Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DO SOFTWARE

Diseño e implementación de una aplicación para el transporte ferroviario de viajeros

Estudante: Cristina Renda López
Dirección: Juan Raposo Santiago

A Coruña, setembro de 2020.

A mi familia y amigos

Agradecimientos

Gracias a mis padres y a mis abuelos, porque sin ellos esto no sería posible. A Álvaro, Menchuj, Sabe, Sara y Adri por su apoyo y su motivación. A Fran y Gorka por la idea original de PickyGo.

Resumen

Este trabajo consiste en el diseño e implementación de una aplicación SPA para la gestión del transporte ferroviario de viajeros. Esta aplicación está destinada a dos tipos de usuarios: los administradores de la compañía ferroviaria y los viajeros que harán uso de sus servicios. Las principales funcionalidades que ofrece son, por un lado, la búsqueda y venta de billetes de tren destinadas a los viajeros y, por otro lado, la gestión interna de la compañía ferroviaria.

La arquitectura elegida es una arquitectura cliente-servidor dividida en capas. El servidor es un backend formado por un servicio REST/JSON y una capa modelo, que implementa las funcionalidades de la aplicación y permite el acceso a la base de datos. El cliente es un frontend que se ejecuta en el navegador y permite la comunicación con el backend.

Los principales lenguajes utilizados son Java en el backend y JavaScript en el frontend. Para la implementación se han utilizado un conjunto de frameworks y librerías que agilizan la programación, simplifican el código y permiten un desarrollo dinámico, entre los que destacan Hibernate, Spring, React y Redux.

La metodología de desarrollo empleada es una metodología ágil basada en iteraciones. En cada iteración, se desarrollan una o varias funcionalidades completas, de forma que al final de la iteración esté disponible una versión funcional de la aplicación.

Abstract

This project consists in the design and implementation of an SPA application for the management of passenger rail transport. This application is intended for two types of users: administrators of the railway company and travelers who will use its services. The main functionalities it offers are, on the one hand, the search and sale of train tickets for passengers and, on the other hand, the internal management of the railway company.

The chosen architecture is a layered client-server architecture. The server is a backend made up of a REST / JSON service and a model layer, which implements the functionalities of the application and allows access to the database. The client is a frontend that runs in the browser and allows communication with the backend.

The main languages used are Java for the backend and JavaScript for the frontend. For the implementation, a set of frameworks and libraries have been used that sped up programming, simplified the code and allowed dynamic development, among which Hibernate, Spring, React and Redux stand out.

The development methodology used is an agile methodology based on iterations. In each iteration, one or more complete functionalities are developed, so that at the end of the iteration a functional version of the application is available.

Palabras clave:

- Aplicación web
- Funcionalidad
- React
- Redux
- Hibernate
- Spring
- Metodología ágil
- Sprint
- Usuario
- Viaje
- Tramo
- Búsqueda
- Venta
- Validación

Keywords:

- Web application
- Functionality
- React
- Redux
- Hibernate
- Spring
- Agile methodology
- Sprint
- User
- Trip
- Stretch
- Search
- Purchase
- Validation

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introdución

Este trabajo consiste en el diseño e implementación de un sitio web SPA de gestión del transporte ferroviario de viajeros. Esta aplicación web actúa de intermediaria entre la compañía ferroviaria y los viajeros y, además, permite la gestión interna de la propia compañía.

Hay dos tipos de usuarios: los administradores de la compañía ferroviaria y los viajeros que harán uso de sus servicios. Según el tipo de usuario que haga uso de la aplicación, tendrá acceso a distintas funcionalidades.

La aplicación permite que un usuario se registre, inicie y cierre sesión en la página, modifique los datos de su perfil y cambie su contraseña. Cualquier usuario, aunque no esté registrado en la aplicación, podrá buscar información sobre los horarios de circulación de los trenes. Los usuarios registrados como viajeros, además, podrán comprar un billete de tren, ver su historial de compras, ver la información de uno de sus billetes a partir del localizador, cambiar un billete y devolver un billete. Los usuarios registrados como administradores podrán realizar tareas relacionadas con la gestión de viajes, estaciones y líneas. Además, podrán obtener estadísticas relacionadas con las ventas y con los viajeros en función de distintos parámetros, como el período tempomral o las estaciones de origen y destino.

Para el desarrollo de la aplicación se ha empleado una metodología ágil, iterativa e incremental. Por lo tanto, el proyecto se ha dividido en una serie de iteraciones y se han implementado una o varias funcionalidades en cada una de estas iteraciones.

El diseño de la aplicación se ha hecho siguiendo una arquitectura en capas. El backend de la aplicación está formado por una capa de acceso a datos, una capa lógica de negocio y una capa servicios REST. Para su implementación en Java se han empleado tecnologías de apoyo como Hibernate y Spring. El frontend de la aplicación está formado por una capa de acceso a servicios y una capa IU. En su implementación, principalmente en JavaScript, destaca el uso de React y Redux.

1.1 Motivación

El transporte es un conjunto de procesos que tienen como finalidad el desplazamiento y la comunicación. Estos procesos no son sencillos, ya que hay muchos factores que influyen en ellos, como las condiciones meteorológicas, el estado del vehículo o el estado del tráfico.

En un mundo donde la sostenibilidad tiene cada vez más protagonismo, el transporte público juega un papel fundamental. Aunque tiene claras desventajas frente al vehículo privado, también presenta una serie de ventajas que se deben tener en cuenta, como la seguridad y la fiabilidad.

Desde el punto de vista del transporte público, ofrecer una experiencia positiva al usuario es importante para mantener el sistema en funcionamiento. Pero cualquier imprevisto puede causar, por ejemplo, un retraso y, en consecuencia, una mala experiencia.

Con las tecnologías vigentes hoy en día, es posible ofrecer al usuario información en tiempo real sobre la oferta de viajes, el estado del viaje, etc. También es posible la coordinación de forma precisa entre los distintos operadores del transporte. En resumen, hay muchas posibilidades para que las compañías encargadas del transporte puedan ofrecerle una mejor experiencia al usuario.

Debido a un interés personal por el ámbito del transporte y, en particular, por el trasporte público he decidido hacer una aplicación con esta temática. Además, por viajar frecuentemente en tren, y ocasionalmente en avión, hago uso constantemente de distintas aplicaciones web destinadas a la compra de billetes. La implementación de dichas aplicaciones siempre me ha producido curiosidad, porque se alejan un poco del típico ejemplo académico de gestión de un almacén de productos y, sin embargo, están también muy presentes en nuestro día a día.

Para la elección de las tecnologías, pensé en qué me convendría aprender y tenía dos opciones: aprender algunas nuevas o emplear las ya conocidas. Considerando que los fundamentos serían similares, decidí que me vendría mejor afianzar los conocimientos en un campo ya familiar, poniéndolos en práctica y aplicándolos a un dominio distinto. Por lo tanto, las tecnologías escogidas son las que se han utilizado en la asignatura de Programación Avanzada.

En cuanto a la metodología, he decido seguir una metodología ágil, además de por sus ventajas frente a las metodologías tradicionales, por el protagonismo creciente que éstas tienen en el entorno laboral.

1.2 Objetivos

Los objetivos del desarrollo de este proyecto han sido los siguientes:

- Afianzar los conocimientos con las tecnologías empleadas. Como se ha mencionado en el apartado anterior, las tecnologías empleadas ya eran conocidas; sin embargo, se

buscaba una mayor comprensión y familiaridad con las mismas.

- Hacer un buen diseño de la aplicación, que facilitase las tareas de implementación posteriores y que diese lugar a una aplicación que funcionase adecuadamente.

- Realizar una planificación adecuada y seguirla durante el desarrollo, de forma que cada funcionalidad estuviese terminada en el tiempo establecido al principio de cada iteración.

- Desarrollar de una aplicación funcional. Dentro del campo académico, se buscaba una aplicación que cumpliese las funcionalidades establecidas, siguiendo buenas prácticas tanto en el diseño como en el desarrollo de la misma.

1.3 Organización

A continuación se citan y describen brevemente los capítulos en los que se estructura el presente documento.

1.3.1 Fundamentos tecnológicos

Se justifica la elección de los elementos que se han utilizado en la elaboración del proyecto, como los lenguajes, frameworks, librerías, entorno de desarrollo, etc.

1.3.2 Metodología

Se exponen las ventajas de las metodologías ágiles y se explica la metodología seguida durante el desarrollo, justificando los motivos de su elección.

1.3.3 Análisis

Se definen los límites de la aplicación: qué hará la aplicación y quién la utilizará. También se explican las distintas fases en que se ha dividido la elaboración del proyecto.

1.3.4 Diseño

En este capítulo se explica la arquitectura de la aplicación, primero de forma global y posteriormente se entra en detalle en cada una de sus capas, explicando y justificando los patrones de diseño que se han seguido.

1.3.5 Implementación

Se explica cómo, partiendo del diseño de cada parte de la aplicación, se ha llevado a cabo la implementación de la misma, haciendo uso de distintas herramientas de apoyo.

1.3.6 Pruebas

Para validar la efectividad de la aplicación, se han realizado una serie de pruebas que se describen en este capítulo.

1.3.7 Planificación y evaluación de costes

En este capítulo se reflejan las distintas fases de elaboración del proyecto, las distintas actividades realizadas en cada fase y la evaluación de costes.

1.3.8 Principales funcionalidades de la aplicación

Se presentan las prestaciones más destacadas de la aplicación, de forma que quede claro qué hace la misma.

1.3.9 Conclusiones

Se revisan los objetivos planteados inicialmente y se contrastan con lo realizado. Se incluyen las lecciones aprendidas y las líneas futuras de la aplicación.

Capítulo 2

Fundamentos tecnológicos

En este capítulo se exponen y justifican los elementos que se han utilizado en el proyecto.

2.1 Hardware

Se ha utilizado un equipo personal con las siguientes características:

- Sistema operativo: Windows 10 Enterprise.
- Procesador: Intel Core i7-7700 3.60GHz
- Memoria RAM: 8GB
- Tipo de sistema: 64 bits

2.2 Lenguajes

A continuación se explica brevemente qué lenguajes se han utilizado en el proyecto.

2.2.1 Java

Java es un lenguaje de programación de propósito general. Se trata de un lenguaje basado en clases y orientado a objetos, compatible con Spring Boot para el desarrollo de aplicaciones web. Es el lenguaje principal elegido para el desarrollo de la lógica de negocio y los controladores de la aplicación. El principal motivo de su elección es la familiaridad con el mismo, así como con los frameworks y librerías que se mencionan más adelante.

2.2.2 JavaScript

JavaScript es un lenguaje interpretado orientado a objetos desarrollado por Netscape. Actualmente, es muy utilizado para páginas web. Se utiliza principalmente del lado del cliente,

implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas. JavaScript soporta construcción de objetos basada en prototipos. Su sintáxis básica es similar a Java. Se ha elegido JavaScript por su popularidad y por la facilidad de aprendizaje debido a sus similitudes sintácticas con Java.

2.2.3 HTML

Se trata de un sistema estandarizado para marcar la estructura de páginas web, lo que permite modificar su apariencia y funcionalidades con ayuda de CSS y JavaScript. Es el lenguaje utilizado para los componentes de la vista de la aplicación.

2.2.4 CSS

Es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado. Es muy utilizado para establecer el diseño visual de los documentos web, e interfaces de usuario escritas en HTML.

2.2.5 SQL

Es un lenguaje diseñado para administrar y recuperar información de sistemas de gestión de bases de datos relacionales. Se ha utilizado para la visualización, adición, modificación y borrado de datos en la aplicación.

2.3 Frameworks y librerías

A continuación se explica brevemente qué frameworks y librerías se han empleado.

2.3.1 JPA / Hibernate

JPA [1] es una API estándar de un mapeador objeto-relacional, que de forma interna hace uso de la API de JDBC. Proporciona anotaciones para mapear entidades a una base de datos relacional, haciendo este proceso muy sencillo. También proporciona una API para interactuar con la base de datos, que incluye operaciones CRUD, lenguaje de consultas, etc. La implementación de JPA elegida ha sido Hibernate.

Hibernate es un mapeador objeto/relacional, de código abierto, creado por un grupo de desarrolladores a finales de 2001. Se utiliza con el objetivo de simplificar el desarrollo de la capa de persistencia, evitando utilizar directamente JDBC. Para esto, permite utilizar objetos persistentes en lugar de manipular directamente datos de la BD. En general, ofrece las siguientes características:

- Asocia clases persistentes con tablas de una BD relacional, haciendo transparente la API de JDBC y el lenguaje SQL.
- Soporta relaciones entre objetos y herencia.
- No oculta el tipo de BD, que se sabe que es relacional, pero sí la BD concreta (Oracle, PostgreSQL, etc.).
- Implementa la API de persistencia de Java (JPA), y además ofrece una propia por si la primera no fuese suficiente.
- Ofrece un lenguaje de consultas, con semántica orientada a objetos y una sintaxis parecida a SQL.
- Aunque el principal objetivo no es manipular directamente los datos de la BD, si en algún caso fuera necesario permite lanzar consultas SQL.

2.3.2 Spring

Spring [2] es un framework de código abierto, creado por Rod Johnson, para el desarrollo de aplicaciones Java. Su objetivo es facilitar el desarrollo de aplicaciones Java EE, promoviendo buenas prácticas de diseño y de programación. Simplifica el uso de muchas de las APIs de Jakarta EE y ofrece alternativas a algunas de estas.

Es un framework modular, es decir, permite usar diferentes módulos sin comprometerse con el uso del resto, y tiene soporte tanto para la capa modelo como para la capa de interfaz web. Algunas de sus características más destacadas son:

- Contenedor de inyección de dependencias. Es el núcleo de Spring. Permite establecer las dependencias entre los objetos Java y se encarga de satisfacerlas, sin tener que ser hecho explícitamente.
- Gestión declarativa de transacciones. Ofrece una implementación del paradigma de la programación orientada a aspectos, con la cual permite que la gestión de transacciones sea transparente al programador.

Spring Data

Es una ibrería que permite implementar ágilmente DAOs (repositorio, en terminología de Spring) sobre diversas fuentes de datos. En particular, se ha utilizado Spring Data para JPA.

Spring Boot

Se utiliza para las capas del backend. Configura automáticamente las librerías (Spring Data, Hibernate, Spring MVC, etc.).

Spring Security

Se utiliza para el control de acceso a las distintas partes de la aplicación por los distintos tipos de usuarios. Incluye características [3] como:

- Soporte para autenticación y autorización.
- Protección contra ataques como clickjacking o cross site request forgery.
- Integración con la API de servlets.

2.3.3 React

Es una biblioteca JavaScript de código abierto diseñada para crear interfaces de usuario con el objetivo de facilitar el desarrollo de aplicaciones SPA. Sus principales características [4] son:

- Virtual DOM. React mantiene un virtual DOM propio, en lugar de confiar únicamente en el DOM del navegador. Esto permite a la biblioteca determinar qué partes del DOM han cambiado comparando los contenidos de la versión nueva y la almacenada en el virtual DOM. De esta forma, se determinará cómo actualizar de forma eficiente el DOM del navegador.
- Basado en componentes. Esto permite abordar el desarrollo de IUs complejas mediante la implementación de componentes pequeños. Además, algunos de estos componentes podrán reutilizarse dentro de la misma aplicación y potencialmente también entre aplicaciones.
- Ciclos de vida. Los componentes, a lo largo de su existencia, pasan por una serie de estados que se clasifican en tres etapas: inicialización, actualización y destrucción. El método render es fundamental en los ciclos de vida, y se llama cada vez que se actualiza el estado del componente, de forma que los cambios se reflejan en la interfaz de usuario.
- JSX. Es una extensión sintáctica a JavaScript que permite escribir expresiones (en XML) para describir un conjunto de elementos devueltos, típicamente, por un componente. Su sintaxis es parecida a HTML, lo que hace el código más legible.

React router

React router es una librería de enrutamiento, que permite navegar entre pantallas de la aplicación mediante enlaces y provocar programáticamente un cambio de pantalla. Cambia la URL que se muestra en el navegador para que sea consistente con la pantalla que se muestra en la aplicación.

React Intl

Es una librería de internacionalización, que proporciona componentes para internacionalizar mensajes, cantidades numéricas y fechas.

2.3.4 Redux

Redux [5] es una librería que permite gestionar el estado de una aplicación de forma centralizada y modular, empleando una arquitectura funcional. No está ligado a React y puede utilizarse con otros frameworks. En este caso, se ha utilizado react-redux.

El estado que sólo es relevante a un componente es almacenado y gestionado por ese componente. Sin embargo, la mayor parte del estado y su lógica de modificación se extrae de los componentes. Los componentes pueden recibir notificaciones cuando el estado cambia.

El store es un objeto de Redux que almacena el estado de la aplicación, y sus correspondientes propiedades.

Cada vez que se quiere modificar el estado, se envía un objeto acción a store. Las acciones contienen la propiedad type para indicar su tipo y pueden contener también propiedades adicionales para los datos.

Gracias a la utilización del Virtual DOM de React, cada vez que se modifica el estado de un componente, éste y los componentes que usa en su renderización se vuelven a renderizar. De esta forma, la IU siempre está sincronizada con los cambios de estado.

El reductor es una función encargada de producir un nuevo estado ante una acción, y se ejecuta cuando se invoca a store.dispatch(action). Devuelve el nuevo estado a partir del estado anterior y la acción.

