sin falla algo cae la aplicación entera, mientras que con microservicios si uno falla el resto siguen activos.
   2. Escalabilidad: cada servicio puede desarrollarse de manera independiente según sus necesidades sin depender del resto o verse limitado en sus capacidades.
   3. Detección de fallos: si suceden errores es más fácil aislarlo y detectar su procedencia para resolverlo con mayor rapidez.

Estos microservicios gracias a Spring Cloud estarían conectados a un Eureka Server, y mediante una Api Gateway también conectada simplificaría toda la comunicación en un solo punto. De esta manera el Frontend en vez de llamar a un microservicio distinto dependiendo de sus necesidades, solo tendría que llamar a la Api Gateway la cual se preocupa de identificar y llamar al microservicio correspondiente.

1. Tests automáticos: mediante herramientas como Selenium o Katalon es posible testear la página web de manera automática. Realizas una grabación interactuando en la página web, navegando por los menús, interactuando con botones, introduciendo datos y luego estas herramientas son capaces de reproducir exactamente lo grabado y validar que el test se completa sin ningún error.