El estado se trata como inmutable; es decir, el reductor no modifica el estado anterior, sino que devuelve un nuevo objeto. Gracias a esto, React realiza algunas optimizaciones. La más clara, es que un componente sólo se vuelve a renderizar si el valor de las nuevas propiedades que recibe o el estado son distintos a los de la última vez.

Los componentes pueden leer el estado con store.getState.

Hooks

Los hooks [6, 7] son funciones que permiten acceder a características de React (como el estado) desde componentes función. Los hooks permiten implementar la mayor parte de los componentes como funciones. Se utilizan para hacer frente a tres problemas habituales cuando los componentes se implementan como clases:

- Dificultad para reusar lógica con estado entre componentes.
- En componentes complejos la redefinición de los métodos de React.Component suele conducir a código difícil de entender.

- Las "clases" de JavaScript tienen características que pueden resultar confusas.

React proporciona un conjunto de Hooks predefinidos. En el desarrollo de la aplicación, se han utilizado los siguientes Hooks:

- useState permite añadir estado a un componente función. Declara una variable de estado que queda ligada a dicho componente. Recibe el valor inicial como parámetro y devuelve un array con dos elementos: el valor actual y una función para actualizar el valor. Si se utiliza esta función, el componente se volverá a renderizar.

- useSelector suscribe un componente al store. Recibe un selector como parámetro. En terminología Redux, un selector es una función pura que devuelve un valor a partir del estado almacenado en el store. Así, se permite ocultar la estructura interna del estado a los componentes, que solamente se conoce en los ficheros reducer y selectors. Si se refactoriza la estructura interna del estado, los componentes suscritos no se verán afectados.

- useEffect se emplea para ejecutar una función tras cada renderización del componente. Recibe dos argumentos: el efecto (obligatorio) y las dependencias (opcional). El efecto es una función que se ejecuta justo después de cada renderización del componente. Opcionalmente, puede devolver una función como resultado, denominada clean-up. Después de que el componente se monte por primera vez, se ejecuta el efecto. Cada vez que el componente se vuelve a renderizar, se ejecutan el clean-up y el efecto. Antes de que el componente se desmonte, se ejecuta el clean-up. Las dependencias hacen que el clean-up y el efecto se ejecuten de forma condicional: si los valores de las dependencias son distintos a los de la última vez que se ejecutó el efecto, se ejecutarán; en caso contrario, no.

2.4 Herramientas para el desarrollo

A continuación se describen brevemente otras herramientas utilizadas.

2.4.1 Eclipse

Eclipse es un entorno de desarrollo integrado multilenguaje con un espacio de trabajo y un sistema plug-in ampliable para personalizar el entorno. Ofrece las siguientes características:

- Editor de código adaptado al lenguaje en cuestión, en este caso Java.

- Compilación integrada.

- Debugging para poder analizar la ejecución del código instrucción por instrucción.

- Integración con Git.
- Refactorización.
- Ahorro de tiempo gracias a diversas opciones como el autocompletado o los atajos de teclado.

2.4.2 Visual Studio Code

Es un editor de código fuente desarrollado por Microsoft. Entre sus funcionalidades, destacan:

- Soporte para la depuración.
- Control integrado con Git.
- Resaltado de sintaxis.
- Autocompletado.
- Refactorización de código.

2.4.3 Apache Maven

En el proceso de desarrollo software existen muchas tareas rutinarias que hacen los desarrolladores, como la compilación del código, la ejecución de pruebas o el despliegue de la aplicación. La automatización de estas tareas permite acelerar el proceso, así como aprovechar mejor los recursos invertidos.

Apache Maven es una de las herramientas más utilizadas para la gestión del software en entornos Java. Promueve la comprensión y productividad en un proyecto aplicando patrones a todo el proceso de construcción del software. No es solo una herramienta de construcción, sino que es una herramienta de gestión y comprensión de proyectos. Ofrece servicios para gestionar la construcción del software, la documentación, las dependencias del proyecto, entre otras. Favorece la reutilización de componentes mediante el uso de arquetipos.

Maven utiliza un Project Object Model (POM) en formato XML para describir el proyecto, sus dependencias con otros módulos y componentes externos así como el orden de construcción de dichos elementos. Viene con objetivos predefinidos para realizar ciertas tareas comunes en todo el proyecto, como la compilación o el empaquetado.

2.4.4 Git

En un proyecto de desarrollo software es esencial el control del versionado del software. Sus principales ventajas son:

- Todos los archivos están etiquetados.
- Todos los artefactos construidos están controlados.
- Se facilita la realización de las tareas de gestión de errores.
- Se permite controlar los cambios en el código.

Además, en un entorno de trabajo en equipo como suele ser el caso, permite que varias personas trabajen sobre el mismo proyecto sin interferir en el trabajo de los demás. Git es un sistema de control de versiones distribuido desarrollado por Linus Torvalds. Es fácil de aprender y además tienen un rendimiento muy bueno, características gracias a las cuales es el gestor de versiones más empleado actualmente. Además, debido a su extendido uso, cuenta con una gran comunidad de desarrolladores, lo que permite su integración con otras herramientas. Además de las características ya mencionadas presentes en cualquier sistema de control de versiones, Git presenta las siguientes ventajas:

- Permite trabajar sin conexión a internet, ya que contempla el uso de repositorios locales y repositorios remotos.
- Permite el desarrollo no lineal, gracias a la posibilidad de creación de distintas ramas y a las herramientas para navegar entre ellas.

2.4.5 MySQL

Es un sistema de gestión de bases de datos relacional open source, considerada como la base de datos de código abierto más popular para entornos de desarrollo web gracias a su rendimiento, fiabilidad y facilidad de uso.

Capítulo 3

Metodología

La metodología de desarrollo establece un marco de trabajo que guía todo el proceso de construcción del proyecto. Esta elección es vital, ya que de ella depende la correcta consecución de los objetivos establecidos. Por lo tanto, es necesario realizar una decisión bien fundamentada.

3.1 Metodologías ágiles

Existen muchas metodologías desarrolladas para su aplicación en proyectos de desarrollo software. El estudio de estas metodologías es un área dentro de la ingeniería del software. El uso de las metodologías conocidas como metodologías ágiles se ha extendido mucho durante los últimos años. Estas se caracterizan por el uso de ciclos de vida iterativos que se suceden a lo largo del tiempo, para construir un producto funcional al final de cada ciclo. El objetivo de esta división en iteraciones es satisfacer los requisitos fundamentales del sistema en las primeras iteraciones, para dedicar las siguientes fases a añadir funcionalidades o refinar características.

Las principales ventajas de las metodologías ágiles son:

- La construcción de productos intermedios a lo largo de todo el desarrollo permite una mejor interacción con el cliente, que puede ver el estado del producto antes de su total finalización.
- Minimizar los riesgos y los costes no previstos, ya que se pueden detectar en una fase más temprana.
- Desarrollar el software en iteraciones permite realizar una mejor gestión del cambio y adaptarse mejor a los cambios de requisitos.
- La construcción por componentes permite una mayor modularidad y un menor acoplamiento en el software.

El Manifiesto Ágil es un documento redactado en 2001 por expertos en el campo que proponían un cambio conforme a los modelos tradicionales. Este documento propone cuatro valores, y sobre estos cuatro valores se sustenta la filosofía de las metodologías ágiles. Estos cuatro postulados son:

- Individuos e iteraciones prevalecen sobre procesos y herramientas.
- Software funcionando prevalece sobre documentación exhaustiva.
- Colaboración con el cliente prevalece sobre negociación contractual.
- Respuesta ante el cambio prevalece sobre seguir un plan.

Estos cuatro valores dan lugar a los siguientes doce principios, que definen el marco de trabajo de cualquier equipo ágil:

1. Satisfacer al cliente mediante la entrega temprana y continua de software con valor.
2. Aceptar bien los cambios, aun cuando estos ocurran en etapas tardías del desarrollo.
3. Entrega frecuente de software funcional.
4. Los responsables del negocio y los desarrolladores deben estar en constante contacto.
5. Motivación en el trabajo.
6. Conversación cara a cara como método de comunicación, dejando la documentación para casos especiales.
7. La forma para medir el progreso se basa en la cantidad de funcionalidades desarrolladas.
8. Desarrollo y trabajo a un ritmo constante de manera indefinida.
9. Atención continua a la excelencia técnica y al buen diseño.
10. Simplicidad, minimizar la cantidad de carga de trabajo.
11. Equipos auto-organizados.
12. Perfeccionar la efectividad del equipo según la experiencia adquirida con el tiempo.

Algunos ejemplos de metodologías ágiles comunes son XP (Extreme Programming), Scrum o Kanban.

3.2 Scrum

Scrum es un marco de trabajo definido por Ikujiro Nonaka y Takeuchi a principios de los 80. Su objetivo es el desarrollo ágil de software mediante la aplicación de un conjunto de buenas prácticas para trabajar en equipo de forma colaborativa y obtener el mejor resultado posible. Sus principales características son:

- Estrategia de desarrollo incremental, en lugar de la planificación y ejecución completa del producto.
- Basar la calidad del resultado en el conocimiento de las personas más que en la calidad de los procesos empleados.
- Solapar las diferentes fases del desarrollo en lugar de realizar una tras otra en un ciclo secuencial.

Las historias de usuario son descripciones breves y concisas sobre una funcionalidad del sistema, descritas desde la perspectiva del usuario que empleará dicha característica. Generalmente, siguen el siguiente formato:

Como [rol] quiero [funcionalidad]

Las historias de usuario forman el Product Backlog, que es una lista que contiene todo lo que se conoce que es necesario para el proyecto y es la única fuente de requisitos del producto. En ella se establecen todas las funcionalidades, cambios y mejoras que deben ser hechos en un futuro. Es un documento que nunca está completo, evoluciona y cambia a la vez que lo hace el producto.

El sprint es un concepto fundamental en scrum. Es un evento que dura entre una y cuatro semanas. Durante cada sprint se desarrolla un incremento del producto formado por varias historias de usuario, que constituye un elemento funcional y usable y que es potencialmente entregable al cliente. El desarrollo del producto se lleva a cabo en varios sprint, que se suceden a lo largo del tiempo. Es recomendable que la duración de estos sea siempre la misma y que no se introduzcan eventos no contemplados, para reducir la aparición de riesgos.

3.3 Adaptación de las metodologías ágiles al proyecto

Muchos de los principios de las metodologías ágiles no se pueden aplicar en este proyecto, debido a los siguientes condicionantes:

- Al ser un proyecto realizado por una única persona, no se pueden aplicar los principios relacionados con la comunicación entre miembros de trabajo, el trabajo en equipo, el

trabajo por pares, etc. Sin embargo, sí se ha aplicado el principio de sencillez, tratado de que todo el trabajo realizado sea suficientemente claro como para comprenderlo en otro momento del desarrollo y con los comentarios correspondientes en el caso de que no lo fuese.

- Al no existir un cliente propiamente dicho, tampoco tienen sentido muchos de los principios relacionados con el feedback que proporciona el cliente, la adaptación a los cambios en los requisitos, o, sencillamente, la satisfacción del cliente.

- El tiempo de dedicación al proyecto no ha sido constante a lo largo del desarrollo, debido a la realización en un largo período de tiempo con situaciones variadas en cuanto a la disponibilidad horaria.

Igualmente, se ha elegido una metodología ágil por la cantidad de ventajas que aporta sobre una metodología tradicional. No se ha elegido ninguna en concreto, sino que basándose en el conocimiento previo de varias de ellas, especialmente de Scrum, se ha intentado adoptar los principios más convenientes en cada caso:

- La simplicidad en el diseño se aplica en varias fases a lo largo del desarrollo. En primer lugar, el diseño se ha hecho por funcionalidades, de forma que cada una de ellas se ha implementado de forma independiente a las demás (por supuesto, reutilizando el código que fuese posible de las funcionalidades realizadas previamente). En segundo lugar, se ha tratado de que todo el código sea lo más sencillo posible, aplicando las refactorizaciones correspondientes en cada momento. Además, la interfaz de usuario es sencilla para facilitar la comprensión del programa.

- Eliminación de interrupciones y gestión del flujo. Aunque se podría abarcar la implementación de varias funcionalidades simultáneamente, se ha seguido estrictamente la práctica de abordar una funcionalidad completa en cada momento, y no comenzar con la siguiente hasta no haberla finalizado. El objetivo de esto es, por un lado, la organización en el trabajo y, por otro lado, evitar distracciones innecesarias, poniendo foco en cada momento en una única cuestión.

- Sprint. Igual que en el caso anterior, se podría realizar la planificación temporal del proyecto de muchas formas, debido a la gran cantidad de tiempo disponible para realizarlo. Sin embargo, por cuestiones de organización y compromiso, se han establecido sprints con una duración fija 3 semanas. En cada uno de ellos, se ha realizado una o varias funcionalidades completas de la aplicación.

- Historias de usuario. Para estimar el esfuerzo necesario para llevar a cabo una historia de usuario, se utilizan los puntos de historia. Estos puntos representan una medida

abstracta del esfuerzo necesario para completar una historia de usuario, que debe contemplar todo lo que puede afectar a la correcta consecución de esta: cantidad de trabajo a realizar, complejidad del trabajo, riesgo o incertidumbre, tec. A la hora de asignar estos puntos a una historia se utiliza una sucesión similar a la sucesión de Fibonacci. Esto se hace porque cuanto más avanza la sucesión más grande es el intervalo entre los números. De igual manera, cuanto más alto es el esfuerzo estimado para una historia, más grandes son el riesgo y la incertidumbre asociados a esta.

- Enfoque desde el punto de vista del cliente. Aunque no exista un cliente propiamente dicho, en todo momento el desarrollo se ha llevado a cabo teniendo en cuenta las necesidades que tendría un usuario final de la aplicación. Este concepto se ha aplicado para elegir qué funcionalidades serían necesarias, la prioridad entre funcionalidades, cómo sería la interfaz de usuario, qué posibles casos de error se podrían producir, etc.

Capítulo 4

Análisis

Como paso previo al diseño de la aplicación, es necesario definir los límites que ésta tendrá. Para ello, se han definido los actores del sistema (es decir, quiénes darán uso a la aplicación) y las historias de usuario (es decir, qué funcionalidades proporcionará la aplicación a los usuarios).

4.1 Actores del sistema

Hay dos tipos de actores:

- Viajeros. Son los usuarios generales de la aplicación, clientes de la compañía ferroviaria. Su principal objetivo es la compra de billetes de tren. Pueden estar registrados en la aplicación o no, y en consecuencia tendrán acceso a unas u otras funcionalidades.
- Administradores. Son los trabajadores de la empresa ferroviaria, cuyo objetivo principal es mantener la información de la página web acorde con el funcionamiento de la empresa. Deben estar registrados en la aplicación para tener acceso a las funcionalidades para las que están autorizados.

4.2 Historias de usuario

Los requisitos funcionales del proyecto se expresan como historias de usuario, que forman el Product Backlog. En la tabla 4.1 se recogen las historias de usuario de la aplicación y los puntos de historia que se le han asignado a cada una.

Las historias de usuario 1, 2 y 3 son muy similares desde el punto de vista de la implementación, ya que todas forman parte de la funcionalidad de búsqueda de viajes. Sin embargo, se trata de una funcionalidad compleja y por lo tanto se han mantenido las historias de usuario y se le han asignado valores altos de puntos de historia. Las historias de usuario 19, 20, 21,

ID	Historia de usuario	Puntos
1	Como viajero, quiero saber qué trenes salen de una estación en una fecha determinada.	10
2	Como viajero, quiero conocer qué trenes llegan a una estación en una fecha determinada.	10
3	Como viajero, quiero conocer qué trenes circulan entre dos estaciones en una fecha determinada.	20
4	Como viajero, quiero saber qué asientos hay disponibles en un viaje que elija.	20
5	Como viajero, quiero comprar un billete de tren para el viaje que elija.	40
6	Como viajero, quiero consultar mi historial de compras.	8
7	Como viajero, quiero consultar los detalles de un billete que he adquirido previamente.	5
8	Como viajero, quiero cambiar un billete que he adquirido previamente.	20
9	Como viajero, quiero devolver un billete que he adquirido previamente.	13
10	Como administrador, quiero conocer todas las líneas que gestiona mi compañía ferroviaria.	8
11	Como administrador, quiero consultar los detalles de una línea.	5
12	Como administrador, quiero crear una nueva estación.	8
13	Como administrador, quiero añadir una estación a una línea.	13
14	Como administrador, quiero conocer los detalles de un viaje.	8
15	Como administrador, quiero asignarle a un tren un trayecto para realizar con determinada frecuencia entre dos fechas determinadas, parando en determinadas estaciones a horas concretas.	40
16	Como administrador, quiero ampliar la capacidad de un tren en circulación, asignándole un coche adicional.	13
17	Como administrador, quiero obtener estadísticas sobre las ventas en un período determinado.	5
18	Como administrador, quiero obtener estadísticas sobre los viajeros que circulan entre dos estaciones en un período determinado.	8
19	Como viajero, quiero poder registrarme en la aplicación.	3
20	Como viajero o administrador, quiero iniciar sesión en la aplicación.	3
21	Como viajero o administrador, quiero actualizar mi perfil.	3
22	Como viajero o administrador, quiero cambiar mi contraseña.	3
23	Como administrador, quiero modificar la hora de las paradas de un tren en circulación.	8
		274

Tabla 4.1: Historias de usuario

22 se habían realizado previamente en otro proyecto, por lo que simplemente se ha realizado su integración con el proyecto actual. Por este motivo, sus valores de puntos de historia son mucho más pequeños que el resto.

4.3 Sprints

El proyecto se ha dividido en 10 sprints. Los criterios seguidos para realizar la agrupación de historias de usuario en los distintos sprints han sido:

- Implementar primero las funcionalidades esenciales de la aplicación, y aquéllas de las que dependan otras. No tendría sentido que un usuario pueda comprar un viaje si no puede antes consultar los viajes disponibles
- Realizar primero las funcionalidades más sencillas. De esta forma, a medida que se adquiere familiaridad con la aplicación, se abarcarán funcionalidades más complejas con mayor eficacia.
- Realizar en un mismo sprint funcionalidades relacionadas.
- Si varias funcionalidades relacionadas son demasiado complejas para realizarlas en un mismo sprint, se realizarán en sprints consecutivos. El caso más notorio es el de las historias de usuario 1, 2 y 3, que se realizan entre los sprints 1 y 2.
- Obtener un equilibrio entre los puntos de historia de cada sprint. Se han obtenido valores de entre 20 y 41 puntos de historia para cada sprint (excepto en el sprint 0, que cuenta con menos puntos de historia porque en él se realizan tareas adicionales). Se ha tenido en cuenta que, por motivos académicos, el esfuerzo dedicado a los primeros sprints sería menor.

4.3.1 Sprint 0

ID	Historia de usuario	Puntos
19	Como viajero, quiero poder registrarme en la aplicación.	3
20	Como viajero o administrador, quiero iniciar sesión en la aplicación.	3
21	Como viajero o administrador, quiero actualizar mi perfil.	3
22	Como viajero o administrador, quiero cambiar mi contraseña.	3
		12

Tabla 4.2: Historias de usuario del sprint 0

Este sprint se dedica a la configuración del entorno de desarrollo, el análisis, el diseño general y la creación de entidades, tablas y DAOs.

Además, se implementan las historias de usuario 19, 20, 21 y 22, que suman un total de 12 puntos de historia, como se ve en la tabla 4.2.

Como resultado se obtiene una primera versión funcional de la aplicación, donde los usuarios pueden registrarse, iniciar sesión, actualizar su perfil o cambiar su contraseña.

4.3.2 Sprint 1

ID	Historia de usuario	Puntos
1	Como viajero, quiero saber qué trenes salen de una estación en una fecha determinada.	10
2	Como viajero, quiero conocer qué trenes llegan a una estación en una fecha determinada.	10
		20

Tabla 4.3: Historias de usuario del sprint 1

En este sprint se implementan las historias de usuario 1 y 2, que suman un total de 20 puntos de historia, como se ve en la tabla 4.3.

El objetivo es obtener una primera versión del buscador de viajes. Además, para realizar las pruebas funcionales es necesario poblar la base de datos con los datos necesarios.

4.3.3 Sprint 2

ID	Historia de usuario	Puntos
3	Como viajero, quiero conocer qué trenes circulan entre dos estaciones en una fecha determinada.	20
		20

Tabla 4.4: Historias de usuario del sprint 2

En este sprint se implementa la historia de usuario 3, con un total de 20 puntos de historia, como se ve en la tabla 4.4.

Esta historia de usuario está muy ligada a las historias de usuario 1 y 2 del sprint anterior. Por este motivo, se refactoriza el código y se realizan las optimizaciones que se consideren correctas.

El resultado de este sprint es la versión final del buscador de viajes.

4.3.4 Sprint 3

ID	Historia de usuario	Puntos
10	Como administrador, quiero conocer todas las líneas que gestiona mi compañía ferroviaria.	8
11	Como administrador, quiero consultar los detalles de una línea.	5
14	Como administrador, quiero conocer los detalles de un viaje.	8
		21

Tabla 4.5: Historias de usuario del sprint 3

En este sprint se implementan historias de usuario 10, 11 y 14, que suman un total de 21 puntos de historia, como se ve en la tabla 4.5.

El objetivo es obtener las primeras funcionalidades que solamente podrán realizar los administradores de la compañía ferroviaria, configurando para ello los permisos necesarios.

Tras este sprint, los administradores de la compañía pueden consultar las líneas gestionadas por la compañía, los detalles de una línea y los detalles de un viaje.

4.3.5 Sprint 4

ID	Historia de usuario	Puntos
12	Como administrador, quiero crear una nueva estación.	8
13	Como administrador, quiero añadir una estación a una línea.	13
		21

Tabla 4.6: Historias de usuario del sprint 4

En este sprint se implementan las historias de usuario 12 y 13, que suman un total de 21 puntos de historia, como se ve en la tabla 4.6.

El objetivo es implementar las primeras funcionalidades donde el usuario introduce datos en la aplicación, por medio de formularios.

Tras este sprint, los administradores de la compañía pueden crear nuevas estaciones y añadir una estación a una línea existente.

4.3.6 Sprint 5

En este sprint se implementa la historia de usuario 15, con 40 puntos de historia, como se ve en la tabla 4.7.

Con esta historia de usuario, los administradores de la compañía ferroviaria pueden crear nuevos viajes, que aparecerán en el buscador implementado en los primeros sprints.

ID	Historia de usuario	Puntos
15	Como administrador, quiero asignarle a un tren un trayecto para realizar con determinada frecuencia entre dos fechas determinadas, parando en determinadas estaciones a horas concretas.	40
		40

Tabla 4.7: Historias de usuario del sprint 5

4.3.7 Sprint 6

ID	Historia de usuario	Puntos
4	Como viajero, quiero saber qué asientos hay disponibles en un viaje que elija.	20
		20

Tabla 4.8: Historias de usuario del sprint 6

En este sprint se implementa la historia de usuario 4, con 20 puntos de historia, como se ve en la tabla 4.8.

Como resultado, los usuarios registrados en la aplicación podrán ver qué asientos hay disponibles para determinado viaje. Este es el primer paso para que los usuarios puedan realizar compras en la aplicación.

4.3.8 Sprint 7

ID	Historia de usuario	Puntos
5	Como viajero, quiero comprar un billete de tren para el viaje que elija.	40
		40

Tabla 4.9: Historias de usuario del sprint 7

En este sprint se implementa la historia de usuario 5, con 40 puntos de historia, como se ve en la tabla 4.9.

El objetivo es que, tras elegir un asiento disponible gracias a la funcionalidad implementada en el sprint anterior, los usuarios registrados en la aplicación puedan comprar billetes de tren para el trayecto deseado.

ID	Historia de usuario	Puntos
6	Como viajero, quiero consultar mi historial de compras.	8
7	Como viajero, quiero consultar los detalles de un billete que he adquirido previamente.	5
16	Como administrador, quiero ampliar la capacidad de un tren en circulación, asignándole un coche adicional.	13
17	Como administrador, quiero obtener estadísticas sobre las ventas en un período determinado.	5
18	Como administrador, quiero obtener estadísticas sobre los viajeros que circulan entre dos estaciones en un período determinado.	8
		39

Tabla 4.10: Historias de usuario del sprint 8

4.3.9 Sprint 8

En este sprint se implementan las historias de usuario 6, 7, 16, 17 y 18, que suman un total de 39 puntos de historia, como se ve en la tabla 4.10.

El objetivo es, por un lado, que los usuarios que hayan comprado previamente un billete en la aplicación, puedan consultar los detalles de dicho billete, así como su historial de compras; por otro lado, completar las funcionalidades que se corresponden con los administradores de la compañía ferroviaria con las historias de usuario 16, 17 y 18.

4.3.10 Sprint 9

ID	Historia de usuario	Puntos
8	Como viajero, quiero cambiar un billete que he adquirido previamente.	20
9	Como viajero, quiero devolver un billete que he adquirido previamente.	13
23	Como administrador, quiero modificar la hora de las paradas de un tren en circulación.	8
		41

Tabla 4.11: Historias de usuario del sprint 9

En este sprint se implementan las historias de usuario 8, 9 y 23, que suman un total de 41 puntos de historia, como se ve en la tabla 4.11.

El objetivo de este sprint es que los viajeros puedan realizar cambios y devoluciones en sus billetes, dejando libres las plazas correspondientes para que puedan ser ocupados por otros viajeros. También que los administradores puedan modificar la hora de las paradas para

gestionar retrasos.

Con la finalización de este sprint, se considerará que la aplicación está terminada.

Capítulo 5

Diseño

En este capítulo se explican y justifican las decisiones de diseño que se han tomado, junto con las prácticas y patrones aplicados. En primer lugar, se explica la arquitectura global de la aplicación. A continuación, se explican el diseño del backend y por último el diseño del frontend.

5.1 Arquitectura global

Una single-page application (SPA) es un tipo de aplicación web en una sola página. Todas las pantallas se muestran en esta página, sin necesidad de recargar el navegador. Esto no significa que haya una sola vista; lo normal es que haya varias. Las vistas es lo que se correspondería a una pantalla en una aplicación de escritorio. Por lo tanto, al pasar de una vista a otra se produce el efecto de que hay varias páginas aunque realmente solo hay una. Su principal ventaja es que ofrecen una experiencia muy fluida para los usuarios.

La aplicación desarrollada es una aplicación SPA , que consta de:

- Un frontend JavaScript que se ejecuta en el navegador.
- Un backend al que accede el frontend, formado por un servicio REST/JSON y una capa modelo.

El diseño por capas es una técnica de diseño muy utilizada. Se basa en la idea de que una capa inferior proporciona servicio a una capa superior. Este servicio se define mediante un contrato de servicio. De esta forma, es posible independizar el software de ambas capas, ya que a la capa superior no le importa cómo se implementa el servicio. Las principales ventajas son:

- Facilidad en el mantenimiento, debido a que los cambios en la implementación de una capa no conllevan cambios en el resto de capas.

- Escalabilidad más sencilla de implementar, ya que cada capa se puede ejecutar en máquinas diferentes y, en consecuencia, se pueden aumentar los recursos solamente de aquellas capas que lo necesiten.
- Tolerancia a fallos.
- Flexibilidad, ya que los desarrolladores de una capa no necesitan conocer las tecnologías usadas en otras capas.
- Cada capa puede ser desarrollada en paralelo con el resto de capas.
- Reutilización: es posible reusar la misma capa modelo en distintas interfaces o distintas aplicaciones.

En cuanto a las desventajas del diseño por capas, las principales son:

- El software es más complejo.
- En ciertos casos, se podría optimizar la implementación de una capa si esta conociese el funcionamiento interno de otra capa. que lo necesiten.

En este caso, la aplicación se divide en las siguientes capas:

- Backend:
  - Capa modelo:
    - Capa de acceso a datos. Permite el acceso a la base de datos de la aplicación. Está formada por distintas entidades y un DAO por cada entidad.
    - Capa lógica de negocio. Implementa la lógica de negocio, utilizando la capa de acceso a datos para leer y escribir los datos que necesite.
  - Capa de servicios REST / JSON. Es una interfaz programática orientada a que otras aplicaciones remotas utilicen la funcionalidad de la capa modelo. El cliente invoca al servidor mediante el protocolo HTTP. Para enviar datos se utiliza el formato de representación JSON, "orientado a programas" y no a personas. Da soporte al estilo arquitectónico RESTful.
- Frontend:
  - Capa de acceso a servicios. Permite al frontend acceder a las funcionalidades del backend.
  - Capa IU. Permite a los usuarios utilizar la funcionalidad de la capa modelo.

En la figura 5.1 (página 29) se ilustra esta división en capas.

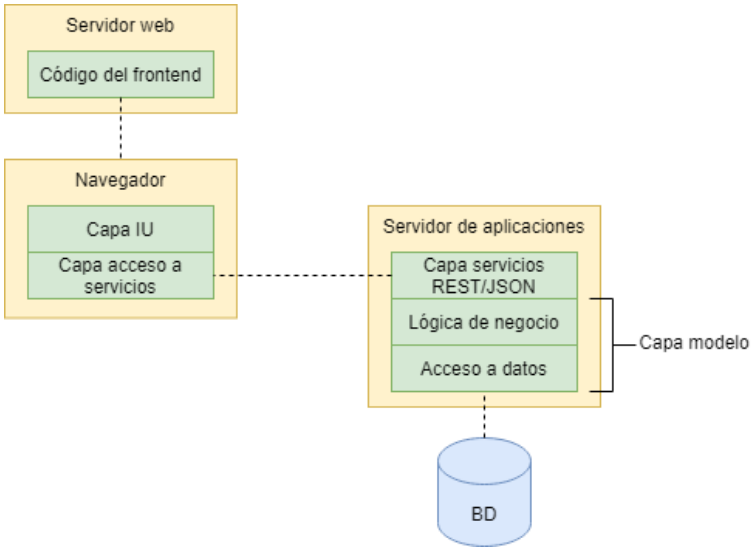

Figura 5.1: Arquitectura de la aplicación

5.2 Diseño del backend

En este apartado se explica y se justifica el diseño de cada una de las capas que forman el backend: la capa de acceso a datos, la capa lógica de negocio y la capa servicios REST. Además, en cada una, se explican las prácticas y patrones que se han aplicado al diseño.

También se recogen los diagramas que se han realizado para cada capa, siendo los principales el diagrama de entidades, el diagrama de servicios locales y el diagrama de controladores. La implementación de la aplicación se ha basado en estos diagramas. Sin embargo, ya que ésta se ha desarrollado por funcionalidades y de forma iterativa, se han realizado algunos cambios y adaptaciones durante el proceso.

5.2.1 Capa de acceso a datos

En el diagrama de entidades, en la figura 5.2 (página 31), se muestran las entidades existentes y las relaciones entre ellas. Este diagrama ha sido el más importante durante el desarrollo, porque en él se reflejan todas las entidades existentes, sus atributos y las relaciones entre ellas. Partiendo de él se han diseñado e implementado todas las funcionalidades de la aplicación. Cada una de las entidades se corresponde con una tabla en la base de datos.

Las principales entidades del modelo son:

- Ticket: Representa un billete de tren para un trayecto determinado, que ha adquirido un usuario de la aplicación, para él mismo o para otra persona.

- Purchase: Representa una compra realizada por un usuario. Un usuario, en una misma

compra, puede adquirir uno o varios billetes.

- User: Es un usuario de la aplicación, que puede ser viajero o administrador.
- Stretch: Representa un tramo entre dos estaciones de un viaje.
- Stop: Representa una parada en una estación a una hora determinada.
- Station: Representa una estación.
- Line: Representa una línea formada por varias estaciones.
- Trip: Representa un viaje realizado por un tren en una fecha determinada. El tren para en una serie de estaciones y, por lo tanto, un Trip está formado por uno o varios tramos.
- Train: Representa un tren.
- TrainType: Representa un tipo de tren.
- Car: Representa un coche. Cada tren está formado por uno o varios coches.
- CarType: Representa un tipo de coche, con una distribución de asientos determinada.
- Seat: Representa un asiento. Cada coche está formado por un conjunto de asientos.

El modelado de las relaciones N:M se ha realizado mediante tablas intermedias con sus correspondientes entidades y atributos. Estas entidades son:

- SeatStretch. Un tren, compuesto por muchos asientos, realiza varios viajes a lo largo del tiempo, y cada viaje está formado por varios tramos. Por lo tanto, un asiento realiza varios tramos y un tramo de un viaje es realizado por varios asientos. La entidad seatStretch representa la relación entre las entidades asiento y tramo; es decir, representa un asiento en un tramo concreto de un viaje determinado, que puede estar libre u ocupado. Por lo tanto, esta entidad es la que se emplea en la entidad Ticket para representar qué asientos y en qué tramos de qué viaje se han reservado.
- StationLine. Una línea de tren está formada por varias estaciones. Por cada estación, pueden pasar varias líneas. La entidad StationLine representa la relación entre las entidades Station y Line. Cuando un tren realiza una parada en una estación, lo hace porque está siguiendo una línea determinada para realizar su trayecto; por lo tanto, la entidad StationLine es la que se emplea en la entidad Stop para indicar dónde se hace la parada. También se utiliza en el servicio de administración para crear líneas y añadir estaciones a líneas ya existentes.

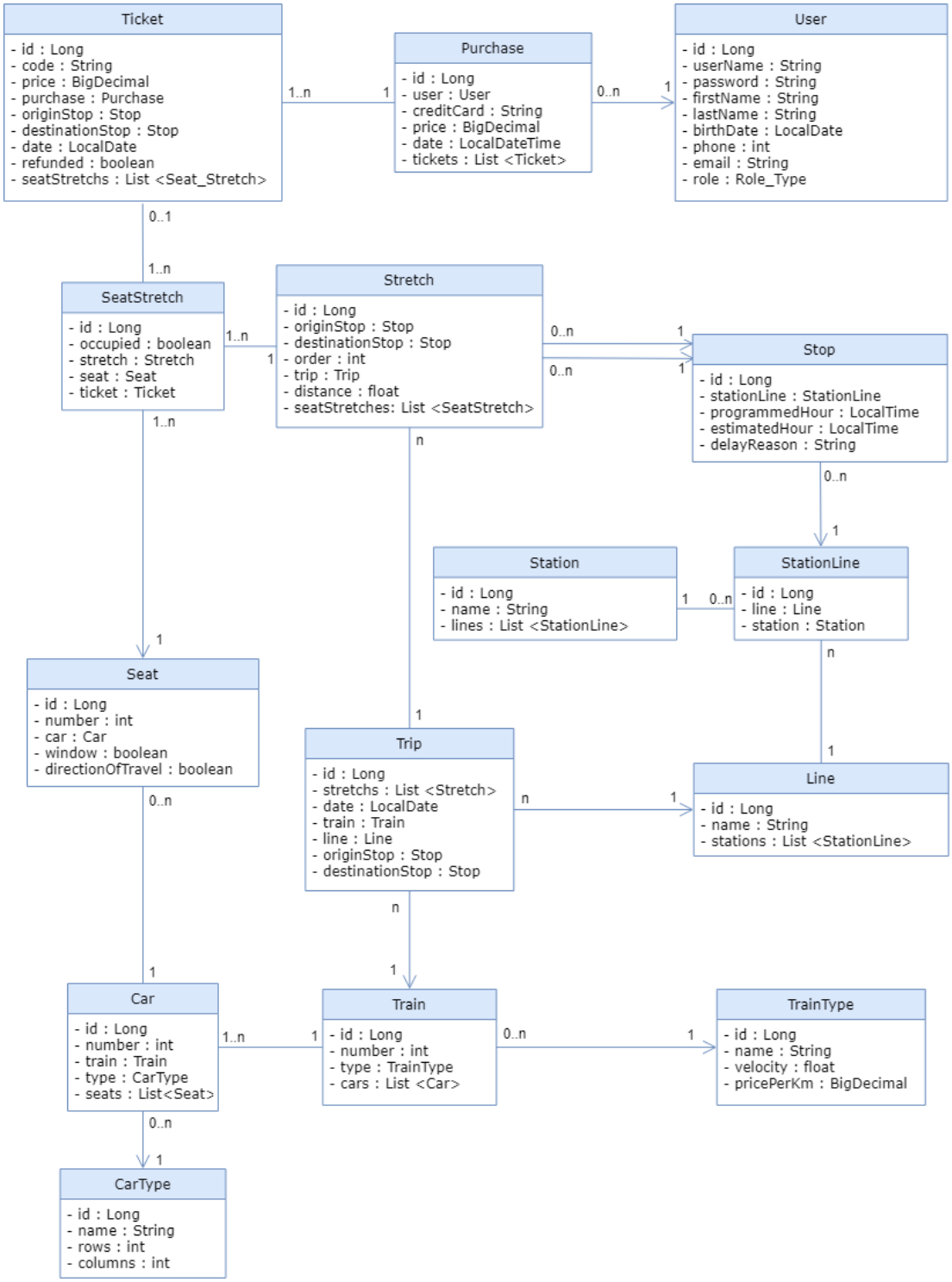

Figura 5.2: Diagrama de entidades

DAOs

Spring Data permite implementar ágilmente DAOs sobre diversas fuentes de datos. En particular, se ha utilizado Spring Data para JPA. Todos los DAOS extienden de PagingAndSortingRepository, que define:

- Operaciones CRUD.
- Algunas operaciones de búsqueda con capacidad de paginación y ordenamiento, como buscar por ID (findById) o añadir (save).

Además, es posible definir métodos de búsqueda usando convenciones de nombrado.

La paginación permite recuperar los datos en bloques, bajo demanda. Para emplear paginación, los métodos de búsqueda reciben un parámetro de tipo Pageable, que especifica el rango de resultados que se quieren recuperar de la BD y devuelven un valor de tipo Slice, que contiene los datos solicitados y una indicación de si hay más.

A modo de ejemplo, en la figura 5.3 (página 33) se incluye el diagrama de StationDao, en el que se reflejan los métodos disponibles en PagingAndSortingRepository (y en CrudRepository) y el método existsByName, definido mediante convención de nombrado.

Desnormalización

En la entidad Trip, se han añadido los atributos originStop y destinationStop, que representan las paradas de origen y destino de un viaje. Esta información se podría obtener a partir de la lista de tramos; sin embargo, se ha incluido en la propia entidad para optimizar operaciones, ya que se ha considerado que las paradas de origen y destino de un viaje (en ambos casos, se incluye tanto la estación como la hora) son información fundamental para identificar el viaje, especialmente desde el punto de vista del usuario.

Por motivos similares, en la entidad Ticket se han añadido los atributos originStop y destinationStop. En este caso, también se ha añadido el atributo date, que representa la fecha del viaje con el que se corresponde el billete.

En ambos casos, las relaciones que generan estos nuevos atributos no se han incluido en el diagrama de entidades, ya que no serían estrictamente necesarias y solo se han indicado con las finalidades descritas.

5.2.2 Capa lógica de negocio

Esta capa ofrece una API que permite invocar de forma sencilla cada caso de uso y oculta sus detalles de implementación. De esta forma, se independiza del resto de capas. Cada una de las funcionalidades que ofrece esta capa se corresponde, normalmente, con una funcionalidad concreta que desea ejecutar el usuario de la aplicación.

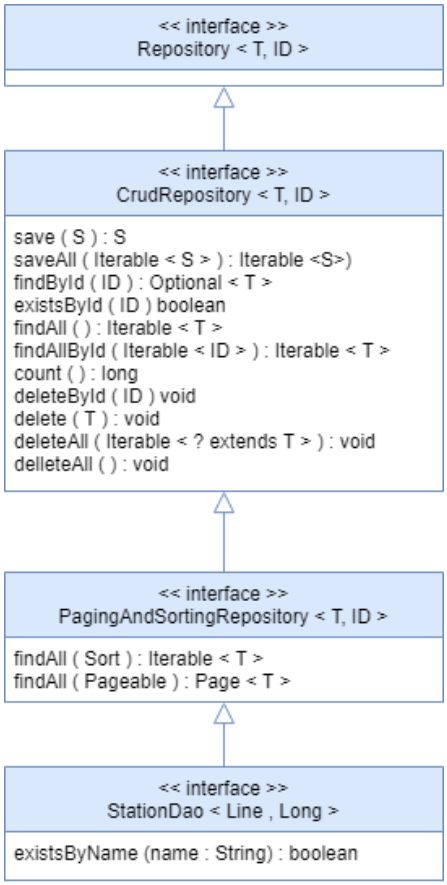

Figura 5.3: Diagrama de StationDao

Los casos de uso se han agrupado según qué tipo de usuarios de la aplicación los van a invocar. Esta agrupación ha dado lugar a cuatro servicios principales y un servicio auxiliar, como se ilustra en la figura 5.4 (página 34) y la figura 5.5 (página 35).

- Servicio de administración (AdministrationService). En él se incluyen todas las funcionalidades que realizarán los administradores de la compañía ferroviaria, como la gestión de líneas y estaciones, la creación de nuevos viajes o la obtención de estadísticas.
- Servicio de información (InformationService). En él se incluyen las funcionalidades a las que puede acceder cualquier usuario de la aplicación, relacionadas con la búsqueda de horarios de los distintos trenes.
- Servicio de ventas (TicketService). En él se incluyen las funcionalidades relacionadas con la venta de billetes, como la venta propiamente dicha, la consulta del historial de compras, y los cambios y devoluciones.
- Servicio de usuarios (UserService). En él se incluyen las operaciones relacionadas con

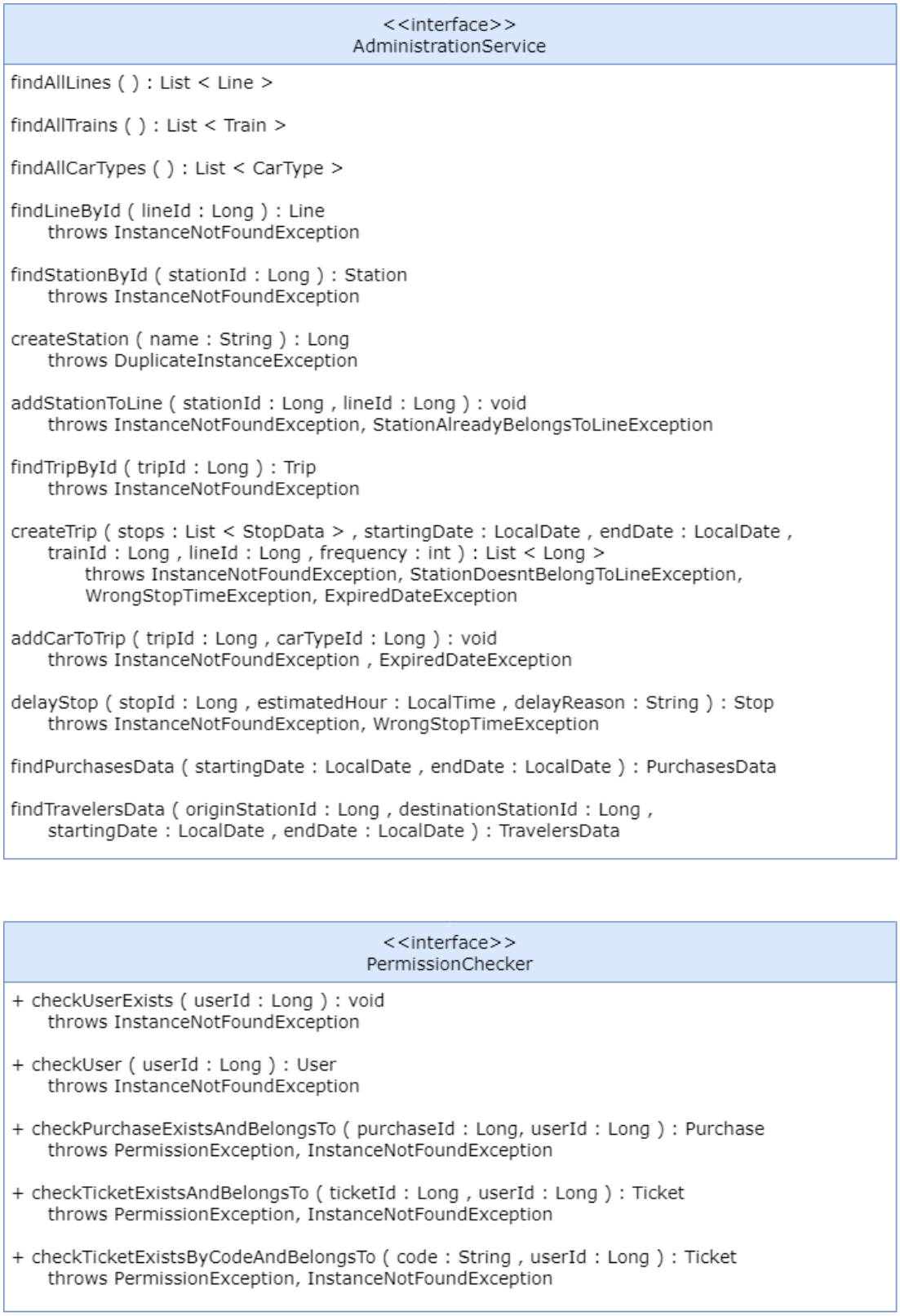

Figura 5.4: Diagrama del servicio de administración y el servicio auxiliar para la comprobación de permisos

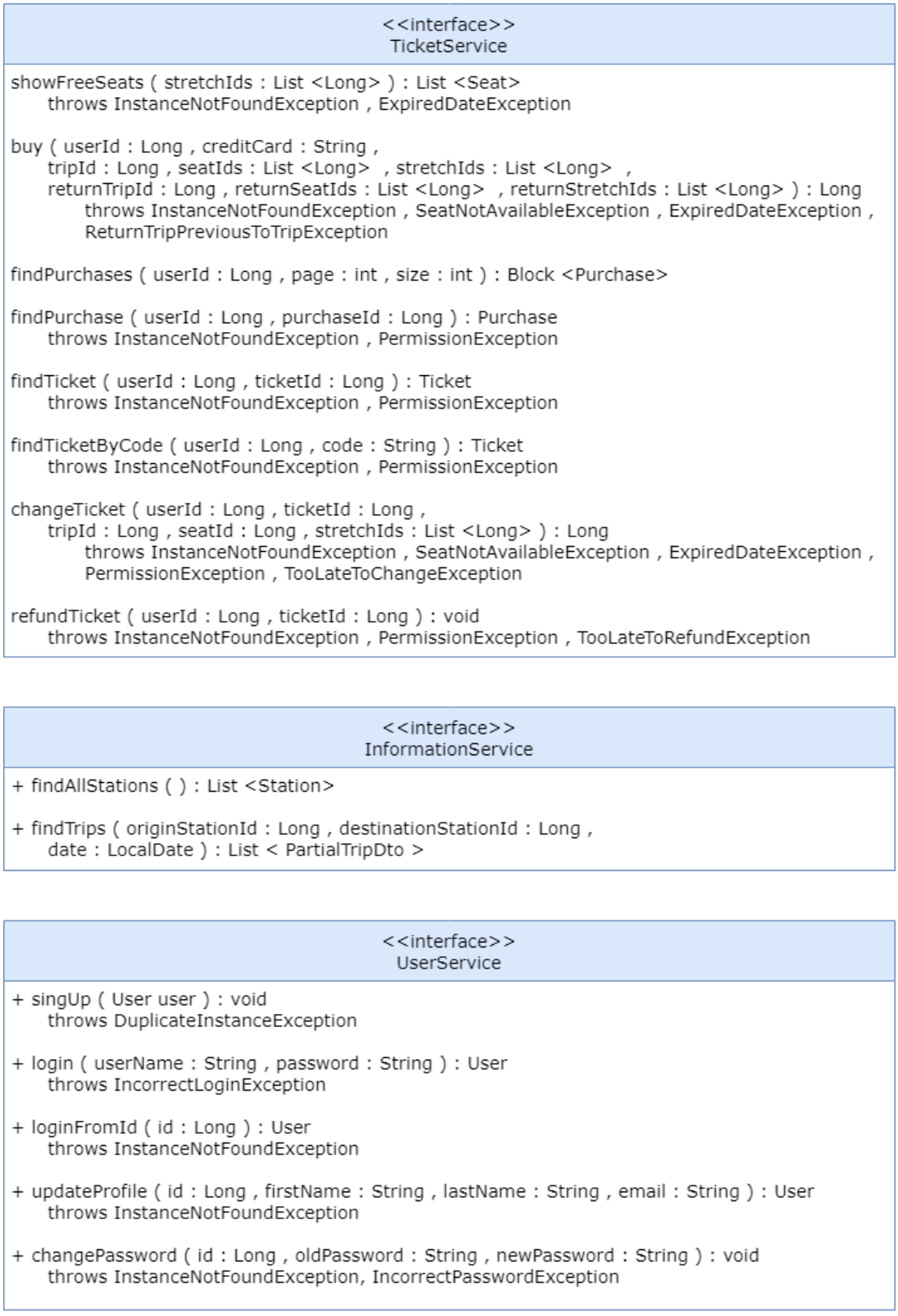

Figura 5.5: Diagrama de los servicios de ventas, información y usuario

el registro de usuarios, inicios de sesión en la aplicación, etc.

- PermissionChecker. Es un servicio interno que no se expone a la capa superior. Se encarga de la verificación de los permisos de acceso. Por ejemplo, cuando un usuario consulta la información de un billete, este servicio comprueba que dicho billete perteneza a ese usuario.

En cada uno de estos servicios se aplica el patrón de diseño Fachada, por lo que se define una interfaz y una clase que implementa las funcionalidades correspondientes.

A modo de ejemplo, en la figura 5.6 (página 36) se muestra el servicio de información. En la interfaz InformationService se definen dos métodos: findAllStations ("buscar todas las estaciones") y findTrips ("buscar viajes"). Estos métodos se implementan en la clase InformationServiceImpl, que necesita hacer uso de dos DAOs: el StationDao y el StretchDao.

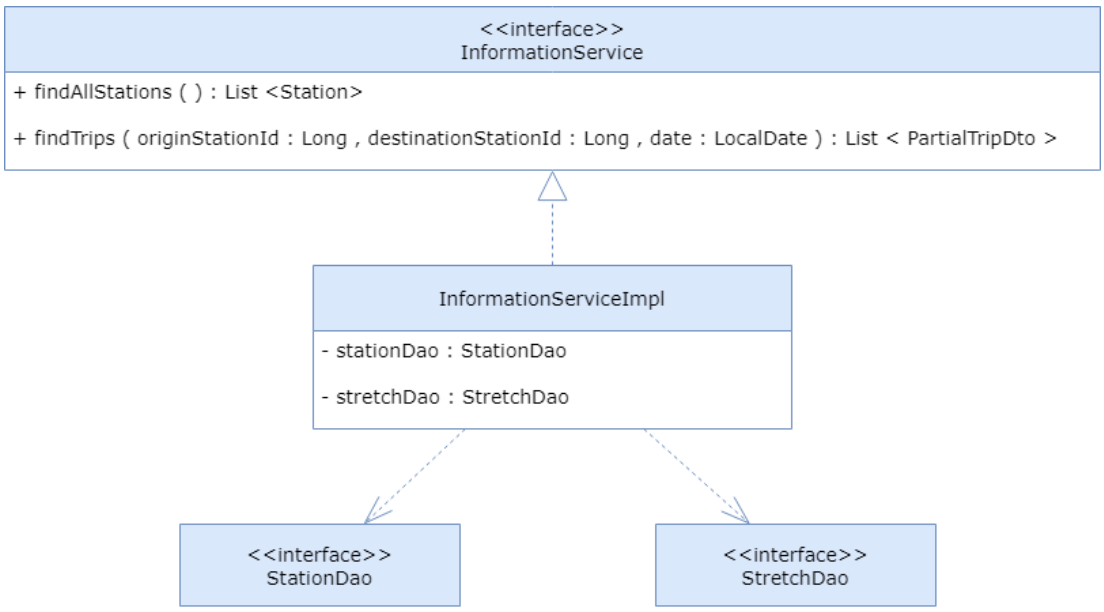

Figura 5.6: Diagrama del servicio de información

Clases adicionales

Se han utilizado algunas clases adicionales para facilitar la implementación de los servicios, que se comentan a continuación:

- Block almacena una parte de la lista de los elementos resultado, según el tamaño previamente indicado, y contiene un campo que indica si existen aún más elementos. Se ha utilizado para la paginación en el caso de mostrar el historial de compras de un usuario.

- StopData contiene los datos básicos que definen una parada; es decir, dónde (StationLine) y cuándo (date) se realiza dicha parada. Esta clase se utiliza en la implementación del método createTrip del servicio de administración. El usuario introducirá los datos mencionados para definir cada una de las paradas del viaje que desea crear.

- PartialTripDto y PartialTripConversor. La entidad Trip representa un viaje desde su origen hasta su destino. Sin embargo, un viajero puede desear realizar un trayecto entre dos puntos intermedios de un Trip. En este caso, no le interesará tanto la información de origen y destino del Trip y le interesará más la información de origen y destino de su trayecto concreto. Por este motivo, esta clase es el tipo devuelto por el método findTrips (buscar viajes) del servicio de información. La clase PartialTripCovnersor permite convertir los viajes devueltos por los DAOs al realizar una búsqueda en PartialTripDto.

- PurchasesData y TraveleresData se utilizan para devolver los datos necesarios para la elaboración de estadísticas de ventas y de viajeros, respectivamente, en los métodos findPurchasesData y findTravelersData del servicio de administración.

Inyección de dependencias

La implementación de los servicios locales necesita hacer uso de los DAOs necesarios y el servicio interno PermissionChecker.

Bajo el paradigma de la inyección de dependencias, existe un contendor de objetos que crea los objetos y les inyecta las referencias a los objetos de los que dependen. En Spring, a los objetos gestionados por su contenedor se les llama beans.

Excepciones

A continuación se explican las excepciones que se han contemplado en la lógica de negocio de la aplicación.

- DuplicateInstanceException. Se utiliza cuando se intenta crear una entidad con una clave única repetida.

- ExpiredDateException. Se emplea cuando una fecha ya ha pasado. Por ejemplo, en el servicio de ventas cuando se intenta comprar un billete de tren para un tren que está en circulación o que ya ha circulado o, en el servicio de administración, cuando se intenta crear un viaje para una fecha pasada.

- IncorrectLoginException. Se utiliza para indicar que los credenciales con los que un usuario intenta iniciar sesión en la aplicación no son correctos.

- IncorrectPasswordException. Se utiliza cuando un usuario intenta cambiar su contraseña, cuando no ha introducido correctamente su contraseña anterior.
- InstanceNotFoundException. Se utiliza para indicar que no se ha encontrado la instancia deseada.
- PermissionException. Se utiliza para indicar que se está intentando realizar una operación para la que no se tienen los permisos suficientes. Por ejemplo, ver los detalles de un billete de otro usuario.
- ReturnTripPreviousToTripException. Se utiliza cuando se intenta comprar un viaje de ida y vuelta, siendo el trayecto de vuelta anterior al de la ida.
- SeatNotAvailableException. Se emplea para indicar que un asiento no está disponible en el caso de querer comprar un billete para ese asiento concreto.
- StationAlreadyBelongsToLineException. Se emplea para indicar que una estación ya pertenece a una línea. Se utiliza en el servicio de administración cuando se intenta añadir una estación a una línea a la que ya pertenece.
- StationDoesntBelongToLineException. Se emplea para indicar que una estación no pertenece a una línea. Se utiliza al crear un viaje en el servicio de administración.
- WrongStopTimeException. Se emplea en el servicio de administración, para indicar que hay un error introduciendo los tiempos de las paradas durante la creación de un viaje o al retrasar la hora de una parada.
- TooLateToChangeException. Se utiliza para indicar que es demasiado tarde para cambiar un billete, cuando quedan menos de 30 minutos para la salida del tren de la estación de origen del trayecto.
- TooLateToRefundException. Se utiliza para indicar que es demasiado tarde para devolver un billete, cuando quedan menos de 2 horas para la salida del tren de la estación de origen del trayecto.

Paginación

Se ha empleado paginación para mostrar el historial de compras del usuario, ya que éste podría contener gran cantidad de registros. Normalmente el usuario querrá acceder a los más recientes, por lo que las compras están ordenadas por fecha. De esta forma, se evita recuperar excesiva cantidad de información de la base de datos. Se podría utilizar paginación en otras funcionalidades de la aplicación. Sin embargo, se ha descartado.

- En el caso de mostrar todas las líneas existentes o mostrar todas las estaciones, se ha considerado que el usuario de la aplicación deseará tener la información completa de las líneas o estaciones existentes y será más cómodo para él visualizarlas todas en la misma pantalla que pasar de una página a otra hasta encontrar la línea o estación deseada.
- Para mostrar el resultado de la búsqueda de viajes, también se ha considerado la opción de la paginación. Sin embargo, se ha considerado que no serán demasiados los viajes que cumplan los criterios seleccionados por el usuario, teniendo en cuenta que tendrá que elegir, al menos, la fecha del viaje y la estación de origen o destino del mismo. Además, podría ser confuso para el usuario el hecho de mostrar en pantalla solo una parte de los viajes de esa fecha. Por último, el usuario podría desear hacer una comparación entre varios viajes antes de elegir uno y, en este caso, es mejor mostrarlos en la misma página.
- Al mostrar los asientos disponibles de un tren, igual que en el caso anterior, se ha considerado importante que el usuario cuente con toda la información en la misma pantalla para facilitar su toma de decisiones.

5.2.3 Capa servicios REST

Se ha contemplado un controlador por cada uno de los cuatro servicios principales de la capa lógica de negocio.

Cada controlador recibe las peticiones HTTP procedentes del frontend y delega en algún método del servicio local asociado. REST (Representational State Transfer) es un estilo arquitectónico propuesto por Roy Fielding en el año 2000. Su objetivo es construir aplicaciones distribuidas inspirándose en las características de la Web, la aplicación distribuida más exitosa. HTTP es el protocolo cliente/servidor utilizado en la Web, que puede transferir cualquier contenido textual mediante un esquema de petición/respuesta.

Para el diseño de los controladores, se ha tenido en cuenta que las URLs deben ser únicas y globales para cada recurso. Se ha hecho un uso consistente de GET, PUT, POST y DELETE. También se ha hecho un uso consistente de los códigos de respuesta HTTP. Para elegir los códigos de error, se ha tenido en cuenta si las excepciones son temporales o permanentes, y se ha seleccionado en cada caso el código que más se ajustase a la petición.

- AdministrationController. Se corresponde con el AdministrationService. Su diseño se recoge en la figura 5.7 (página 40). Las clases PurchasesData y TravelersData que se emplean en la capa lógica de negocio tienen los mismos atributos que sus DTOs equivalentes (PurchasesDataDto y TravelersDataDto). Podrían emplearse directamente, pero sería una solución incorrecta porque iría en contra de la filosofía del diseño por capas, sería poco consistente y podría dar problemas si se realizase alguna modificación en una de las dos capas.

Administration Controller					
Recurso	Método	Entrada	Salida	Código error	Excepción asociada
/administration/lines	GET		List<LineSummaryDto>		
/administration/trains	GET		List<TrainDto>		
/administration/carTypes	GET		List<CarTypeDto>		
/administration/lines/{id}	GET	id [path]	LineDto	NOT_FOUND	InstanceNotFoundException (temporal)
/administration/stations/{id}	GET	id [path]	StationDto	NOT_FOUND	InstanceNotFoundException (temporal)
/administration/stations	POST	CreateStationParamsDto [params]	IdDto	BAD_REQUEST	DuplicateInstanceException (permanente)
/administration/lines/{lineId}/addStation/{stationId}	POST	lineId [path] stationId [path]		BAD_REQUEST	StationAlreadyBelongsToLineException (permanente)
				NOT_FOUND	InstanceNotFoundException (temporal)
/administration/trips/{id}	GET	id [path]	TripDto	NOT_FOUND	InstanceNotFoundException (temporal)
/administration/trips	POST	CreateTripParamsDto [params]	IdDto	NOT_FOUND	StationDoesntBelongToLineException (temporal)
				BAD_REQUEST	WrongStopTimeException (permanente)
				BAD_REQUEST	ExpiredDateException (permanente)
				NOT_FOUND	InstanceNotFoundException (temporal)
/administration/trips/{tripId}/addCar/{carTypeId}	POST	tripId [path] carTypeId [path]		NOT_FOUND	InstanceNotFoundException (temporal)
				BAD_REQUEST	ExpiredDateException (permanente)
/administration/stops/{id}	PUT	id [path] DelayStopParamsDto [params]	StopDto	NOT_FOUND	InstanceNotFoundException (temporal)
				BAD_REQUEST	WrongStopTimeException (permanente)

Figura 5.7: Diagrama del controlador de administración

- TicketController. Se corresponde con el TicketService. Su diseño se recoge en la figura 5.8 (página 41). Para mostrar los asientos libres, se envía una petición POST a /ticket/seats. Al ser una operación de búsqueda, podría hacerse con una petición GET. En ese caso, habría que pasar como parámetros del path la lista de identificadores de todos los tramos del viaje para el que se buscan asientos. Se ha considerado que esta solución sería poco elegante, por lo que finalmente se ha optado por la petición POST y pasar los identificadores en el cuerpo de la petición. Para devolver un billete, no se ha utilizado DELETE porque el recurso no se borra de la BD; simplemente se marca como devuelto.

Ticket Controller					
Recurso	Método	Entrada	Salida	Código error	Excepción asociada
/ticket/seats	POST	showFreeSeatsParamsDto [body]	List<SeatDto>	NOT_FOUND	InstanceNotFoundException (temporal)
				GONE	ExpiredDateException (permanente)
/ticket/purchases	POST	userId [jwt] buyParamsDto [body]	IdDto	NOT_FOUND	InstanceNotFoundException (temporal)
				NOT_FOUND	SeatNotAvailableException (temporal)
				GONE	ExpiredDateException (permanente)
				BAD_REQUEST	ReturnTripPreviousToTripException (permanente)
/ticket/purchases/{id}	GET	userId [jwt] id [path]	PurchaseDto	NOT_FOUND	InstanceNotFoundException (temporal)
				FORBIDDEN	PermissionException (permanente)
/ticket/purchases	GET	userId [jwt] page [param]	BlockDto <PurchaseSummaryDto>		
/ticket/tickets/{id}	GET	userId [jwt] id [path]	TicketDto	NOT_FOUND	InstanceNotFoundException (temporal)
				FORBIDDEN	PermissionException (permanente)
/ticket/tickets/ findByCode/{code}	GET	userId [jwt] code [path]	TicketDto	NOT_FOUND	InstanceNotFoundException (temporal)
				FORBIDDEN	PermissionException (permanente)
/ticket/tickets/change/{id}	POST	userId [jwt] id [path] ChangeParamsDto [body]	IdDto	NOT_FOUND	InstanceNotFoundException (temporal)
				FORBIDDEN	PermissionException (permanente)
				NOT_FOUND	SeatNotAvailableException (temporal)
				GONE	ExpiredDateException (permanente)
				GONE	TooLateToChangeException (permanente)
/ticket/tickets/refund/{id}	POST	userId [jwt] id [path]		NOT_FOUND	InstanceNotFoundException (temporal)
				FORBIDDEN	PermissionException (permanente)
				GONE	TooLateToRefundException (permanente)

Figura 5.8: Diagrama del controlador de ventas

- InformationController. Se corresponde con el InformationService. Su diseño se recoge en la figura 5.9 (página 42).
- UserController. Se corresponde con el UserService. Su diseño se recoge en la figura 5.9 (página 42).

Information Controller					
Recurso	Método	Entrada	Salida	Código error	Excepción asociada
/stations	GET		List<StationDto>		
/trips	GET	originStationId [param] destinationStationId [param] date [param]	List<PartialTripDto>	NOT_FOUND	InstanceNotFoundException (temporal)
				GONE	ExpiredDateException (permanente)

User Controller					
Recurso	Método	Entrada	Salida	Código error	Excepción asociada
/users/signUp	POST	UserDto [body]	AuthenticatedUserDto	BAD_REQUEST	DuplicateInstanceException (permanente)
/users/login	POST	LoginParamsDto [body]	AuthenticatedUserDto	NOT_FOUND	IncorrectLoginException (temporal)
/users/loginFromServiceToken	POST	userId [jwt]	AuthenticatedUserDto	NOT_FOUND	InstanceNotFoundException (temporal)
/users/{id}	PUT	userId [jwt] id [path] UserDto [body]	UserDto	NOT_FOUND	InstanceNotFoundException (temporal)
				FORBIDDEN	PermissionException (permanente)
/users/{id}/changePassword	POST	userId [jwt] id [path] ChangePasswordParamsDto [body]		FORBIDDEN	PermissionException (permanente)
				NOT_FOUND	InstanceNotFoundException (temporal)
				NOT_FOUND	IncorrectPasswordException (temporal)

Figura 5.9: Diagrama de los controladores de información y usuario

Patrón DTO

El patrón DTO tiene como objetivo crear un objeto plano con una serie de atributos que puedan ser enviados entre el servidor y el cliente en una sola invocación, aunque éstos procedan de distintas fuentes de datos. También permite ocultar inoformación que el usuario no requiere o que podría suponer un riesgo para la seguridad de la aplicación. Además de ser un objeto plano, un DTO debe ser de solo lectura y serializable (tanto el objeto como sus atributos).

En la capa servicios REST se han empleado distintos DTOs para la transferencia de información entre el backend y el frontend de la aplicación.

Algunas funcionalidades de la aplicación requieren que el usuario complete ciertos datos.

En los casos en que estos datos se envíen en el cuerpo de la petición, se ha utilizado un DTO que los contenga. Por ejemplo, para comprar un billete de tren es necesario indicar qué viaje, qué tramos y qué asientos desea el usuario, así como su tarjeta de crédito. BuyParamsDto es un DTO que contiene todos estos parámetros.

En otros casos, se han empleado DTOs para mostrarle información al usuario. Por ejemplo, TicketDto se emplea para mostrar toda la información relativa a un billete: localizador, precio, fecha del viaje, estación y hora de salida, estación y hora de llegada, número de asiento y si el billete ha sido cambiado o devuelto.

En ocasiones se necesita información más reducida. Por ejemplo, en una misma compra puede haber varios billetes. Cuando el usuario ve la información de la compra, verá una lista de billetes con información reducida de cada uno. Para mostrar esta información, se emplea TicketSummaryDto, que contiene el precio del billete, la fecha, las estaciones de origen y destino y si el billete ha sido cambiado o devuelto.

5.3 Diseño del frontend

En este apartado se explica y se justifica el diseño de cada una de las capas que forman el frontend: la capa de acceso a servicios y la capa IU.

5.3.1 Capa de acceso a servicios

La función de esta capa es gestionar las peticiones y respuestas HTTP que se envían el frontend y el backend de la aplicación.

Para mantener la misma estructura que en el backend, se emplea un fichero por cada controlador REST. En la figura 5.10 (página 44), se muestra a modo de ejemplo la firma de las funciones contenidas en el fichero ticketService, que se corresponde con el TicketController.

Para ilustrar el envío de peticiones y respuestas entre el backend y el frontend, en la figura 5.11 (página 45) y la figura 5.12 (página 46) se muestran las peticiones y respuestas generadas para mostrar los asientos disponibles y para comprar un billete de tren, respectivamente.

5.3.2 Capa IU

La capa IU se ha dividido en módulos, de forma que se facilita la gestión del código. Dentro de cada módulo, se ha aplicado el paradigma del desarrollo basado en componentes, que permite abordar el desarrollo de IUs complejas mediante la implementación de componentes pequeños.

Cada módulo gestiona una parte del estado de la aplicación, como se muestra en la figura 5.13 (página 46).

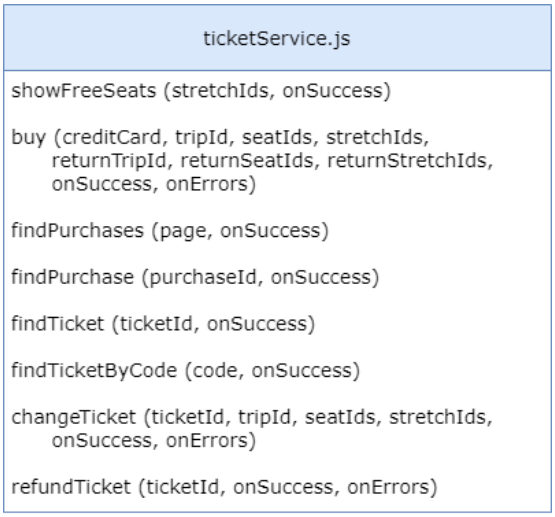

Figura 5.10: Funciones del fichero ticketService.js

Módulo app

Es el layout de la IU. Gestiona el estado de los errores y de la carga de páginas.

Módulo common

Contiene componentes reusables entre módulos.

Módulo administration

Gestiona el estado relacionado con las operaciones que realizarán los administradores de la compañía ferroviaria. Se corresponde con el administrationService del backend y el administrationController.

- lines: todas las líneas existentes.
- trains: todos los trenes existentes.
- cars: todos los tipos de coche existentes.
- line: línea seleccionada.
- station: estación seleccionada.
- trip: viaje seleccionado.
- tripStops: paradas de un viaje.
- purchasesData : datos obtenidos de la búsqueda de estadísticas de ventas.

Figura 5.11: Petición y respuesta al invocar el método showFreeSeats de la capa de acceso a servicios

- travelersData : datos obtenidos de la búsqueda de estadísticas de viajeros.

Módulo information

Gestiona el estado relacionado con las operaciones de búsqueda de viajes. Se corresponde con el informationService y el informationController.

- stations: todas las estaciones existentes.
- tripSearch: contiene el resultado de la búsqueda de viajes realizada por el usuario.

Módulo ticket

Gestiona el estado relacionado con la venta de billetes y la consulta de compras realizadas. Se corresponde con el ticketService y el ticketController.

- freeSeats: asientos libres en un viaje.
- lastPurchaseId: id de la última compra realizada por el usuario.
- tripId y returnTripId: viajes seleccionados por un usuario (de ida y de vuelta).
- stretchIds y returnStretchIds: tramos seleccionados por un usuario (de ida y de vuelta).
- seatIds y returnSeatIds: asientos seleccionados por un usuario (de ida y de vuelta).

Figura 5.12: Petición y respuesta al invocar el método buy de la capa de acceso a servicios

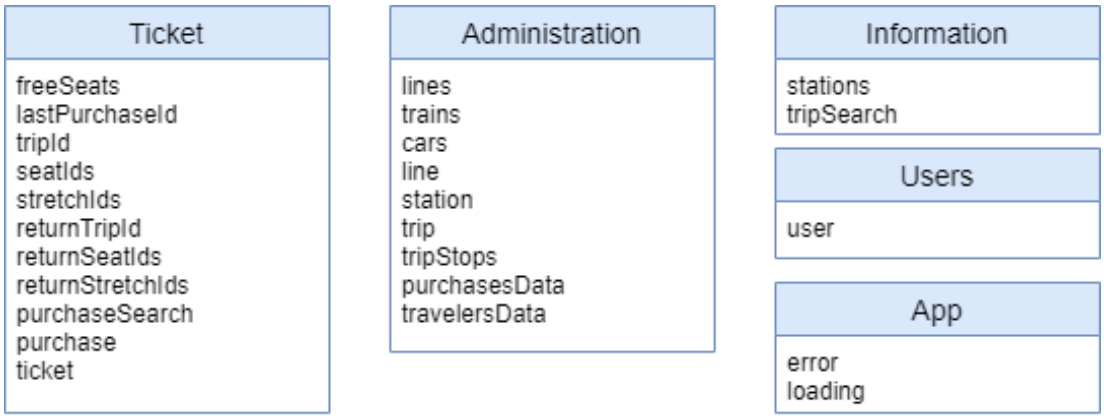

Figura 5.13: Módulos de la capa IU

- purchaseSearch: historial de compras de un usuario.
- purchase: compra seleccionada por un usuario.
- ticket: billete seleccionado por un usuario.

Módulo user

Gestiona el estado de los usuarios de la aplicación. Se corresponde con el userService y el userController.

- user: usuario que ha iniciado sesión en la aplicación.

Mockups

Para el diseño de esta capa, se han realizado mockups que simulen la interfaz de usuario. De esta forma, se permite una rápida y simple comprensión del funcionamiento de la aplicación y de cómo será ésta en su versión final. Desde el punto de vista del programador, ayuda a entender las funcionalidades necesarias, permite saber qué componentes se pueden reutilizar y aporta una idea del diseño de la interfaz de usuario.

Aunque no es el caso, la elaboración de mockups facilita la comunicación con el cliente, al permitir que éste se haga una idea de cómo será la aplicación y de su funcionamiento. De esta forma, le resultará sencillo expresar si es lo que busca o qué cambios desea realizar. Se incluyen algunos mockups a modo de ejemplo, donde se ha realizado una búsqueda de viajes y se muestran los resultados obtenidos.

En la figura 5.14 (página 47) se muestra el componente App, que es el que se visualiza en todo momento en la aplicación. Está formado por tres componentes: Header ("cabecera"), Body ("cuerpo") y Footer ("pie"). El componente Body es el encargado de mostrar el contenido principal de la aplicación en cada momento, en función del path. En el componente Header se incluye el acceso a las principales funcionalidades. En el componente Footer se incluye información general.

Figura 5.14: Mockup correspondiente a la búsqueda de viajes, componentes del módulo app

En la figura 5.15 (página 48) está el mismo mockup, pero se detallan los componentes propios del módulo correspondiente; en este caso, el módulo de información. La búsqueda de viajes se realiza en el componente FindTrips ("buscar viajes"), que está formado, a su vez,

por dos componentes StationSelector (“selector de estaciones”), un componente DateSelector (“selector de fecha”) y el botón de buscar. Los resultados se muestran en el componente FindTripsResult (“resultado de buscar viajes”). Este componente se encarga de evaluar qué tipo de resultados ha producido la búsqueda. Si ha producido un error, se mostrará dicho error. Si no se han obtenido resultados, no se mostrará nada. Si se han encontrado viajes, como es el caso del ejemplo, se mostrará el componente Trips (“viajes”). El componente Trips es una tabla cuyas columnas son los distintas características de un viaje y en cuyas filas aparecen los distintos viajes. Cada uno de estos viajes es un componente Trip (“viaje”).

Figura 5.15: Mockup correspondiente a la búsqueda de viajes, componentes propios del módulo

Capítulo 6

Implementación

En este capítulo se explica cómo, partiendo del diseño de la aplicación, se ha llevado a cabo la implementación de la misma con el apoyo de diversos frameworks y librerías. Aunque la aplicación se ha desarrollado por funcionalidades de forma iterativa, trabajando continuamente tanto en backend como en frontend, en este capítulo se ha decidido explicar ambas partes de forma separada, para mayor claridad. En primer lugar se explica la implementación del backend, a continuación la implementación del frontend y, por último, cuestiones comunes a la implementación de ambas partes.

6.1 Implementación del backend

En este apartado se habla de los detalles de implementación en cada capa del backend: la capa de acceso a datos, la capa lógica de negocio y la capa servicios REST.

6.1.1 Capa de acceso a datos

Para implementar la capa de acceso a datos se ha utilizado JPA/Hibernate y Spring Data. Se han utilizado dos BD distintas, de forma que una de ellas se utiliza únicamente para la ejecución de pruebas.

Entidades y relaciones

El mapping de entidades a tablas se ha realizado por medio de anotaciones sobre las entidades:

- @Entity indica que una clase es una entidad. Esta clase debe tener, entre otros requisitos, un constructor sin argumentos público o protegido y una clave primaria.
- @Id indica que un atributo de una clase es la clave primaria de la entidad.

- @GeneratedValue se utiliza conjuntamente con @ID para indicar que la clave es numérica y que se generará de forma automática.
- @Column se emplea para indicar que el nombre de la columna a la que se mapea el atributo de la clase es distinto al de dicho atributo. Esto sucede en el caso del "order", ya que es una palabra reservada en SQL.

Por otro lado, también se han utilizado anotaciones para modelar las relaciones entre entidades:

- @OneToMany en el lado con cardinalidad uno.
- @ManyToOne en el lado N. También se utiliza @JoinColumn sobre el método get correspondiente, para indicar el nombre de la columna que actúa como clave foránea.

Para facilitar la implementación de los servicios, en las propias entidades se han incluido algunos métodos de lógica de negocio. A continuación se comentan brevemente algunos de ellos.

Entidad Trip

- El método addStretch permite añadir un tramo a un viaje. También guarda el viaje como viaje correspondiente a dicho tramo. Este método se utiliza para hacer más sencilla la implementación del método createTrip en el servicio de administración. Además, facilita la implementación de las pruebas de los servicios de información y de ventas.
- Los métodos selectStretchesBefore y selectStretchesAfter permiten seleccionar únicamente aquellos tramos de un viaje que son anteriores o posteriores a un tramo determinado, respectivamente. Facilitan la implementación del método findTrips en el servicio de información, concretamente para aquellos casos en los que solo se ha seleccionado origen o destino del viaje, pero no ambos.
- El método selectStretchesBetweeen permite seleccionar aquellos tramos de un viaje que se encuentran entre dos tramos determinados. Facilita la implementación del método findTrips en el servicio de información en aquellos casos en los que se han seleccionado tanto el origen como el destino del viaje.

Entidad Stretch

- El método addSeatStretch añade un SeatStretch a facilita la implementación del constructor de la entidad SeatStretch, ya que en la misma creación de un SeatStretch, añade el SeatStretch creado a la lista de seatStretches del Stretch.

Entidad Station

- El método addStationLine añade una StationLine a la lista de stationLines de la línea.
- El método LineBelongsToStation devuelve un booleano que indica si la estación pertenece a la línea indicada. Se utiliza para devolver una excepción en el caso de intentar añadir una estación a una línea a la cual ya pertenece. Se utiliza en el constructor de StationLine para añadir la StationLine creada a la estación.

DAOs

Las interfaces de los DAOs no revelan ningún detalle de implementación. En la mayor parte de los casos, Spring Data implementa automáticamente los métodos de los DAOs en tiempo de ejecución.

Se han empleado convenciones de nombrado en StretchDao, para definir tres métodos que permiten:

- Buscar aquellos tramos cuya estación de origen y fecha sean las indicadas.
- Buscar aquellos tramos cuya estación de destino y fecha sean las indicadas.
- Buscar aquellos tramos cuyas estaciones de origen y destino y fecha sean las indicadas.

Estos métodos se utilizan en el servicio de información para recuperar los trayectos deseados por el usuario. También se han empleado convenciones de nombrado en StationDao. En este caso, existe un método existsByName que comprueba si existe una estación con determinado nombre. Este método se utiliza para comprobar, en el momento de crear una nueva estación, que el nombre no esté repetido. Por último, se han utilizado convenciones de nombrado en PurchaseDao y TicketDao, para generar varios métodos que permiten recuperar de BD, en función de distintos parámetros, información sobre las ventas y los viajeros. Estos métodos se emplean en el servicio de administración, en los métodos findPurchasesData y findTravelersData.

Se ha empleado paginación para mostrar el historial de compras de un usuario en el servicio de ventas. Por lo tanto, también se ha empleado paginación en el purchaseDao. La implementación usa la API de JDBC para informar a la BD de que no devuelva todas las filas que concuerdan con la búsqueda, sino sólo aquellas que están en el rango solicitado. Para que la paginación produzca una visualización de datos consistente entre páginas, es fundamental ordenar los datos devueltos por la consulta según algún criterio. En este caso, las compras realizadas por el usuario se han ordenado por fecha, de la más reciente a la más antigua, con el criterio de que son las más recientes las que tendrán un mayor interés para el usuario.

A continuación se muestra la interfaz de PurchaseDao, donde se han empleado paginación y convenciones de nombrado, para ilustrar las explicaciones anteriores.

```
1 public interface PurchaseDao extends
      PagingAndSortingRepository<Purchase, Long> {
2
3   Slice<Purchase> findByUserIdOrderByDateDesc(Long userId, Pageable
      pageable);
4
5   List<Purchase> findByDateBetween(LocalDateTime startingDate,
      LocalDateTime endDate);
6
7 }
```

6.1.2 Capa lógica de negocio

Esta capa lee los datos necesarios de la capa de acceso a datos, a través de los DAOs.

Inyección de dependencias

Este paradigma se aplica a los DAOs y a los servicios locales, mediante el uso de anotaciones de Spring:

- Los DAOs se implementan de forma automática con Spring Data y son beans, por lo que no es necesaria ninguna anotación.
- @Service se utiliza para anotar los servicios locales.
- @Autowired se utiliza para inyectar un bean que implemente la interfaz del atributo.

Durante el arranque del backend, el contenedor de objetos de Spring inspecciona las clases anotadas (en este caso, los DAOs y los servicios) y crea una instancia de cada una (bean). Para cada bean creado, inspecciona los atributos anotados con @Autowired y los inicializa con un bean de ese tipo.

Transaccionalidad

Spring permite usar un enfoque declarativo para implementar la transaccionalidad de los métodos de los servicios locales. Es posible especificar la transaccionalidad mediante la anotación @Transaccional.

Cuando se invoca un método anotado con @Transactional, antes de la ejecución del método:

- Si se hace en el contexto de una transacción, se engancha a ella.
- Si se hace fuera del contexto de una transacción, se crea una nueva.

Durante la ejecución del método, todas las operaciones que se invoquen de los DAOs o servicios se enganchan a la transacción.

Tras la ejecución del método, si éste no lanzó ninguna excepción o lanzó una excepción checked, la transacción termina con commit. Si se ha lanzado una excepción de runtime, la transacción termina con rollback.

Además, si un método sólo ejecuta operaciones de lectura, se puede especificar readOnly=true para permitir que el gestor de transacciones realice optimizaciones.

Se ha empleado la anotación @Transactional a nivel de clase en todos los servicios locales excepto en el de información. Esto quiere decir que la anotación afecta a todos los métodos de cada uno de esos servicios.

Se ha utilizado @Transactional(readOnly=true) en el servicio de información, en el PermissionChecker y en los métodos que solo utilizan operaciones de lectura, de forma que sobrescribe a la anotación a nivel de clase.

A continuación se muestra la anotación @Transactional(readOnly=true) utilizada a nivel de clase en el servicio de información:

```
@Service
@Transactional(readOnly = true)
public class InformationServiceImpl implements InformationService {
  ...
}
```

A continuacióin se muestra la anotación @Transactional utilizada a nivel de clase y @Transactional(readOnly=true) utilizada a nivel de método en el servicio de administración.

```
@Service
@Transactional
public class AdministrationServiceImpl implements
    AdministrationService {
  ...
  @Override
  @Transactional(readOnly = true)
  public List<Line> findAllLines() {
    ...
  }

  @Override
  public Long createStation(String name) throws
    DuplicateInstanceException {
    ...
  }
}
```

6.1.3 Capa servicios REST

Para la implementación ágil de la capa servicios REST, se ha empleado la librería spring-web. Internamente, spring-web utiliza la API de Servlets. Por cada servicio local, existe una clase controlador con tantos métodos como el servicio local.

Inyección de dependencias

En esta capa se aplica de nuevo el paradigma de inyección de dependencias con Spring. Se han utilizado las siguientes anotaciones:

- @RestController se utiliza para anotar cada controlador y que sus instancias sean gestionadas por le contenedor de objetos de Spring.
- @Autowired se utiliza para inyectar el servicio local asociado.

Mapping de peticiones HTTP

Se han empleado anotaciones para especificar cómo mapear las peticiones HTTP.

- @RequestMapping se utiliza a nivel de controlador para especificar que las peticiones dirigidas a un path que comience por el valor especificado serán procesadas por ese controlador.
- @GetMapping, @PostMapping, @PutMapping se utilizan a nivel de operación para especificar el resto del path y el método HTTP que causarán la ejecución de dicha operación.
- @PathVariable permite inyectar el valor de una variable del path (expresada entre “”) en un parámetro del método.
- @RequestParam permite inyectar el valor de un parámetro HTTP en un parámetro del método.
- @RequestBody permite inyectar la representación contenida en el cuerpo de la petición en el parámetro anotado.

En el siguiente fragmento de código, correspondiente al controlador de ventas, se ilustra la explicación anterior:

```
1 @RestController
2 @RequestMapping("/ticket")
3 public class TicketController {
4
5   ...
```

```
6
7    @PostMapping("/purchases")
8    public IdDto buy(@RequestAttribute Long userId, @Validated
       @RequestBody BuyParamsDto params)
9          throws InstanceNotFoundException, SeatNotAvailableException,
       ExpiredDateException {
10       ...
11   }
12
13   ...
14
15   @GetMapping("/purchases/{purchaseId}")
16   public PurchaseDto findPurchase(@RequestAttribute Long userId,
       @PathVariable Long purchaseId)
17         throws InstanceNotFoundException, PermissionException {
18
19       return toPurchaseDto(ticketService.findPurchase(userId,
       purchaseId));
20
21   }
22
23   ...
24
25 }
```

DTOs

Se han empleado DTOs para representar la información a intercambiar entre el cliente y el servidor. Spring-web utiliza Jackson para realizar la conversión automática entre JSON y DTOs, que incluye soporte para numerosos tipos Java.

Las fechas se han modelado como long para facilitar la portabilidad a la hora de convertirlas a objetos Date de JavaScript. Esto se debe a que el objeto Date contiene un número que representa los segundos desde el 1 de enero de 1970.

Validación básica de datos

En la capa servicios REST se realizan validaciones básicas de los datos recibidos en las peticiones HTTP utilizando el soporte de spring-web y la API estándar de validación de Java Beans Validation. De esta forma, la implementación de los casos de uso en la capa lógica de negocio no tiene que encargarse de este tipo de validaciones y se simplifica su implementación. Spring-web valida automáticamente la obligatoriedad y la conversión de los valores recibidos en las variables insertadas en el path y los parámetros HTTP.

En los casos en que se desean realizar validaciones más específicas, las reglas de validación se definen mediante la anotación @Validated en el DTO correspondiente. Por ejemplo, CreateStationParamsDto es el DTO que se recibe cuando un usuario invoca al método de crear una estación. Para crear una nueva estación, el usuario simplemente tiene que indicar cuál será su nombre ("name"). En este DTO, sobre el método getName se encuentran las anotaciones:

- @NotNull para indicar que se trata de un campo obligatorio.
- @Size(min = 3, max = 100) para indicar que el tamaño ha de estar comprendido entre 3 y 100 caracteres.

A continuación se muestra el código comentado:

```
public class CreateStationParamsDto {

  private String name;

  public CreateStationParamsDto() { }

  @NotNull
  @Size(min = 3, max = 100)
  public String getName() {
    return name;
  }
  ...
}
```

Gestión de excepciones

Spring-web ofrece varios mecanismos para que un método de un controlador pueda devolver un código de respuesta y una representación en el cuerpo de la respuesta. Cuando la ejecución de un método de un controlador puede devolver una excepción, spring-web permite indicar el código de la respuesta y devolver información sobre el error en el cuerpo de la respuesta.

En la implementación de los métodos de los controladores no se tratan las excepciones devueltas por los métodos de los servicios locales; se declaran en la cláusula throws de los métodos de los controladores y por medio de anotaciones se especifica cómo tratar cada excepción. Por cada excepción se define un método que la trata, anotado con:

- @ExceptionHandler: hace que spring-web invoque al método cuando alguno de los métodos del controlador lance la excepción especificada.
- @ResponseStatus: especifica el código de estado de la respuesta.

- @ResponseBody: es el contenido del cuerpo que devolverá el método.

En el siguiente fragmento de código se han utilizado estas anotaciones para gestionar la excepción StationAlreadyBelongsToLineException, producida cuando se intenta añadir una estación a una línea de la que ya forma parte.

```
@ExceptionHandler(StationAlreadyBelongsToLineException.class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ResponseBody
public ErrorsDto handleStationAlreadyBelongsToLineException( ...
  ) {
   ...
}
```

6.2 Implementación del frontend

El frontend se ha creado a partir de create-react-app. A continuación se describe el proceso de implementación que se ha seguido, basado en el diseño previo, y las consideraciones que se han tenido en cuenta para el desarrollo de sus dos capas: la capa IU y la capa de acceso a servicios.

6.2.1 Capa IU

En este apartado se muestran algunos detalles de implementación de la capa IU: la estructura común de los módulos y la especificación de propiedades a nivel de componente.

Estructura de los módulos

En cada módulo, se encuentran:

- Componentes: los componentes propios del módulo.
- actions.js, actionTypes.js, reducer.js y selectors.js: ficheros utilizados para gestionar el estado del módulo.
- index.js: define lo que se exporta del módulo.

Además, existe un reductor raíz que combina los reductores de los módulos y produce un estado formado por todos los estados de los distintos módulos.

Especificación de propiedades

Se ha utilizado la librería prop-types para especificar qué propiedades hay que pasarle a un componente cuando se usa. Por ejemplo:

- El componente LineLink es un enlace que lleva a los detalles de una línea. Para ello, necesita dos propiedades:
  - id: es el ID de la línea, de tipo number.
  - name: es el nombre de la línea, de tipo string.
- El componente Trips muestra una lista de viajes. Requiere la propiedad trips, que es la lista de viajes que se va a mostrar, de tipo array.

6.2.2 Capa de acceso a servicios

En esta capa, existe un fichero por cada controlador REST del backend, que define las funciones correspondientes con los casos de uso de dicho controlador ocultando los detalles de implementación.

Peticiones HTTP

Para el envío de peticiones HTTP al backend se utiliza la API Fetch [8], que está soportada de forma nativa por los navegadores modernos. Mediante el método fetch() se solicitan recursos de forma asíncrona a través de la red. Este método devuelve una promesa que se cumple cuando la respuesta está disponible. Una promesa solo se rechaza cuando se produce un error de red, pero no cuando se rechaza la petición por otro tipo de errores.

Se ha empleado un wrapper sobre la función fetch de la API Fetch, con la siguiente firma: const appFetch = (path, options, onSuccess, onErrors) donde:

- path se emplea para definir la URL.
- options indica el tipo de petición y puede incluir un objeto cuya representación JSON se incluirá en el cuerpo.
- onSuccess (opcional) es la función que se invocará si el código de estado de la respuesta es 2xx.
- onErrors (opcional) es la función que se invocará si el código de estado de la respuesta es 4xx y el cuerpo de la respuesta incluye información sobre el error.

Acciones asíncronas

Cuando se realiza una petición al backend, ésta es asíncrona; es decir, el código que hace la petición no se queda bloqueado esperando hasta que llegue la respuesta. De forma genérica, se pueden desear emitir tres acciones al reductor:

- Informar del comienzo de la petición.
- Informar de que la petición terminó correctamente.
- Informar de que la petición terminó con un error.

Las dos últimas acciones no se emitirían de forma inmediata. Para la ejecución de acciones asíncronas, se utiliza la librería Redux Thunk. Gracias a esta librería, un action creator puede devolver una función en lugar de un objeto. La acción asíncrona contiene la lógica de emisión de acciones internas.

Validaciones en el lado cliente

Para evitar el envío de peticiones innecesarias al backend, se realizan algunas validaciones en el lado cliente utilizando el soporte de validación de formularios de HTML5. Por ejemplo, para validar la tarjeta de crédito, en el formulario correspondiente se incluye:

- required para indicar que se trata de un campo obligatorio.
- pattern="[0-9]16" para indicar que es un campo formado por 16 dígitos entre el 0 y el 9.

Es importante aclarar que, independientemente de las validaciones realizadas en el lado cliente, el backend tiene que ejecutar todas las validaciones, tanto las que se pueden ejecutar en el cliente como las que no.

Gestión de errores devueltos por el backend

Los errores devueltos por el backend se mantienen en el estado local del componente correspondiente, ya que no son de interés para otros componentes. Si la respuesta del backend devuelve errores, el componente se vuelve a renderizar y como parte del a renderización se imprimen los errores.

6.3 Otras cuestiones de implementación

En este apartado se comenta cómo se han gestionado en la aplicación la internacionalización y la autenticación, autorización y control de acceso de usuarios.

6.3.1 Internacionalización

En el backend se trata la internacionalización de los mensajes en inglés, gallego y castellano. En el frontend se tratan, además, otros aspectos como el formateo de fechas.

Con Spring, se puede realizar la internacionalización de mensajes externalizando los mismos en ficheros de propiedades en los que se especifica la clave y el texto del mensaje. Se utiliza un bean de tipo MessageSource para recuperar un mensaje en un locale concreto a partir de su clave. En cuanto a los mensajes correspondientes a las reglas de validación, la implementación utilizada de la API de Beans Validation incluye los mensajes en inglés y español. Los mensajes en gallego se han incluido en un fichero de propiedades.

En el frontend, se han utilizado varios componentes de React Intl, por ejemplo: FormattedMessage para los mensajes, FormattedDate para las fechas, FormattedTime para las horas y FormattedNumber para las cantidades numéricas. Se han incluido tres ficheros de mensajes para los mensajes en inglés, gallego y castellano, respectivamente.

6.3.2 Autenticación, autorización y control de acceso

JSON Web Token (JWT) define un formato compacto para transmitir información en JSON de forma segura entre dos partes. Algunas peticiones requieren conocer el id del usuario, por lo que es necesario enviarla desde el frontend hasta el backend de forma segura. Para ello, se ha empleado JWT. El formato es el siguiente: cabecera.cuerpo.firma, donde:

- La cabecera especifica el tipo de token y el algoritmo de firma.
- El cuerpo contiene información sobre una entidad (en este caso, el usuario).
- La firma se obtiene a partir de la cabecera y el cuerpo.

El JWT se guarda utilizando el almacenamiento web proporcionado por el navegador, en sessionStorage. De esta forma, se elimina cuando se cierra la página, pero no cuando se recarga.

Para la autenticación de usuarios, se recupera la entidad user a partir de userName. Se cifra la contraseña y se comprueba si coincide con el valor de user.getPassword().

Para la autorización, se genera un JWT firmado con un algoritmo de clave privada. En el cuerpo del token se incluyen la fecha de expiración del token, el ID del usuario y el rol del usuario.

El control de acceso se realiza con Spring Security [3] mediante ficheros de configuración.

Capítulo 7

Pruebas

Para probar el correcto funcionamiento de las distintas partes de la aplicación y del conjunto de la misma, se han realizado distintos tipos de pruebas: unitarias, de integración y funcionales, que se describen a continuación.

7.1 Pruebas unitarias

Se han realizado pruebas unitarias para los métodos de lógica de negocio con los que cuentan las entidades, mencionados anteriormente. No se han realizado pruebas unitarias al resto de métodos de las entidades y a los DAOs que acceden a la BD, ya que se trata de código muy sencillo o generado automáticamente y se prueba de forma indirecta en las pruebas de integración.

7.2 Pruebas de integración

Se ha utilizado JUnit para implementar las pruebas de integración de cada uno de los servicios. Como se ha mencionado, también prueban de forma indirecta el funcionamiento de los DAOs y las entidades.

Hay una clase de pruebas por cada servicio, y cada caso de prueba se implementa en un método anotado con @Test. Con la librería spring-test, es posible implementar ágilmente estos clases de prueba.

Se ha utilizado la anotación @SpringBootTest a nivel de clase, que permite:

- Que las pruebas se ejecuten con JUnit 5.
- Hacer uso de las características propias de Spring Boot.
- Inyectar beans en las clases de pruebas.

También se ha utilizado la anotación @Transactional a nivel de clase. Con spring-test, cada método que lleve esta anotación termina con rollback; es decir, las modificaciones que haya hecho ese caso de prueba en la BD se deshacen automáticamente. De esta forma:

- Cada caso de prueba se ejecuta en una transacción.
- Se garantiza que los casos de prueba sean independientes entre sí.
- No es necesario eliminar los datos creados.

A continuación se muestra un fragmento de código correspondiente al test del servicio de información, donde se emplean las anotaciones mencionadas: @SpringBootTest y @Transactional a nivel de clase y @Text en el método que implementa el caso de prueba de buscar todas las estaciones.

```
1  @SpringBootTest
2  @ActiveProfiles("test")
3  @Transactional
4  public class InformationServiceTest {
5
6    ...
7
8    @Autowired
9    private StationDao stationDao;
10
11   ...
12
13   private Station createStation(int number) {
14     Station station = new Station("station" + number);
15     stationDao.save(station);
16     return station;
17   }
18
19   ...
20
21   @Test
22   public void findAllStationsTest() {
23
24     List<Station> foundStations =
       informationService.findAllStations();
25     List<Station> expectedStations = Arrays.asList();
26
27     assertEquals(expectedStations, foundStations);
28
29     Station station1 = createStation(1);
30     Station station4 = createStation(4);
```

```
31      Station station3 = createStation(3);
32      Station station2 = createStation(2);
33
34      foundStations = informationService.findAllStations();
35      expectedStations = Arrays.asList(station1, station2, station3,
        station4);
36
37      assertEquals(expectedStations, foundStations);
38    }
39
40    ...
41
42  }
```

7.3 Pruebas funcionales

El objetivo de este tipo de pruebas es verificar el correcto funcionamiento de la aplicación en su conjunto, utilizando directamente la interfaz como haría el usuario final. Normalmente, cada caso de prueba se corresponde con una secuencia de navegación del usuario, que es una funcionalidad concreta de la aplicación. Son más completas que las anteriores, aunque más complejas de realizar.

Estas pruebas se han realizado de forma manual. Para ello, se han probado todos los casos de uso de la aplicación y sus correspondientes excepciones. En la medida de lo posible, se ha intentado:

- Probar varias veces un mismo caso de uso, siguiendo la misma secuencia o distinta secuencia. Por ejemplo, se ha buscado varias veces un viaje con el mismo origen, el mismo destino y la misma fecha, completando esos tres campos en el mismo orden en algunas ocasiones y en distinto orden en otras.

- Probar varios casos de uso en órdenes distintos. Por ejemplo, se han realizado varias veces estas operaciones en distintos órdenes, comprobando siempre que los resultados obtenidos fuesen coherentes: iniciar sesión, comprar un billete, cerrar sesión, iniciar sesión y mostrar el historial de compra. Si el usuario no ha iniciado sesión, no se le permite comprar. Si el usuario ya ha iniciado sesión, no la puede iniciar de nuevo. Al mostra el historial de compra, aparecen todas las compras de la sesión actual y de las sesiones anteriores.

Capítulo 8

Planificación y evaluación de costes

En este capítulo se detalla la planificación temporal de cada uno de los sprints, incluyendo su duración en semanas y las horas de esfuerzo dedicadas. Partiendo de esta planificación, se estima el coste del proyecto.

8.1 Planificación

El tiempo dedicado a la realización del proyecto ha sido de 31 semanas (febrero-septiembre de 2020). Siguiendo la filosofía de Scrum, la duración de los sprints ha sido constante. Como hay 10 sprints, se ha establecido que la duración de cada sprint sea de 3 semanas. Esto hace un total de 30 semanas de trabajo, por lo que queda 1 semana para márgenes y descansos.

Para estimar la duración del proyecto, se ha seguido el criterio de que un punto de historia equivale a 1 hora y 30 minutos de trabajo. El proyecto está formado por 274 puntos de historia, lo que equivale a 411 horas de trabajo en total.

Por circunstancias externas al proyecto (académicas, personales, etc.), el esfuerzo en cada sprint no ha sido constante y se han dejado para el final los sprints con más esfuerzo.

Para desarrollar cada historia de usuario, se han llevado a cabo actividades de análisis, diseño, implementación y pruebas. Se estima que el análisis supone un 10% del esfuerzo, el diseño un 30% , la implementación un 30% y las pruebas un 30% .

En el sprint 0 se realiza un análisis general del proyecto, que se ha estimado que representa el 70% de todo el análisis realizado (es decir, un 7% del esfuerzo total del proyecto). En cada sprint se realizan el análisis restante (el 30% no incluido en el análisis general), el diseño, la implementación y las pruebas de las historias de usuario correspondientes.

8.1.1 Sprint 0

En el sprint 0 se realiza el análisis general del proyecto, que incluye a todas las historias de usuario y supone un 7% del esfuerzo total del proyecto; es decir, 28 horas y 46 minutos.

Además, se implementan las historias de usuario 19, 20, 21, 22, que suman un total de 32 minutos en tareas de análisis, 5 horas y 24 minutos en tareas de diseño, 5 horas y 24 minutos en tareas de implementación y 5 horas y 24 minutos en tareas de pruebas.

El tiempo total de esfuerzo en este sprint es de 45 horas y 31 minutos, repartidos a lo largo de 3 semanas.

8.1.2 Sprint 1

En el sprint 1 se implementan las historias de usuario 1 y 2, que suman un total de 54 minutos en tareas de análisis, 9 horas en tareas de diseño, 9 horas en tareas de implementación y 9 horas en tareas de pruebas.

El tiempo total de esfuerzo en este sprint es de 27 horas y 54 minutos, repartidos a lo largo de 3 semanas.

8.1.3 Sprint 2

En el sprint 2 se implementa la historia de usuario 3, que se descompone en 54 minutos en tareas de análisis, 9 horas en tareas de diseño, 9 horas en tareas de implementación y 9 horas en tareas de pruebas.

El tiempo total de esfuerzo en este sprint es de 27 horas y 54 minutos, repartidos a lo largo de 3 semanas.

8.1.4 Sprint 3

En el sprint 3 se implementan las historias de usuario 10, 11 y 14, que se suman un total de 57 minutos en tareas de análisis, 9 horas y 27 minutos en tareas de diseño, 9 horas y 27 minutos en tareas de implementación y 9 horas y 27 minutos en tareas de pruebas.

El tiempo total de esfuerzo en este sprint es de 29 horas y 18 minutos, repartidos a lo largo de 3 semanas.

8.1.5 Sprint 4

En el sprint 4 se implementan las historias de usuario 12 y 13, que suman un total de 57 minutos en tareas de análisis, 9 horas y 27 minutos en tareas de diseño, 9 horas y 27 minutos en tareas de implementación y 9 horas y 27 minutos en tareas de pruebas.

El tiempo total de esfuerzo en este sprint es de 29 horas y 18 minutos, repartidos a lo largo de 3 semanas.

8.1.6 Sprint 5

En el sprint 5 se implementa la historia de usuario 15, que se descompone en 1 hora y 48 minutos en tareas de análisis, 18 horas en tareas de diseño, 18 horas en tareas de implementación y 18 horas en tareas de pruebas.

El tiempo total de esfuerzo en este sprint es de 55 horas y 48 minutos, repartidos a lo largo de 3 semanas.

8.1.7 Sprint 6

En el sprint 6 se implementa la historia de usuario 4, que se descompone en 54 minutos en tareas de análisis, 9 horas en tareas de diseño, 9 horas en tareas de implementación y 9 horas en tareas de pruebas.

El tiempo total de esfuerzo en este sprint es de 27 horas y 54 minutos, repartidos a lo largo de 3 semanas.

8.1.8 Sprint 7

En el sprint 7 se implementa la historia de usuario 5, que se descompone en 1 hora y 48 minutos en tareas de análisis, 18 horas en tareas de diseño, 18 horas en tareas de implementación y 18 horas en tareas de pruebas.

El tiempo total de esfuerzo en este sprint es de 55 horas y 48 minutos, repartidos a lo largo de 3 semanas.

8.1.9 Sprint 8

En el sprint 8 se implementan las historias de usuario 6, 7, 16, 17 y 18, que suman un total de 1 hora y 45 minutos en tareas de análisis, 17 horas y 33 minutos en tareas de diseño, 17 horas y 33 minutos en tareas de implementación y 17 horas y 33 minutos en tareas de pruebas.

El tiempo total de esfuerzo en este sprint es de 54 horas y 24 minutos, repartidos a lo largo de 3 semanas.

8.1.10 Sprint 9

En el sprint 9 se implementan las historias de usuario 8, 9 y 23, que suman un total de 1 hora y 51 minutos en tareas de análisis, 18 horas y 27 minutos en tareas de diseño, 18 horas y 27 minutos en tareas de implementación y 18 horas y 27 minutos en tareas de pruebas.

El tiempo total de esfuerzo en este sprint es de 57 horas y 12 minutos, repartidos a lo largo de 3 semanas.

Sprint	Análisis (h)	Diseño (h)	Impl. (h)	Pruebas (h)	Total (h)
0	29,31	5,4	5,4	5,4	45,51
1	0,9	9	9	9	27,9
2	0,9	9	9	9	27,9
3	0,945	9,45	9,45	9,45	29,295
4	0,945	9,45	9,45	9,45	29,295
5	1,8	18	18	18	55,8
6	0,9	9	9	9	27,9
7	1,8	18	18	18	55,8
8	1,755	17,55	17,55	17,55	54,405
9	1,845	18,45	18,45	18,45	57,195
Total	41,1	123,3	123,3	123,3	411

Tabla 8.1: Horas dedicadas al proyecto

8.1.11 Total

El esfuerzo total es de 411 horas, que se descomponen en 41 horas y 6 minutos en tareas de análisis, 123 horas y 3 minutos en tareas de diseño, 123 horas y 3 minutos en tareas de implementación y 123 horas y 3 minutos en tareas de pruebas.

En la tabla 8.1 se recoge la descomposición de las horas dedicadas al proyecto en los distintos sprints y según la actividad (análisis, diseño, implementación y pruebas).

8.2 Evaluación de costes

Los recursos humanos dedicados al proyecto son los siguientes:

- Análisis, diseño, implementación y pruebas: Cristina Renda López.
- Dirección: Juan Raposo Santiago.

Para calcular los costes de los recursos humanos, se ha consultado el Estudio salarial del Sector TI en Galicia 2015-2016 [9], donde se han tomado las siguientes elecciones:

- Dirección: Jefe de proyecto Software (Provincia de A Coruña) experto, 42,00 € / hora.
- Análisis y diseño: Analista Programador Java (Provincia de A Coruña), 23,00 € / hora.
- Implementación y pruebas: Programador Java Web junior, 18,00 € / hora.

Las tareas de análisis y diseño suman un total de 164 horas y 24 minutos, lo que equivale a 3.781,20 €.

Las tareas de implementación y pruebas suman un total de 246 horas y 36 minutos, lo que equivale a 4.438,80 €.

Tarea	Recurso	Coste	Dedicación	Total
Análisis y diseño	Analista	23,00 €/h	164,4 h	3.781,20 €
Implementación y pruebas	Programador junior	18,00 €/h	246,6 h	4.438,80 €
Dirección	Jefe de proyecto	42,00 €/h	20 h	840,00 €
			Total	9.060,00 €

Tabla 8.2: Costes del proyecto

Se estima que el director ha dedicado al proyecto 2 horas por cada sprint, lo que hace un total de 20 horas y equivale a 840,00 €.

En cuanto a los recursos materiales, se ha utilizado un ordenador personal propiedad de la alumna. No se ha imputado su coste ya que habría que calcular la parte proporcional de uso respecto a la vida útil y no supondría un gran cambio con respecto al coste total del proyecto.

Por lo tanto, el coste estimado total sin IVA del proyecto es de 9.060,00 €. En la tabla 8.2 se recoge la descomposición de los costes del proyecto comentada.

Aplicando un IVA del 21%, el coste estimado total sería de 10.962,60 €.

Capítulo 9

Funcionalidades más destacadas

En este capítulo se explican las principales funcionalidades de la aplicación desde el punto de vista del usuario.

9.1 Búsqueda de viajes

El usuario elige una estación de origen, una estación de destino y la fecha del viaje. Puede dejar en blanco una de las estaciones, pero no ambas. De esta forma, es posible ver todos los trenes que salen o llegan un día a una estación. Se devuelven todos los viajes que cumplen los criterios indicados por el usuario. Es importante aquí, desde el punto de vista de la implementación, que se devuelve solo la información correspondiente al trayecto seleccionado por el usuario, y no la del trayecto completo que realiza el tren.

9.2 Compra de billetes, cambios y devoluciones

El usuario elige el trayecto deseado, obtenido como resultado de la búsqueda de viajes. A continuación, se le muestran los asientos disponibles, y elige los que desee. En caso de que lo desee, puede seleccionar también un trayecto de vuelta y elegir los asientos correspondientes. Por último, introduce los datos de su tarjeta de crédito para completar la compra. Al finalizar la compra, se le muestran los datos de la compra, donde se incluyen los localizadores de los billetes que acaba de adquirir (un billete por cada asiento y cada trayecto).

Un viajero puede devolver un billete que haya adquirido previamente, siempre y cuando queden más de dos horas para la realización del viaje. También puede cambiar un billete, siempre y cuando quede más de media hora para la realización del viaje. Para ello debe seleccionar el billete que desea cambiar y elegir un nuevo viaje, siguiendo un proceso similar al de compra.

9.3 Visualización de las compras realizadas y de los billetes

Los usuarios registrados en la aplicación pueden ver su historial de compras, donde aparece un resumen de sus compras ordenadas por fechas, de más a menos reciente. En este caso se utiliza paginación, de forma que no todas las compras del usuario se recuperan a la vez de la base de datos de la aplicación

Desde el historial de compra se pueden ver los detalles de una compra, donde se muestran los billetes adquiridos en ella. Se puede elegir un billete para ver sus detalles: fecha, origen, destino, etc. En esta información se incluyen las nuevas horas de las paradas en caso de que haya modificaciones. También es posible buscar directamente un billete a través de su localizador.

9.4 Gestión de líneas y estaciones

Existen varias funcionalidades relacionadas con la gestión de líneas y estaciones, reservadas para los administradores de la compañía ferroviaria. Es posible consultar los detalles de líneas y estaciones, crear una nueva estación y añadir una estación a una línea existente.

9.5 Gestión de viajes

Los usuarios con rol de administrador pueden crear nuevos viajes. Para ello, seleccionarán una línea e indicarán la fecha, las horas de parada en cada estación, el tren que realizará el viaje y la frecuencia de circulación. Tras la creación, los nuevos viajes se incorporarán a la base de datos de la aplicación.

También pueden modificar la hora de las paradas de los viajes ya existentes. En este caso, indicarán la nueva hora de parada y, si lo desean, el motivo del cambio. En la visualización de un billete, en caso de que la hora de su parada de origen o destino haya sido modificada, aparece la información relacionada (es decir, la nueva hora de parada y el motivo del cambio).

Por último, pueden añadir nuevos coches a los trenes en circulación, para aumentar la capacidad de los mismos.

9.6 Estadísticas

Los administradores pueden consultar distintos datos relacionados con las ventas y los viajeros, como los ingresos totales, las devoluciones realizadas en determinado período de tiempo o el número de viajeros que ha circulado entre dos estaciones. Estos datos se generan en función de distintos parámetros que elige el usuario: el período temporal y, en el caso de los viajeros, las estaciones de origen y/o destino.

Capítulo 10

Conclusiones

Tras la realización del proyecto, se ha realizado un pequeño análisis en el que se contrastan los objetivos planteados inicialmente con lo realizado en el proyecto. Se incluyen también las lecciones aprendidas durante el desarrollo y las posibles líneas futuras de la aplicación.

10.1 Concordancia de objetivos

A continuación se citan los objetivos iniciales de la aplicación, y se comenta en qué grado y por qué motivos se considera que se ha alcanzado cada uno de ellos.

- Afianzar los conocimientos con las tecnologías empleadas. Utilizando de nuevo tecnologías ya empleadas previamente, se ha conseguido una mayor comprensión del funcionamiento de las mismas. Se ha adquirido práctica y familiaridad con las mismas, que se traducen en una mayor facilidad, flexibilidad y rapidez a la hora de programar. Previsiblemente, este conocimiento en mayor profundidad implique más facilidad a la hora de aprender tecnologías distintas.

- Hacer un buen diseño de la aplicación. Se han seguido los patrones de diseño estudiados a lo largo de la carrera. Sin embargo, durante la implementación se han detectado algunos aspectos del diseño que se podían mejorar y que se han corregido. Al emplear una metodología ágil, estas correcciones no han tenido gran repercusión en la planificación, pero sí han servido para adquirir más conocimientos, especialmente orientados a la práctica, sobre las técnicas de diseño empleadas.

- Realizar una buena planificación y seguirla durante el desarrollo. Este aspecto ha sido fundamental, ya que al ser un proyecto de larga duración en comparación con los realizados hasta el momento, tener fechas de entrega intermedias ha contribuido en gran

medida a llevar un ritmo de desarrollo más o menos constante. Las semanas de descanso planificadas también han sido necesarias y relevantes.

- Desarrollar una aplicación funcional. Sí se ha conseguido una aplicación que implementa las principales funcionalidades planteadas. Sin embargo, es un ejemplo académico y tendría algunos aspectos a mejorar para su utilización en un contexto real. Esto se debe, principalmente, a que inicialmente se habían planeado demasiadas funcionalidades y se han tenido que reducir algunas de ellas porque harían el proyecto excesivamente largo.

10.2 Lecciones aprendidas

A continuación se explican las principales lecciones aprendidas durante el desarrollo del proyecto.

- Importancia de la planificación. La elección de la metodología se consideraba inicialmente un aspecto poco relevante, pero ciertamente condiciona el desarrollo del proyecto. El hecho de tener fechas de entrega intermedias ha facilitado que se siga un desarrollo constante a lo largo del tiempo.

- Al realizar un proyecto de forma individual, a diferencia de la gran mayoría de los realizados hasta el momento, es obligatorio conocer cada pequeño detalle del proyecto. Esto se ha traducido en una mayor comprensión global del mismo.

- A lo largo de la carrera nos han insistido en gran cantidad de ocasiones en que el cliente no tiene claro lo que quiere. Aunque he realizado muchos trabajos, siempre había un enunciado al que ceñirse, o una persona a la que preguntarle en caso de duda. En esta ocasión, yo he puesto el enunciado y efectivamente, en gran cantidad de ocasiones no tenía claro lo que quería. Por lo tanto, he experimentado en primera persona lo que significa ser el cliente y no saber lo que quiere, y también me he enfrentado como programadora a un cliente "perdido".

- Todos los dominios en los que había trabajado previamente eran similares entre ellos, normalmente basados en la gestión de un almacén de productos. El hecho de elegir un dominio distinto ha hecho que adquiera mayores conocimiento sobre los patrones de diseño aplicados. Por otra parte, el dominio es más complejo y ha ralentizado la implementación, haciendo que el número de funcionalidades implementadas sea menor que en un dominio más sencillo.

10.3 Líneas futuras

En este apartado se mencionan algunas ampliaciones que se podrían realizar, otras que estaban previstas y que finalmente no se implementaron, y otras que no se incluyeron debido a que no formaban parte de los objetivos del proyecto.

10.3.1 Grupos de edad

En un principio, estaba planeado incluir grupos de edad en la aplicación. De esta forma, cada billete tendría un precio distinto según el grupo de edad al que perteneciese el viajero. Esta sería la implementación más realista de la aplicación y la que, por supuesto, se llevaría a cabo en un caso real. Sin embargo, se ha considerado que es poco interesante desde el punto de vista académico por varios motivos:

- Un mismo usuario puede comprar billetes para distintos viajeros. Por lo tanto, no hay forma de comprobar si los grupos de edad que indica el usuario son los que realmente realizarán el viaje posteriormente.
- Para incluir esta funcionalidad, simplemente habría que modificar el precio final de cada billete aplicando el descuento correspondiente al grupo de edad del viajero.

En conclusión, complicaría la implementación de la aplicación sin aportar ningún valor añadido a la misma, por lo que no se ha realizado.

10.3.2 Gestión de la flota de trenes

Durante el desarrollo de la aplicación, se ha considerado que sería muy interesante incluir funcionalidades relativas a la gestión de trenes. Por ejemplo:

- Saber qué trayectos realiza un tren en un día, con detalles sobre los horarios.
- Comprobar qué trenes están disponibles para realizar un trayecto fuera de los horarios habituales; por ejemplo, por una incidencia inesperada o un evento extraordinario.
- Gestionar el mantenimiento de los trenes, asignándoles sustitutos para la realización de los viajes durante los tiempos de mantenimiento.

Estas funcionalidades serían muy útiles desde el punto de vista de los administradores de la compañía ferroviaria, y podrían incorporarse a la aplicación en un futuro. Sin embargo, no estaban dentro de los objetivos y quedan fuera del alcance de la aplicación.

10.3.3 Mejoras en la experiencia de usuario

En mi opinión, se podría realizar una gran mejora en la funcionalidad de mostrar los asientos disponibles para un viaje, ya que se muestra una lista de los asientos en forma tabular, pero no se muestra gráficamente la localización de dichos asientos, que sería mucho más interesante desde el punto de vista del usuario. Lo mismo sucede en el caso de las estadísticas; la visualización gráfica de las mismas haría que el usuario las percibiese de una forma mucho más rápida e intuitiva.

10.3.4 Promociones y descuentos

Con un conocimiento más profundo de los movimientos de viajeros, que se podría obtener gracias a la aplicación, se podrían hacer descuentos por viajar en horas de mínima demanda, que resultarían beneficiosos para la compañía e interesantes para los viajeros. Además, se podrían hacer promociones adaptadas a cada viajero. Por ejemplo, por viajar por mucha frecuencia o por la acumulación de puntos.

10.3.5 Clases de viajeros

Sería posible implementar distintas clases de viajeros, como turista y preferente, que se le asignarían a cada asiento de un tren. El precio del billete variaría en función de esta característica.

10.3.6 PMR

Los trenes cuentan con algunas plazas reservadas para PMR, que no se han contemplado en el desarrollo de la aplicación. Además de contemplarlas, sería interesante la posibilidad de que otros viajeros ocupen una plaza reservada para PMR en el caso de que ésta no vaya a ser utilizada.

Apéndices

Lista de acrónimos

API *Application Programming Interface, Interfaz de programación de aplicaciones.*

BD *Base de datos.*

CSS *Cascading Style Sheets, hojas de estilo en cascada.*

CRUD *Create Read Update Delete, operaciones básicas de acceso a datos (crear, leer, actualizar y eliminar).*

DAO *Data Access Object, objeto de acceso a datos.*

DOM *Document Object Model, modelo de objetos del documento.*

DTO *Data Transfer Object, objeto de transferencia de datos.*

HTML *HyperText Markup Language, lenguaje de marcas de hipertexto.*

HTTP *HyperText Transfer Protocol, protocolo de transferencia de hipertexto.*

ID *Identificador único.*

IU *Interfaz de usuario.*

JDBC *Java Database Connectivity, conectividad a bases de datos de Java.*

JPA *Java Persistence API, API de persistencia de Java.*

JSON *JavaScript Object Notation, notación de objeto de JavaScript.*

JSX *JavaScript XML.*

MVC *Modelo-Vista-Controlador.*

PMR *Persona con Movilidad Reducida.*

POM *Project Object Model.*

REST *REpresentational State Transfer, Transferencia de Estado Representacional.*

SPA *Single Page Application, aplicación de página única.*

SQL *Structured Query Language, lenguaje de consulta estructurada.*

XML *eXtensible Markup Language, lenguaje de marcado extensible.*

Bibliografía

[1] Spring data jpa. [En línea]. Disponible en: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference

[2] Documentación de spring. [En línea]. Disponible en: https://spring.io/

[3] Spring security. [En línea]. Disponible en: https://spring.io/projects/spring-security

[4] Documentación de react. [En línea]. Disponible en: https://es.reactjs.org/docs/getting-started.html

[5] Documentación de redux. [En línea]. Disponible en: https://es.redux.js.org/

[6] Hooks api reference. [En línea]. Disponible en: https://reactjs.org/docs/hooks-reference.html

[7] Rules of hooks. [En línea]. Disponible en: https://reactjs.org/docs/hooks-rules.html

[8] Uso de fetch. [En línea]. Disponible en: https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Utilizando_Fetch

[9] Guia salarial sector ti galicia 2015-2016. [En línea]. Disponible en: https://es.scribd.com/document/288511179/Guia-Salarial-Sector-TI-Galicia-2015-2016